

**ESCUELA POLITÉCNICA SUPERIOR DE MONDRAGON
UNIBERTSITATEA**
MONDRAGON UNIBERTSITATEKO GOI ESKOLA POLITEKNIKOA
MONDRAGON UNIVERSITY FACULTY OF ENGINEERING

Trabajo presentado para la obtención del título de

Titulua eskuratzeko lana

Final degree project for taking the degree of

GRADO EN INGENIERÍA EN INFORMÁTICA
INFORMATIKAKO INGENIARITZA GRADUA
DEGREE IN COMPUTER ENGINEERING

Título del Trabajo *Lanaren izenburua* Project Topic

**RUNTIME VERIFICATION FOR SPATIO-TEMPORAL PROPERTIES OVER
IOT NETWORKS**

Autor *Egilea* Author

OIHANA GARCIA ANACABE

Curso *Ikasturtea* Year

2021/2022

Título del Trabajo *Lanaren izenburua* Project Topic

**RUNTIME VERIFICATION FOR SPATIO-TEMPORAL
PROPERTIES OVER IOT NETWORKS**

Nombre y apellidos del autor

Egilearen izen-abizenak

Author's name and surnames

GARCIA ANACABE, OIHANA

Nombre y apellidos del/los director/es del trabajo

Zuzendariaren/zuzendarien izen-abizenak

Project director's name and surnames

EZIO BARTOCCI

ILLARRAMENDI, MIREN

Lugar donde se realiza el trabajo

Lana egin deneko lekua

Company where the project is being developed

TU WIEN

Curso académico

Ikasturtea

Academic year

2021/2022



El autor/la autora del Trabajo Fin de Grado, autoriza a la Escuela Politécnica Superior de Mondragon Unibertsitatea, con carácter gratuito y con fines exclusivamente de investigación y docencia, los derechos de reproducción y comunicación pública de este documento siempre que: se cite el autor/la autora original, el uso que se haga de la obra no sea comercial y no se cree una obra derivada a partir del original.

Gradu Bukaerako Lanaren egileak, baimena ematen dio Mondragon Unibertsitateko Goi Eskola Politeknikoari Gradu Bukaerako Lanari jendeaurrean zabalkundea emateko eta erreproduzitzeko; soilik ikerketan eta hezkuntzan erabiltzeko eta doakoa izateko baldintzarekin. Baimendutako erabilera honetan, egilea nor den azaldu beharko da beti, eragotzita egongo da erabilera komertziala baita lan originaletatik lan berriak eratortzea ere.

Abstract

Key Words:

Laburpena

(Laburpen amaieran ipini dokumentuaren amaierarantz informazio gehiago dagoela euskaraz *Erreferentzia bat sartu atal horretara)

Hitz gakoak:

Resumen

Palabras clave:

List of contents

1. Introduction	1
1.1. Problem definition	1
1.2. Introduction to the department	2
2. State of the art	3
2.1. Cyber Physical Systems	3
2.2. Monitors	4
2.3. Spatio-Temporal Logics	6
2.4. Middleware	7
2.5. Communication protocols	7
2.5.1. MQTT	7
2.5.2. Bluetooth Low Energy	8
2.5.3. Wi-Fi	8
2.6. Dashboards	9
3. Product definition	10
3.1. Objectives	10
3.2. Project phases	11
3.2.1. Introduction	11
3.2.2. Product development	11
3.2.3. Other tasks	12
3.2.4. Gantt Chart	12
3.3. Product requirements	13
3.3.1. Software	13
<i>Moonlight</i>	13
<i>ThingsBoard</i>	13
<i>GitHub Actions and SonarCloud</i>	13
<i>Programming languages</i>	14
<i>Gradle</i>	14
<i>Zephyr</i>	14
<i>Integrated Development Environment (IDE)</i>	14
3.3.2. Hardware	14
<i>Thingy52</i>	14
<i>Segger J-Link</i>	15
<i>ESP32</i>	15
4. Development	17

4.1.	Features of Moonlight monitor	17
4.2.	Monitor initialization and basic workflow	19
4.2.1.	Set up	19
	<i>Formula</i>	19
	<i>Atomic Propositions</i>	19
	<i>Spatial Model</i>	19
	<i>Distance Functions</i>	20
4.2.2.	Increment - Data conversion	20
	<i>Recording the received data</i>	20
	<i>Joining the values</i>	20
4.3.	Middleware	21
4.4.	Services.....	22
4.4.1.	Sensor service	23
4.4.2.	Moonlight service	24
	<i>Buffer</i>	24
	<i>Conversion – Message to Increment</i>	25
	<i>Moonlight monitor</i>	25
4.4.3.	Dashboard service.....	27
4.4.4.	Service Builders.....	29
4.5.	Data Bus (Horizontal messages)	30
4.6.	Client	31
4.7.	Physical system.....	31
4.7.1.	Thingy52.....	32
4.7.2.	ESP32	33
5.	Problems and solutions	35
5.1.	Dealing with null values.....	35
5.2.	Managing the buffer.....	36
5.3.	Sensors time synchronization.....	36
5.4.	ESP32 problems.....	38
5.4.1.	Connections	38
5.4.2.	Memory	38
5.5.	Not solved problems	38
5.5.1.	Sensors error.....	38
5.6.	Other problems	38
6.	Use cases	40
6.1.	Office use case.....	40

7. Results	41
7.1. Efficiency	41
7.2. Robustness	41
8. Economic memory	42
9. Conclusions and future lines	43
9.1. Conclusions.....	43
9.2. Future lines.....	43
10. Personal evaluation	44
11. Gradu Bukaerako Lanaren laburpena (Basque summary).....	45
11.1. Sarrera	45
11.2. Ondorioak.....	45
11.3. Etorkizuneko ildoak	45
12. Appendix A Gantt chart	46
13. Bibliography	47

List of figures

Figure 1-A Project outline	2
Figure 1-B TU Wien logo and library building	2
Figure 2-A CPS different spatial and temporal scales [15].....	4
Figure 3-A Resumed Gantt chart	12
Figure 3-B Hardware	16
Figure 4-A Features of the monitor	18
Figure 4-B Online Monitoring Model	19
Figure 4-C Tuple.java usage example.....	20
Figure 4-D Middleware Architecture	21
Figure 4-E SOA Diagram	22
Figure 4-F Data buffer	25
Figure 4-G Monitor service flow chart	26
Figure 4-H ThingsBoard example charts	27
Figure 4-I Middleware ThingsBoard connection	28
Figure 4-J Builder design pattern	29
Figure 4-K Physical system	32
Figure 5-A Missing values problem	35
Figure 5-B Missing values solution.....	35
Figure 5-C Time synchronization problem	37
Figure 5-D Register of the time of the messages	38

List of tables

Table 2-A Software monitors	5
Table 2-B Number of specification coverage on 1000 requirements [20]	6
Table 2-C Structure of an MQTT message	7
Table 2-D Classic Bluetooth vs BLE [25]	8
Table 2-E IEEE 802.11 technology comparison [27]	9
Table 4-A Example of a sensor message class	23
Table 4-B JSON to Message conversion	24
Table 4-C Monitor call	25
Table 4-D SpaceTimeSignal results in a String format	26
Table 4-E ThingsBoard connector abstract class	28
Table 4-F Singleton Data Bus class	30
Table 4-G Thingy52 prj.conf, Kernel file	33
Table 4-H Thingy read permissions	33
Table 4-I Thingy read callback	33
Table 4-J BLE connection characteristics	34
Table 5-A Thingy get time function	37

Acronyms

B

BLE: Bluetooth Low Energy.

C

CO₂: Carbon dioxide.

CPS: Cyber Physical Systems.

E

ESB: Enterprise Service Bus.

I

IoT: Internet of Things.

L

LTL: Linear Temporal Logic.

M

MFOTL: Metric First-Order Temporal Logic.

MQTT: Message Queue Telemetry Transport.

R

RTC: Real-Time Clock.

RV: Runtime Verification.

S

SaaS: Software as a Service.

SaSTL: Spatial Aggregation Signal Temporal Logic.

SOA: Service Oriented Architecture.

SpaTeL: Spatial-Temporal Logic.

SSTL: Signal Spatio-Temporal Logic.

STL: Signal Temporal Logic.

STREL: Spatio-Temporal Reach and Escape Logic.

T

TDD: Test Driven Development.

TVOC: Total Volatile Organic Compounds.

U

UUID: Universally Unique Identifier.

W

WEF: World Economic Forum.

1. Introduction

This chapter introduces the Bachelor's Degree Final Project "Runtime verification for spatio-temporal properties over IoT networks". In this section, the concepts involved in the project are defined.

1.1. Problem definition

IoT (**I**nternet **o**f **T**hings) is the area of computer science that collects the challenges of connecting millions of intelligent devices and sensors and making them accessible via the Internet. This field is growing fast. The forecast is that the number of connected devices by 2030 will be 25 billion worldwide [1]. Over the past few years, IoT has become one of the most important technologies of the 21st century. These devices are already part of several fields (e.g., e-health services, smart cities, e-farm, and intelligent transportation systems (ITS)). Through low-cost computing, the cloud, big data, analytics and mobile technologies, IoT is a big part of the digitalization of the society to build an intelligent world. The physical world is meeting the digital world and they cooperate [2].

Among the systems that can exploit an IoT infrastructure, a noteworthy category is **Cyber Physical Systems** (CPS), where a computational core monitors and controls physical systems. A definition of CPS refers to systems that interact with physical processes through sensors and actuators. The increasing numbers of IoT devices and intelligent systems made CPS influence society. CPS, an emerging innovative information and communications technology, can be found in different sectors such as self-driving cars, home equipment, wireless sensor networks and medical devices. CPS and IoT are reshaping how we perceive and interact with the physical world [3] [4]. The following definition is the most famous one for the term "Cyber Physical Systems":

"Cyber-Physical Systems are engineering, physical and biological systems whose operations are integrated, monitored, and/or controlled by a computational core. Components are networked at every scale. Computing is deeply embedded into every physical component, possibly even into materials. The computational core is an embedded system, usually demands real-time response, and is most often distributed. The behaviour of a cyber-physical system is a fully-integrated hybridisation of computational (logical) and physical action."

(Helen Gill, US National Science Foundation) [5]

Monitoring is an activity related to the broader category of **Runtime Verification** (RV), which refers to the act of observing and evaluating temporal behaviours. So, RV's purpose is to track information from a system while operating and analyze the behaviour to detect if it satisfies or

violates specific properties. Monitoring the status of a CPS at runtime can give the precise information to ensure reliability, safety, robustness and security [6] [7].

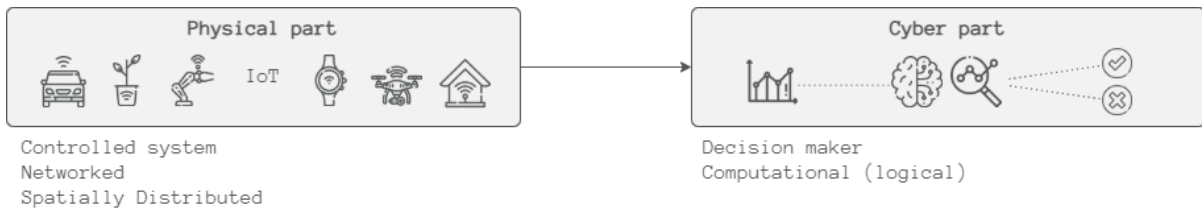


Figure 1-A Project outline

This project focuses precisely on the challenges when doing the monitoring on CPS over IoT. The project solves the difficulties of communicating sensors and interacting with various services together and provides an implementation of different services to be able to monitor at runtime. It is closely related to some aspects of Helen Gill's definition. The IoT devices are in the physical part, spatially distributed and networked. The data will be collected both across space and time. One main task of the project is to connect the sensors with the monitor so they can share information (i.e., networking). Finally, this data will be sent to Moonlight to monitor everything in real-time and receive a statement on the system's status.

1.2. Introduction to the department

The project is held at Technische Universität Wien (TU Wien), one of the largest universities in Austria. It currently has more than 27.000 students and 4.000 scientists and researchers. The university is highly regarded internationally and nationally in teaching and research and is a valued partner of innovation-oriented companies. The university's teaching and research focus on technology and natural sciences [9].

The project is held in the department of the Cyber-Physical System Group. This group was established in 2012. Nowadays, the main focuses are on: The specification (e.g., spatial-temporal logics), the design (e.g., probabilistic hybrid systems), the analysis (e.g., symbolic and stochastic model checking) and the control (e.g., PID, supervisory, and optimal) [10].

TU Wien has some buildings around Vienna, and the CPS department is in the library building.



Figure 1-B TU Wien logo and library building

2. State of the art

In the state of the art chapter, the technologies used in the project are analyzed to study the current level of development by searching and reading the previously done researches.

2.1. Cyber Physical Systems

CPS plays a vital role in Industry 4.0. The concept of the Fourth Industrial Revolution was introduced in 2010 by the German government. This is an interpretation of Klaus Schwab, the executive chairman of the World Economic Forum (WEF): *“Now a Fourth Industrial Revolution is building on the Third, the digital revolution that has been occurring since the middle of the last century. It is characterized by a fusion of technologies that is blurring the lines between the physical, digital, and biological spheres.”* [9]. The mechanical systems on their own are not relevant anymore; that is why CPS is widely applied in various fields. By providing the physical objects with computing and communication capabilities, they can turn into intelligent objects and surpass the previous systems. CPS can learn from the physical world and, in some cases, even interact with the world, turning environments into Smart Environments [10] [11]. In conclusion, CPS profoundly influences society and reshapes the perception and interaction with the physical world [4].

Nowadays, CPS is built together within some form of IoT architecture. The IoT applications collect enormous amounts of data from many places and then send them to a cloud [13]. These large amounts of data have no value if previously are not processed. That is why CPS enables monitoring to interpret the received values and detect any property violation. In CPS, it is essential to monitor the system’s behaviours to detect anomalies, avoid problems, improve the system and ensure that the defined properties are satisfied.

CPS is widely used in various contexts, such as smart cities, medical devices, aeronautics, automotive, traffic control systems, robotics, manufacturing, maintenance and water management. All these different fields share the same technological scenario; the physical world, components and sensors deeply connect to computational entities, and they interact. CPS is usually integrated as a system-of-systems communicating with each other and with the humans. Managing and monitoring such an ultra-large-scale system is becoming extremely challenging. For this reason, the design phase of the monitoring is being performed on simulations of a model to test different initial conditions, parameters and inputs [14].

Finally, the cyber and physical parts are so connected that they bring some disadvantages too. Due to the significant impact that CPS is having on our lives, failures and security vulnerabilities can cause fatal accidents. With the connection of a CPS to the Internet, security becomes a crucial factor [4].

2.2. Monitors

As previously mentioned, a fundamental task in CPS is monitoring their behaviours. CPSs are susceptible to faults, so there is a need to detect the status of the systems. Monitoring is an activity to observe systems' behaviour and enabling the runtime verification (RV) techniques; monitors can track systems while operating.

CPS has physical components that operate at different spatial and temporal scales. Therefore, the spatial and the temporal requirements are fundamentals for their safe and correct execution.

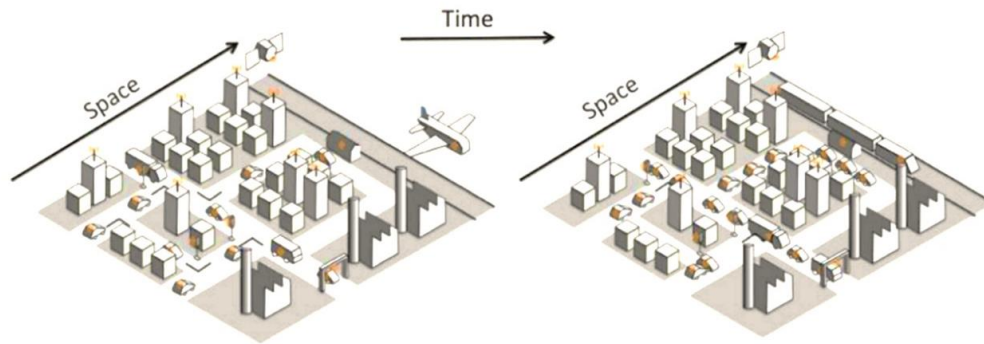


Figure 2-A CPS different spatial and temporal scales [15]

Monitoring spatio-temporal properties over CPS executions was first proposed in 2008 [16], when the author introduced the notion of a spatial-temporal event-based model for CPS. The generated events contained time and space stamps, and a monitor processed those events. However, currently, the available tools for monitoring formal specifications are restricted to temporal properties, ignoring, in most cases, the spatial dimension of CPS [17]. Temporal specification languages are not always expressive enough to capture the rich and complex spatio-temporal patterns of CPS [14].

As previously said, the monitor used in this project is **Moonlight**. Moonlight is a lightweight tool that can monitor both temporal and spatial scales of spatially distributed CPS. Thanks to its spatio-temporal properties, the distributed elements can move in space and change their connectivity (mobile CPS).

Moonlight is a software monitor solution. These are some related monitors used for CPS:

- **RuMoR**: A tool for temporal properties specified in Linear Temporal Logic (LTL). It enables non-intrusive (and scalable) online monitoring [18].
- **Online MonPoly**: In this work [6], they adapt the monitoring package MONPOLY (an offline verification tool) in order to be able to verify Metric First-Order Temporal Logic (MFOTL) properties in an online manner.
- **jSSTL**: This is a Java tool for the specification of Signal Spatio-Temporal Logic (SSTL) properties, able to control spatio-temporal behaviours. Using spatio-temporal logic is

more expressive and complex to analyze than standard temporal logic languages. jSSTL is a usable tool; the inconvenience is that it only supports offline monitoring [18].

- **Moonlight**: This monitor is a java tool for monitoring both temporal and spatio-temporal properties of distributed complex systems. Moonlight and its interfaces are still under development; however, it is sufficiently qualified to apply in real environments and, therefore, in this project. In this framework, space is represented as a weighted graph, describing the topological configurations in which the single CPS entities (nodes of the graph) are arranged. Both nodes and edges have attributes modelling physical and logical quantities that can change in time [14]. Furthermore, it supports both offline and online monitoring.

Table 2-A Software monitors

Monitor	Specification Languages	Properties	Online / Offline
RuMor	LTL (Linear Temporal Logic)	Temporal	Online
[6] Online MonPoly	MFOTL (Metric First-Order Temporal Logic)	Temporal	Online
jSSTL	SSTL (Signal Spatio-Temporal Logic)	Spatio-temporal	Offline
<u>Moonlight</u>	STREL (Spatio-Temporal Reach and Escape Logic)	Spatio-temporal	Online

Comparing the monitors from above, Moonlight has some advantages compared to the others. In the first place, it handles spatio-temporal properties. Most of the monitors that can be found nowadays support only the temporal properties. Actually, many developed prototypes built are more for demonstration purposes rather than becoming usable tools. The only usable tool for spatio-temporal properties found is jSSTL. Afterwards, Moonlight also supports online monitoring compared to jSSTL, the other monitor with spatio-temporal properties. It is common to find online monitoring with temporal properties, but Moonlight has that addition of working with spatio-temporal properties. Moreover, jSSTL operates over a static topological space, while the tool proposed in this paper can also monitor dynamical locations, such as in mobile wireless sensor networks.

In conclusion, Moonlight is the only tool capable of monitoring spatio-temporal properties of distributed systems in RV. However, this does not mean that the other monitors are not worthwhile. They have attributes that make them adequate for other case studies.

2.3. Spatio-Temporal Logics

The monitors should include a language to specify the requirements needed to monitor the system efficiently. Signal Temporal Logic (STL) is widely used for analyzing programs in CPSs. Following the earlier chapter, most specification languages and tools available for CPS support only the monitoring of temporal properties (e.g., MTL, STL and TRE). However, STL is not expressive enough for real-world cases because they require not only temporal information but also spatial needs. Thereby, many researchers have extended temporal specification languages such as STL to express also spatial requirements. These are some existing spatial extensions of STL: SSTL, SpaTeL, SaSTL and STREL [14][20].

In the following table, there is a comparison of the specification coverage on 1000 quantitatively specified actual city requirements between STL, SSTL, STREL and SaSTL:

Table 2-B Number of specification coverage on 1000 requirements [20]

	SaSTL	STREL	SSTL	STL
Number of covered requirements	950	431	431	184
Covered key elements	Temporal & Spatial Range & Aggregation, Counting, Percentage	Temporal & Spatial Range	Temporal & Spatial Range	Temporal

As we can see in Table 2-B Number of specification coverage on 1000 requirements , the specification language with the most coverage is SaSTL. Nevertheless, STREL, the language used with Moonlight, is expressive enough to cover the essential requirements.

Despite not being the language with most specification coverage, STREL has some conveniences that the other specifications do not have:

- STREL can handle online monitoring; this solution is preferable because it permits to take immediate action during execution and is computationally cheaper [15].
- STREL can handle mobile or dynamic CPS. In SSTL, the topology is assumed to be static; it is impossible to monitor nodes changing locations [16].

STREL functions mapping each node and time instant into a vector of values, describing the internal state of each location. In **¡Error! No se encuentra el origen de la referencia.**, there is more information about STREL and how to express the spatio-temporal behaviours in this specification language.

2.4. Middleware

A distributed system is composed of several hardware and software elements which must be integrated, the element that facilitates the management of such environments is the middleware. For example, IoT requires integrating and working with data and different algorithms distributed in other applications and operating in real-time with a wide variety of processes.

Distributed systems are not only applicable to computers and physical components but also a logical dimension. A *distributed system* can be defined as a set of independent processes or applications that interact with each other. The middleware is in charge of enabling the communication and data management of the distributed applications [17].

The middleware developed for this project communicates different services with each other in real-time; these services can be found locally or distributed.

2.5. Communication protocols

A communication protocol is required to exchange messages between computing systems. These are the communication standards used in the project:

2.5.1. MQTT

MQTT (Message Queue Telemetry Transport) is a standard messaging protocol widely used for the Internet of Things. It is ideal for connecting remote devices with low computational power and minimal network bandwidth. It is designed as an extremely lightweight publish/subscribe messaging transport, and thanks to this, it has become one of the most used IoT protocols.

Table 2-C Structure of an MQTT message

	0	1	2	3	4	5	6	7
Byte 1	Message Type				DUP	QoS Level		RETAIN
Byte 2	Remaining Length (1-4 bytes)							
	Optional: Variable Length Header							
	Optional: Variable Payload							

MQTT messages have a mandatory fixed-length header (2 bytes) and optional header and message payload. These optional fields usually complicate the protocol processing, but in some cases, they can help reduce data transmission as much as possible.

Moreover, MQTT can scale to connect with millions of IoT devices. Today is used in a wide variety of industries, such as automotive, manufacturing and telecommunications [23] (More information in **¡Error! No se encuentra el origen de la referencia.**).

2.5.2. Bluetooth Low Energy

Bluetooth Low Energy (BLE) is one of the most widely applicable low-power connectivity standards. It can be used in different situations, such as blood pressure monitoring, industrial monitoring sensors, Fitbit devices and other IoT applications. Wireless gadgets conforming from Bluetooth version 1.0 to 3.0 are called Bluetooth Classic devices. Version 4.0 (introduced in 2009) and onwards, BLE is established. BLE is intended to provide considerably reduced power consumption and cost while maintaining a similar communication range [19]. BLE does not replace Classic Bluetooth; each technology supports a specific market space.

Table 2-D Classic Bluetooth vs BLE [25]

	Advantages	Disadvantages
Classic Bluetooth	Streaming data, ideal for audio streaming for example	Energy intensive
	Large amount of data: music, videos, photos	Fairly short emission range: between 10 and 15 meters maximum
Bluetooth Low Energy	Low power consumption offering a very high autonomy of BLE devices	Non-continuous connection
	Long emission range of up to several hundred meters	Relatively light information
	Competitive acquisition cost	

2.5.3. Wi-Fi

Wi-Fi is a family of wireless network protocols based on the IEEE 802.11 family of standards, commonly used for local area networking of devices and Internet access. Wi-Fi is one of the most widely used wireless technologies. It allows all kinds of different devices to interface with the Internet. Internet connectivity occurs through a wireless router, when the Wi-Fi compatible devices connect to exchange information, creating a network [26].

Within the IEEE 802.11 standard, several variants are included that provide different coverage areas and even different signal strengths inside the coverage area. The ESP32 device used in the project supports all three standards, b, g and n.

Table 2-E IEEE 802.11 technology comparison [27]

Standards	Year	Frequency band	Average speed
IEEE 802.11 b	1999	2.4 GHz	11 Mbps
IEEE 802.11 g	2003	2.4 GHz	54 Mbps
IEEE 802.11 n	2009	2.4 / 5 GHz	600 Mbps

2.6. Dashboards

A dashboard is a tool to display all the desired data. They provide the users at-a-glance views of key performance indicators and other relevant information to a particular objective or business process. There are different platforms to create these boards (e.g. ThingsBoard, Grafana, Power BI...).

In this project, ThingsBoard is used. This is an open-source IoT platform that enables rapid development, management and scaling of IoT projects. With this tool, the designed dashboards can be dynamic and responsive; this is an important feature for this project because it enables the visualization of the sensor and the monitor values on the fly [28].

3. Product definition

In this chapter, there is more information about the developed project and the process of creating the product. The project definition, scope, planning and the product specification and requirements are explained.

3.1. Objectives

As previously mentioned, this project consists of monitoring a CPS over IoT devices. Primarily, the objective of this project is to feed Moonlight (the provided monitor) with live data. Moonlight has been previously tested with a Matlab interface since this tool has several CPS models and analysis tools available [8]. With that being said, this is the first time for the monitor to receive real-time data, so the purpose of this project is to help Moonlight be able to monitor different use cases and see how it works at runtime.

Therefore, the main objective is to implement a middleware. This middleware will enable the monitor to communicate with different applications in a distributed network. The middleware will provide some services and be capable of expanding by adding more services.

Several diverse goals must meet to implement this. On the one hand, the communication of the sensors with the monitor has to be established. The sensors are spatially distributed, collecting the surrounding data. The communication between the physical world and the middleware must be enabled using net access. The protocols used are BLE for the communication between different sensors and MQTT to communicate with the middleware. However, with this connection, the monitor cannot yet monitor. The received message needs to be adapted to be compatible with the monitor and the use cases to observe the data correctly. Finally, Moonlight must be capable of monitoring the system's behaviour at runtime, analyzing the current state of the physical part to detect if it satisfies the specified properties.

On the other hand, some objectives involve the process of understanding. The majority of topics and resources used in this project were never learned before, for example, a monitor like Moonlight. Moonlight, as the central resource, must be comprehended to implement it; this also entails understanding STREL, the logic-based specification language and the reading of papers. Other resources and topics to learn were the sensors, BLE and ThingsBoard.

To conclude, the middleware must adapt to the different use cases. The middleware is a software which offers and communicates different services. The services that must be present are the monitor, the data collector service (sensor service) and the dashboard service to show the results to the user. Furthermore, it should be easy to add more services and use cases, making it flexible, accepting different upcoming data and adding more value to the framework.

3.2. Project phases

The project duration is eight months, from November 2021 to June 2022. The project has been divided into some tasks and scheduled to manage the work and achieve the objectives. The development has been divided into the subsequent phases:

3.2.1. Introduction

The first month consisted of learning the subjects involved in the project and the tools employed during the development. The principal tool used was Moonlight; therefore, most of the time was dedicated to learning the concepts to comprehend it (i.e., CPS and STREL) and how to use it by studying and doing some examples.

3.2.2. Product development

After studying the project bases and making some trial examples with the Moonlight framework, the product development started. The development was carried out during the entire process, beginning with the middleware, and the framework, continuing with the networking and the sensors and finishing with adding new services like the dashboard.

During the development, all the work was being uploaded to a GitHub repository¹. Apart from being a directory to save and modify the code, the repository also holds all the versions of the content. Furthermore, this project used GitHub Actions, a feature to automate all the software workflows. Every time the repository had a push, the software started testing.

The planning was scheduled to ensure that the objectives were achieved before the deadline. There was a beta release planned for the beginning of April. The beta release consisted of a product that was close in look, feel and function to the final version of the project. This beta release happened one week after the planned date; however, the product had a better-structured interface than planned, making the future work more bearable. The product not only had the main functions of the final product done (i.e., all the flow beginning in the sensors until the Moonlight monitor), the sensors collected data, it was sent to the middleware and finally monitored at runtime. Also, it was able to support SOA (Service Oriented Architecture) fully. The services implemented in the middleware were in charge of the MQTT connections and the Online Moonlight service, able to monitor the input data. Besides, the tests had good coverage, then, making future changes occur in a shorter time.

Afterwards, a dashboard service was implemented. This allows the user to see the results clearly. Then, more sensors were added to test the scalability. Finally, the Wiener Linien use case was also added apart from the office use case.

Some project objectives had changed, or some things were added from the beginning to the end. During the project's development, the supervisors proposed new ideas, like using the

¹ <https://github.com/oihanagarciaa/GBL-IoTMonitoringSTREL>

dashboard. In the beginning, the objective was to develop the way from the sensors to the monitor, but finally, adding another step, the middleware became bi-directional. It was not only able to take the data and monitor it but also to get the results and communicate them to the client through the dashboard.

3.2.3. Other tasks

- **Tests:** Software testing has high importance in the process of developing and evaluating a product. It has several benefits, like preventing bugs and reducing development costs. Tests were for several purposes, to validate that the units perform as expected, verify the functions work well together, and, in some cases, do a TDD (Test Driven Development) where the development of the test cases occurred before the software. Lastly, with the continuous integration, the tests were automated every time the commits were pushed to the repository.
- **Documentation:** During the entire project, the details of the product were recorded in this document.
- **Meetings and more:** Every week, there was a meeting with the supervisors to keep track of the project, ask some questions and do or ask for suggestions. Now and then, there were other extra meetings if needed. Besides, the IoT master's courses took place in the first months, which helped with the sensors and the MQTT protocol later.

3.2.4. Gantt Chart

A Gantt chart was created at the beginning of the project to organize the development. Figure 3-A Resumed Gantt chart is the summarized version of the planning. Some tasks took longer than expected due to the changes in the product and other unexpected issues. Nevertheless, the project was successfully completed. As time passed, the track of the project was recorded in the chart of Appendix A **Gantt chart**, where the actual detailed development of the whole project appears.







TASK	START	END	
Introduction to the subject	03/11/2021	13/12/2021	
Documentation	19/11/2021	31/05/2022	
Product development	14/12/2021	30/06/2022	
Beta release	14/12/2021	04/04/2022	
Project release	05/04/2022	30/06/2022	
Others			
Attend IoT courses	15/11/2021	31/01/2022	

Figure 3-A Resumed Gantt chart

3.3. Product requirements

This thesis consists mainly in developing a software project. Due to this, the primary resources used are software. Nevertheless, some hardware devices were used too. Down below there is a list of the requirements needed during this project divided into two groups, software and hardware:

3.3.1. Software

Moonlight²

This is the principal resource, a lightweight Java-tool monitor; it can monitor temporal, spatial and spatio-temporal properties of distributed complex systems, like Cyber-Physical Systems. It has two different monitoring approaches, offline and online.

- Supports the specification of properties written with the Reach and Escape Logic (STREL)
- Implemented in Java
 - Features MATLAB interface
 - Python Interface under development

ThingsBoard

ThingsBoard is an open-source IoT platform for data collection, processing, visualization, and device management. It enables rapid development, management, and scaling of IoT projects. With ThingsBoard, there are many things to do:

- Provision devices, assets and customers, and define relations between them.
- Collect and visualize data from devices and assets.
- Analyze incoming telemetry and trigger alarms with complex event processing.
- Control the devices using remote procedure calls (RPC).
- Build workflows based on a device life-cycle event, REST API event, RPC request, etc.
- Design dynamic and responsive dashboards and present device or asset telemetry and insights to the customers.
- Enable use-case specific features using customizable rule chains.
- Push device data to other systems.
- Much more [11]

GitHub Actions and SonarCloud

GitHub Actions is a continuous integration and continuous delivery platform that allows for automating the build, test and deployment pipeline. This project is in a GitHub repository, so it

² [GitHub - MoonlightSuite/Moonlight: a light-weight framework for runtime monitoring.](https://github.com/MoonlightSuite/Moonlight)

is possible to run a workflow. Every time a push is done, the middleware is built, tested and analyzed in SonarCloud.

SonarCloud automatically analyzes branches, the code quality and code security. Knowing the bugs, code smells, security hotspots and coverage and the fast feedback made it possible to maintain the code cleaner.

Programming languages

Nearly all the project is developed in Java. As previously mentioned, Moonlight is implemented as a Java program, and so is the developed middleware. The version used is Java 17, the last Long Term Support release.

C is used to develop the program of the devices of this project. A language that can be considered one of the most widely used programming languages in IoT.

Gradle

Gradle is a build automation tool. It is open-source, and it is focused on flexibility and performance. Its features include the management of the dependencies.

Zephyr

The Zephyr Project is a scalable open-source real-time operating system (RTOS) supporting multiple hardware architectures (more than 350 boards); Thingy52 can be found between them. The Zephyr projects are CMake-based. CMake is an open-source, cross-platform family of tools designed to build, test and package software.

Integrated Development Environment (IDE)

First, for the middleware development, IntelliJ IDEA was used. This tool provides code completion, static code analysis and refactoring. This makes the development workflow experience smoother.

Then, Visual Studio Code was for the Thingys. This IDE provides just the tools a developer needs for a quick code-build debug.

Finally, Arduino IDE was also used. This makes it easy to write code and upload it to the board, in this case, to the ESP32.

3.3.2. Hardware

Thingy52

The Nordic Thingy52 board is an easy-to-use prototyping platform. It is designed for helping to build IoT demos with power optimization and several sensors. It is based on the nRF52832 Bluetooth 5 system on chip (SoC). This project uses Bluetooth Low Energy (BLE) and environmental sensors such as temperature, humidity, and air quality (CO2 and TVOC).

Features:

- Arm Cortex M4 32-bit, 64 MHz
- Configurable RGB LED and button
- Environmental sensors: Temperature, humidity, air pressure, air quality (CO2 and TVOC), colour and light intensity
- Nine-axis motion sensing: Tap detection, orientation, step counter, quaternions, Euler angles, rotation matrix, gravity vector, compass heading, raw accelerometer, gyroscope, and compass data
- Sound: speaker for playing pre-stored samples, tones, or sound streamed over BLE (8 bit 8 kHz LoFi) and microphone streaming (ADPCM compressed 16 bit 16 kHz)
- Secure over-the-air device firmware upgrade (OTA DFU)
- Battery, rechargeable, 1440 mAh
- Low power consumption
- 2.4 GHz transceiver, Bluetooth Low Energy
- Near field communication (NFC) support

Segger J-Link

SEGGER J-Links are the most widely used debuggers on the market thanks to their many supported CPUs and compatibility with the popular environments. The J-Link EDU Mini is a reduced version to allow students and educational facilities to debug different devices. USB-powered can communicate at high speed.

Features:

- Able to achieve ultra-low power consumption
- Built-in ESP-WROOM-32 chip
- Breadboard Friendly module
- Light Weight and small size
- On-chip Hall and temperature sensor
- Uses wireless protocol 802.11b/g/n and Bluetooth (Classic and LE).
- Built-in wireless connectivity capabilities
- Built-in PCB antenna on the ESP32-WROOM-32
- Capable of PWM, I2C, SPI, UART, 1-wire, 1 analogue pin
- Uses CP2102 USB Serial Communication interface module
- Programmable with ESP-IDF Toolchain, LuaNode SDK supports Eclipse project (C language)

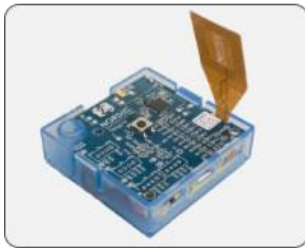
ESP32

ESP32 is a low-cost, ultra-low power consumption system on a chip microcontroller. ESP32 can interface with other systems to provide Wi-Fi and Bluetooth functionality through SPI/SDIO

or I2C/UART interfaces. This project case uses Bluetooth to communicate with the Thingy sensors and the Wi-Fi to publish it in the MQTT broker.

Features:

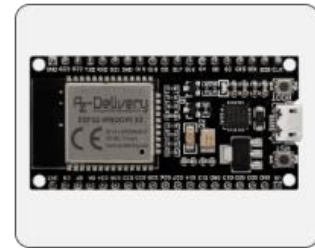
- Integrated low power 32-bit MCU
- Integrated TCP/IP protocol stack
- 802.11 b/g/n WiFi 2.4 GHz, support WPA/WPA2
- Support STA/AP/STA+AP operation modes
- 10-bit ADC, SDIO 2.0, (H) SPI, UART, I2C, I2S, IR
- Remote Control, PWM, GPIO
- Deep sleep power < 10uA, Power down leakage current < 5 uA
- Wake up and transmit packet in < 2 ms
- Standby power consumption of < 1.0 mW (DTIM3)
- +20 dBm output power in 802.11b
- Operating temperature -40C – 125C [11]



Nordic Thingy:52



Segger J-Link EDU mini



ESP32

Figure 3-B Hardware

4. Development

This chapter is dedicated to the development of the project. Here the processes to achieve the project are explained.

4.1. Features of Moonlight monitor

Firstly, in this section, the primary tool is explained in detail. This project is not focused on developing the Moonlight monitor, but it must incorporate it, so it is essential to elucidate. The used Moonlight monitor is divided into five elements to explain better the functionality. Some of these elements are taken from the paper [30], where software runtime monitoring techniques' common points and ground are extracted. Figure 4-A Features of the monitor shows how the monitor works in this project. It is noteworthy to repeat that Moonlight is under development, and this graph cannot be accurate in describing the monitor in the future.

- 1) **Monitored objects – Distributed:** The software entities used during the project, such as programs and services, are distributed. The program receives different streams of events during execution. The monitor communicates with different services such as sensors communication service and a dashboard during the process.
- 2) **Monitoring Access Methods:** This feature is presented in two aspects.
 - a. **Monitoring Code Instrument Methods – Automatic (Static):** For the initialization of the monitor, the formulas and the requirements are established by a client, without any intervention of programmers (Automatic). Once the requirements for the monitor are set up, the monitor service is initialized, and the values cannot be inserted or changed again (Static).
 - b. **Response Mechanisms – None:** The software developed during this project does not contain any technique to change the behaviour of the monitored system. The objective of monitoring is to guarantee that software is running as expected. This monitor does not have any response feature. Therefore, the client must have the ability to obtain the software runtime state information. In this case, the analysis results are displayed in a dashboard to the user.
- 3) **Monitoring Mechanism Implementation – Logic:** Moonlight employs the human-understandable logic STREL as a method to monitor. This framework can also handle mobile/dynamic CPS.
- 4) **Execution Relationships:** The execution relationship feature consists of two parts.

- a. **Monitoring Execution Models – Multi-Process Model:** The monitor is a separate service from the system being monitored. The behaviour and state of the monitored objects are communicated to the monitor externally. Moonlight works in a single thread; for now, the parallelization of the monitor is expected to be added in the near future [8].
- b. **Interaction Methods – Middleware:** The interaction method defines the communication between the Moonlight monitor and the monitored system. In this case, the middleware is used for the interaction. The monitor is not related to the monitored objects, and there is no need to know where the observed system is. The middleware is also used to communicate with different applications so that it can provide a distributed monitoring facility.

5) **Monitoring Algorithms:** Moonlight enables two types of monitoring techniques:

- a. **Offline:** All the data is available at the beginning of the execution. The offline method monitors the stored traces generated during the execution time. For this reason, generating and storing the system's execution traces so that they can be monitored is computationally expensive.
- b. **Online:** The online algorithm enables the RV during the system's execution. This technique is computationally less expensive and allows to take immediate action during the system's execution [21].

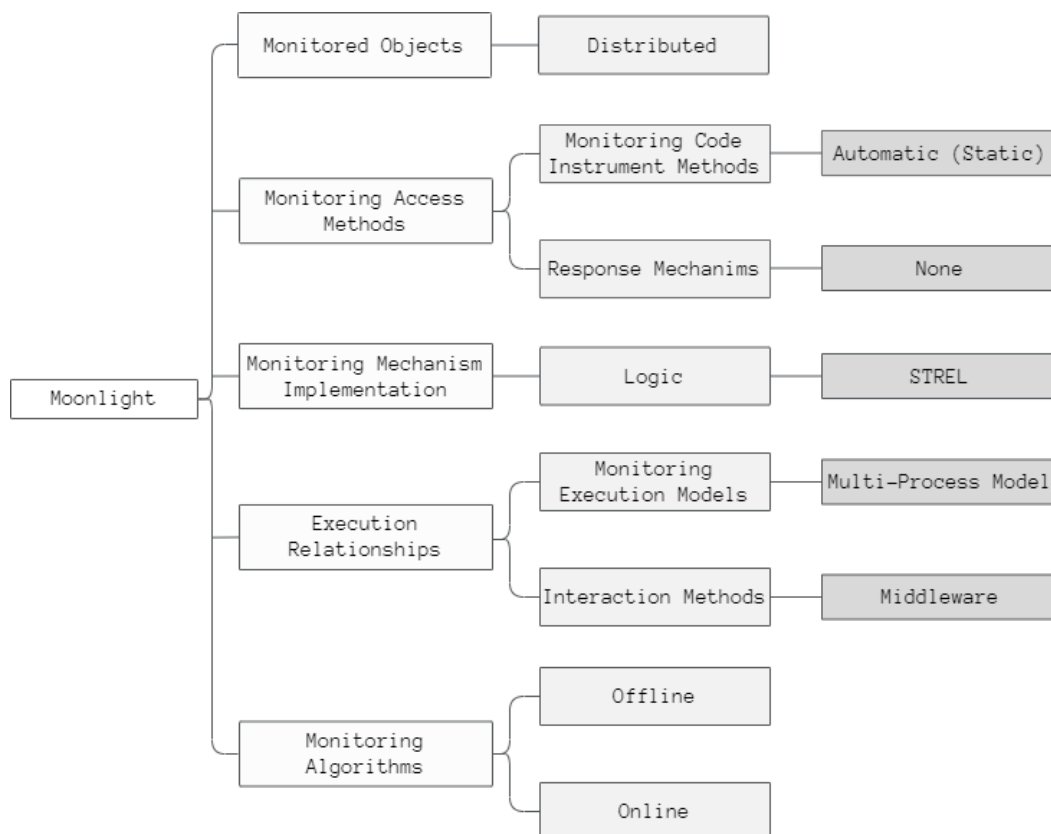


Figure 4-A Features of the monitor

4.2. Monitor initialization and basic workflow

Referring to the previous chapter, explains that the way to initialize the monitor is automatic and static. The requirements to observe the system are set up at the very beginning and cannot be changed in the future. Afterwards, the monitoring algorithm used in the project is the online technique. The online monitoring approach enables RV; this means that it is performed incrementally. When a new piece of data is available, the monitor can observe it. In the following figure appears the online monitoring model.

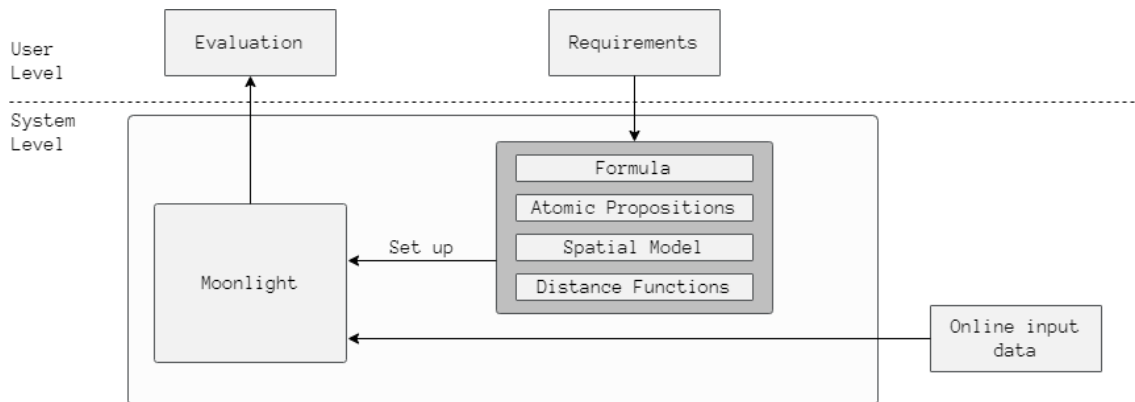


Figure 4-B Online Monitoring Model

4.2.1. Set up

The user specifies the requirements needed. “OnlineSpatialTemporalMonitor” is the exact name of the used monitor, and to initialize it, the user needs to specify the following requirements:

Formula

Moonlight is a monitoring algorithm for STREL; therefore, one input is a STREL formula. This formula must be specified to decide the satisfaction or violation of the observed events. The formula can be very flexible because it can return a Boolean or a quantitative verdict.

Atomic Propositions

One formula can be composed of multiple formulas. These propositions’ function is to complete the formula, and as the formula, the proposition sentence can be true, false or other value like an Integer or Double.

Spatial Model

STREL operates over a weighted graph representing the spatial arrangement of spatially distributed entities (i.e., maps each node and time instant into a vector of values, describing the internal state of each location). Hence, the monitor must be provided with the system’s spatial model.

Distance Functions

These functions are used to complete the formula. STREL has some formulas like Reach and Escape that use the distance to browse the spatial model, draw conclusions and satisfy the spatial property.

4.2.2. Increment - Data conversion

The data that arrives from the sensors is not adequate to monitor before transforming it (i.e., it is raw data). Due to this reason, the new information in the system goes through two types of changes.

Recording the received data

The first change receives the data of each sensor. The developed software expects a JSON file whose values are extracted to create a *Tuple.java* and save important variables like the time and the sensor ID. It is worth mentioning that this class is only declared just in the necessary classes. Whenever possible generic types are used to make future changes easier (i.e., if Tuple or other classes need to be replaced, there will be few changes). *Tuple.java* is a Moonlight class to record immutable objects. It is suitable for this project because the number and the type of elements specified inside can be modified depending on the needs of the use cases, making it versatile. Figure 4-C *Tuple.java* usage example is an example of how to use *Tuple.java*.

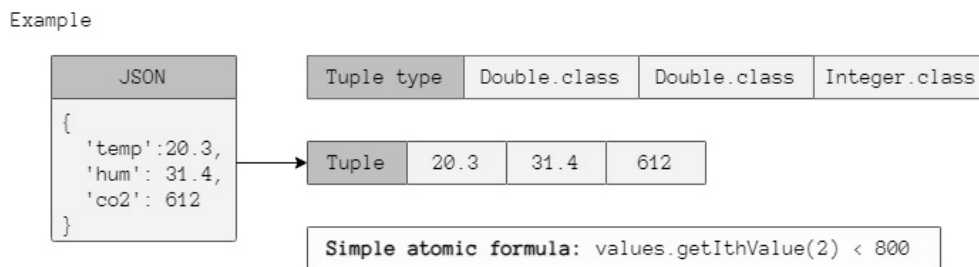


Figure 4-C *Tuple.java* usage example

All the received sensor messages are divided into three parts, **ID**, **time** and **values** (in this case, Tuple). With this separation, *Message.java* is created, and the information is ready for the next step.

Joining the values

The purpose of Moonlight is to monitor the increments. The online monitor can monitor two types of data: *Update.java* and *TimeChain.java*. Both java classes are composed of a list of values (i.e., Moonlight needs the information of all the nodes of the spatial model to monitor the increment). Consequently, there is a need to combine all the sensors' values. To do this, the middleware needs a **Buffer** to save the sensors' values and a converter to **create increments**. The process of joining the sensor messages has some complications that are explained in the Problems and solutions part. Finally, after joining the values, the increment is created, the monitor is executed, the evaluation is achieved and the results are sent to the user.

4.3. Middleware

The middleware is the main developed component in this project. Middleware is the software that connects different components or applications with each other. The middleware developed for this project supports distributed networks and provides services like messaging. The following figure shows the architecture of the middleware:

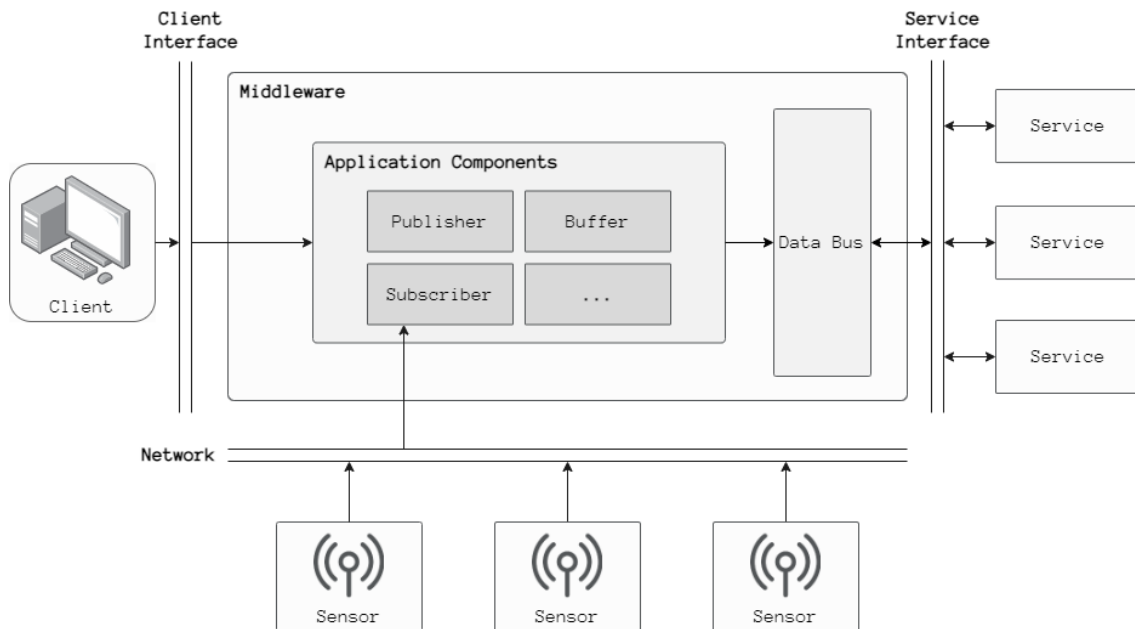


Figure 4-D Middleware Architecture

Middleware technologies are increasingly important due to the growth of network-based applications. It acts as the connective tissue between applications, data, and users [22].

There is a broad category of different kinds of middleware depending on the main functionalities. On the one hand, the developed middleware works as an integration tool, connects external and internal applications, sharing and streaming data among the components and the services in an asynchronous way. On the other hand, the middleware can support runtime for the use cases of this project, working agilely across services and the internal components, making the middleware behave as a new application on its own.

The middleware integrates the custom components, its own and purchased services (Software as a Service, SaaS). The middleware handles data management, application services and messaging [23]. These are the provided services:

- 1) **Sensor service (4.4.1):** The objective of this service is to facilitate the communication between the sensors and the monitor.
- 2) **Monitor service (4.4.2):** This is the largest service of the middleware. Includes methods that save messages, convert the data and monitor.
- 3) **Result service (¡Error! No se encuentra el origen de la referencia.):** This service is in charge of getting the monitor's results and communicating them to the user.

4.4. Services

SOA (Service Oriented Architecture) is an essential function for software evolution, development and integration. This architecture has lots of positive aspects that make it ideal for the development of this middleware:

- The service interfaces make the software components reusable and interoperable.
- The services use a common pattern making new services incorporate rapidly.
- The service interfaces provide loose coupling (i.e., services do not need to know how the services are implemented underneath). The internal functionality of a service does not affect other services; they are independent.

In this way, SOA represents a critical stage in application development and integration evolution over the last few decades. Before SOA emerged, connecting an application to data or other functionality in another system required complex point-to-point integration for each new developer project. SOA makes the integration of applications more accessible; thereby, the application's scalability and maintenance are improved [24].

The following diagram represents the architecture used to build this middleware. The services, data bus and other components are explained below.

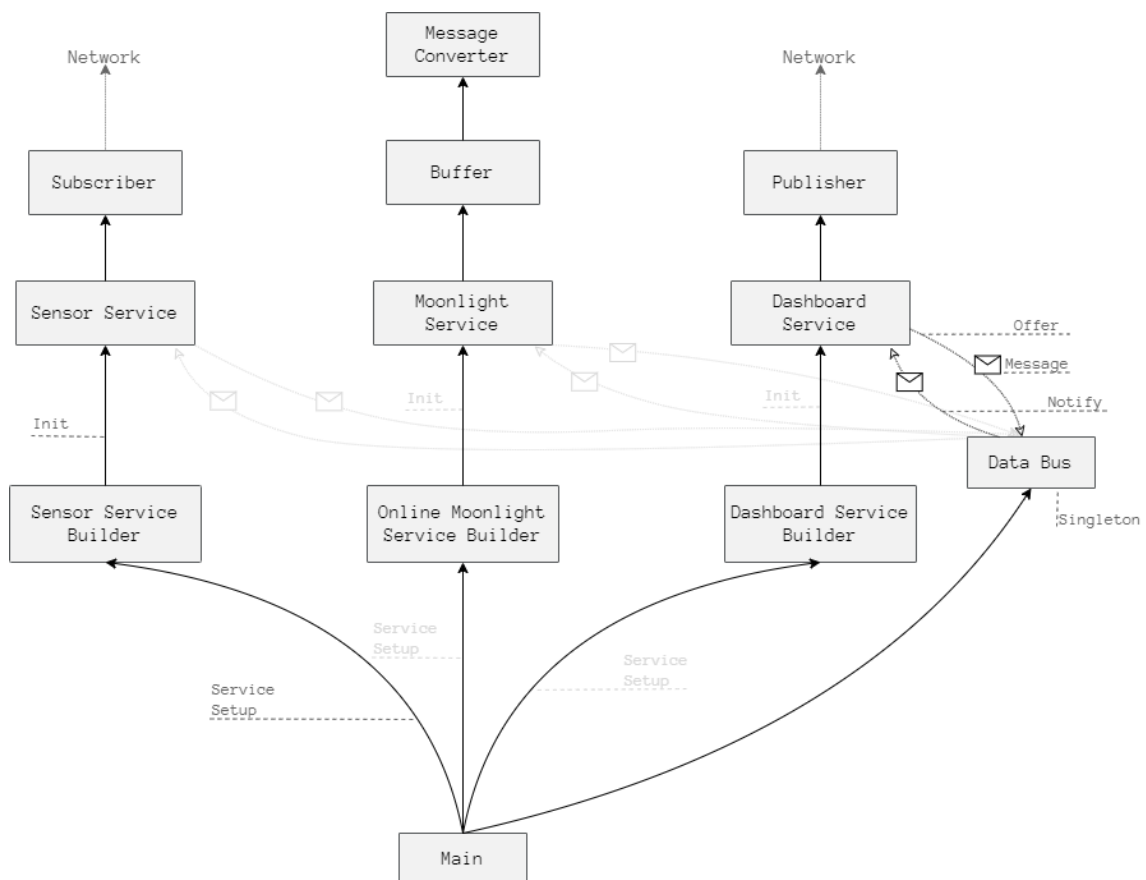


Figure 4-E SOA Diagram

4.4.1. Sensor service

This service facilitates the communication between the sensors and the monitor service. The sensor service contains a subscriber; this component works as a listener to the MQTT broker and receives the data to monitor.

The sensor service, as well as the other services, is as general as possible. The client specifies the distinctive details, and then these instructions are passed from the *Sensor Service Builder*. This service's two customizable variables are the broker and the topic to subscribe for the communication. For the communication with the MQTT broker, Eclipse Paho is implemented, an open-source library for lightweight publishing/subscribing messages.

Then the service needs the necessary information to transform the data to a *Message.java*. As explained in Recording the received data, sensor messages arrive in a JSON format, which must be transformed to a *Message.java*.

This conversion is achieved in a few steps, thanks to Gson. Gson is a Java open-source library that converts Java Objects into their JSON representation and vice versa. It does not need any Java annotations in the classes and provides simple *toJson()* and *fromJson()* methods for the conversions, which makes it a handy tool [25]. The client sends a class that implements *Message* and *CommonSensorsMessage<>* interfaces with the necessary information to fulfil the use case requirements.

The java class for the Office use case looks like this:

```

1. public class OfficeSensorMessage implements Message, CommonSensorsMessage<Tuple>{
2.     private int id;
3.     private double time;
4.     private double temp;
5.     private double hum;
6.     private int co2;
7.     private int tvoc;
8.
9.
10.    @Override
11.    public int getId() { return id; }
12.
13.    @Override
14.    public double getTime() { return time; }
15.
16.    @Override
17.    public Tuple getValue() {
18.        TupleType tupleType = TupleType.of
19.            (Double.class, Double.class, Integer.class, Integer.class);
20.
21.        return Tuple.of(tupleType, temp, hum, co2, tvoc);
22.    }
23. }
24.

```

Table 4-A Example of a sensor message class

By sending just a short class like the *OfficeSensorMessage.java* class, the client provides the necessary information for all the data transformation, beginning in the sensor service until the monitoring step.

The transformation is done in a single line in the sensor service, as is shown in Table 4-B. *jsonMessage* being a string with a JSON format and *messageClass* the specified desired java class (e.g., *OfficeSensorMessage.class*).

```
1. Message message = new Gson().fromJson(jsonMessage, (Type) messageClass);
2.
```

Table 4-B JSON to Message conversion

After transforming the data, the last step for this service is to offer this message to the data bus.

4.4.2. Moonlight service

This is the service that contains most components. It is worth mentioning that apart from the Moonlight monitor, the other components were created from scratch. First and foremost, here is the Moonlight monitor, to be more specific, the online monitor. To initialize it, first, all the requirements needed for Moonlight to get to work are received. Once the monitor can run, receives a stream of messages from the sensors, also known as increments. These messages cannot be monitored before adapting them, as explained in Joining the values, Moonlight needs all the nodes of the spatial model to monitor the increment. Accordingly, there is a buffer component that saves the values.

Buffer

A buffer or a data buffer is a region of a memory usually used to store data for a short period of time. Buffers are often used in conjunction with input/output to hardware and sending or receiving data to or from a network, as it happens in this project. This buffer is used because of two reasons:

On the one hand, there is a difference between the rate at which data is received and the rate at which it can be processed. The buffer gets new data every time a sensor sends new updates at a variable time rate. These updates are very frequent, physical systems consist of multiple sensors and each sensor is sending messages repeatedly. On the other hand, the monitor needs to know the information and state of different sensors to be able to monitor. There is a need to save the sensors' values temporally. The time rate of the monitor to read from the buffer and process the data is slower than the one to update the data to the buffer.

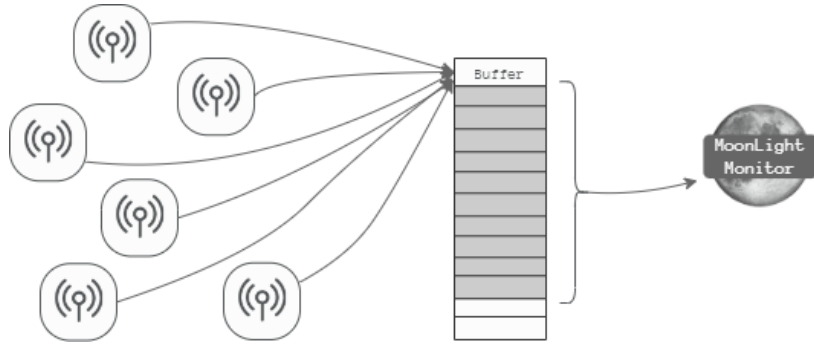


Figure 4-F Data buffer

The buffer type implemented is a **fixed size buffer**. A fixed size buffer is a storage for a fixed length of variables of a given type. The sensor messages are saved and when the buffer arrives to the length value, all the values are collected to join them and monitor them all together. Letting the buffer empty and ready to receive more messages.

Conversion – Message to Increment

After collecting the necessary amount of information, the values need to be converted in order to monitor. Two types of classes are possible to use, *Update.java* and *TimeChain.java*. Both classes have their advantages and disadvantages, but after analyzing both possibilities, it was decided that the best class to use was *TimeChain.java*.

Update.java is a data class to store increments of the kind $[start, end) \rightarrow value$. Using this class, the issue is that it is not possible to know the end time. The sensors are programmed to send the values in a fixed time period; however, there can be delays and other issues that make delay the messages. Instead, *TimeChain.java* is similar to a LinkedList, providing some specific features like checking temporal integrity constraints (i.e., the chain must be in monotonic time order) and a custom iterator. It does not need to specify an end to the value and can support having multiple values linked in a chain, which makes it more suitable for this project.

Moonlight monitor

Lastly, Moonlight monitors the *TimeChain*. The formula and other necessary information to analyze the system are transferred at the beginning. The online monitor is also created at the start, so a call is made to the monitor when the buffer is full.

```
1. SpaceTimeSignal<Double, Box<T>> results = onlineMonitor.monitor(buffer.get());
2.
```

Table 4-C Monitor call

The *buffer.get()* gets the *TimeChain* with just the last values. Thanks to the online method, the monitor does not need to save the previous values. In offline monitoring, the system's execution traces must be stored. In this case, the *TimeChain* contains the values that the monitor has not received yet. When these new values are sent to the monitor, they are deleted, making the online method less expensive.

Although the values assigned to the monitor contain only the new data, the monitor returns the results of all the monitored time. The middleware gets the *SpaceTimeSignal* with the monitor analysis. Moonlight works with STREL, a spatio-temporal logic, so the results received have time, place and value properties. The values are the obtained results, these values can be Boolean or can be another quantitative verdict.

```

1. Segment(start=0.0,
2. value=[Interval: [true, true], Interval: [true, true], Interval: [true, true]])
3. Segment(start=6.0,
4. value=[Interval: [false, false], Interval: [true, true], Interval: [true, true]])
5. Segment(start=13.0,
6. value=[Interval: [false, true], Interval: [false, true], Interval: [false, true]])
7.

```

Table 4-D *SpaceTimeSignal* results in a String format

Table 4-D shows an example of the results in a String format. In this example the monitor has these two characteristics: the spatial model size is three and the results are Boolean. For example, in the first segment, at time 0, the formula is true in all the places. At time 6 the value of the first entity turns false. The last message that the monitor has received is at time 13. From the 13-th time onward, there is a total absence of knowledge about what values the signal could have, that is why until then, the value to refer to them fill be [false, true] or $[-\infty, \infty]$. Furthermore, if the monitor receives new data but does not change the obtained results, it will not appear as another segment (e.g., if in the previous example the monitor had received an increment at time 3 and the results were all true, in the *SpaceTimeSignal* it would not have appeared as another segment; the Table 4-D would not have suffered any changes). Finally, the results are sent to the data bus.

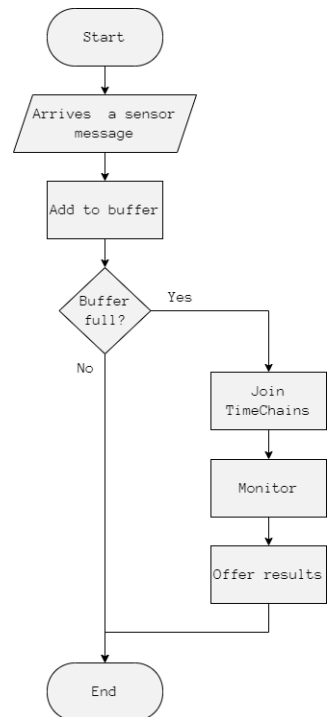


Figure 4-G Monitor service flow chart

4.4.3. Dashboard service

The dashboard service is in charge of displaying everything that is happening in the system to the user. As explained in one of the previous chapters, the objective of monitoring is to guarantee that the system's behaviour is correct. Neither the monitor nor the middleware have any response feature. Therefore, the client must be informed of the software runtime state information. Then, a new service was implemented in the middleware, *ThingsBoard*.

This service receives the state of the sensors and the results of the Moonlight monitor. It is interesting for the client to see the actual values too. The formula that analyzes the monitor may contain multiple factors to decide the final verdict. With just the final results it can be hard for the client to know where the issue is. So, the user can see both the sensors' actual values and the monitor's analysis.

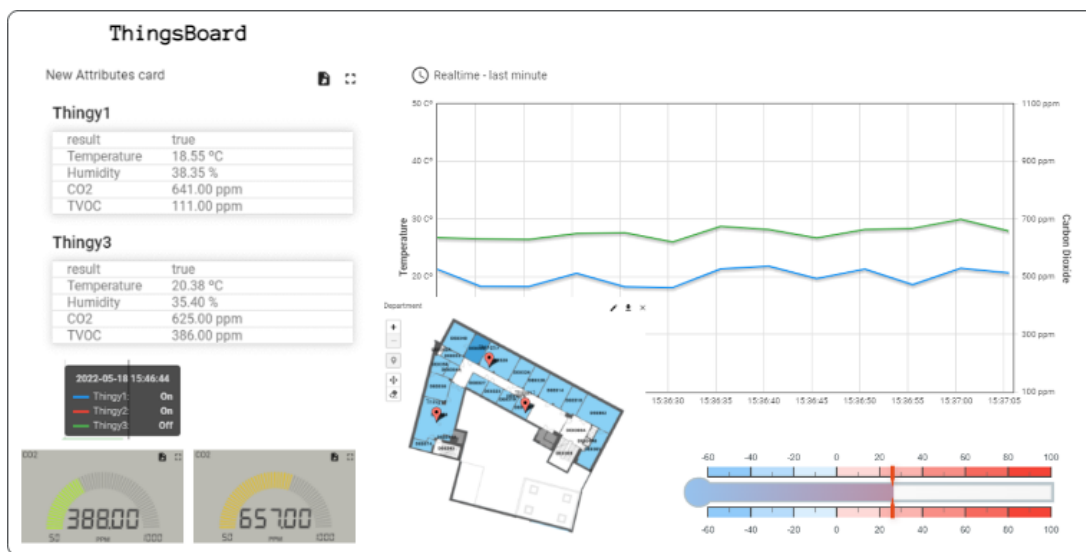


Figure 4-H ThingsBoard example charts

In ThingsBoard there are many different widgets available. They are dynamic and responsive, so every time a new value arrives, they are plotted and shown on the screen.

To manage the dashboard, the ThingsBoard provides multiple entity types. ThingsBoard is an IoT platform, so the user can create devices, and the basic entity type and relate them to manage the dashboard better. For example, in this project one device was created for each thingy and then they were grouped together with the alias *Thingys*.

ThingsBoard has a lot of advantages, it is an open-source IoT platform, it has multiple widgets for the visualization, and it enables rapid development, management, and scaling of IoT projects. But it has one inconvenience. The values of the sensors are saved with the timestamp when the message arrives (e.g. the temperature is 20°C at the time when the information arrives to the dashboard). In the case of the sensors' values, although it is not perfectly accurate, the time difference is not very different. In the case of Moonlight, the accuracy decreases, because it does not monitor so often. There is a possibility to change the time values with another global timestamp. This project does not use global timestamps, so there is no way to change the time.

Anyways, this issue is not critical, the user can still get good quality information from the dashboard.

The communication between the middleware and the ThingsBoard is done with MQTT.

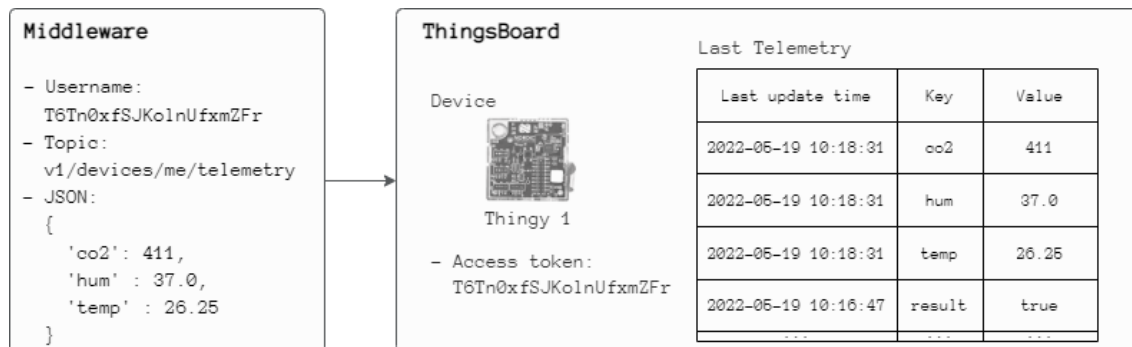


Figure 4-I Middleware ThingsBoard connection

Figure 4-G shows the elements that take part in the communication in an example. In ThingsBoard there are different created entities, and each entity has an access token. To update the values of the device, the established MQTT connection must have the value of the username the same as the access token. The default topic to send the message is “v1/devices/me/telemetry” but it can be changed. Finally, a JSON file is sent with the values to update. These values are recorded, and they can be checked in the “Last Telemetry” tab. Here all the values that arrived at the device are shown accompanied by the last update time and a key.

As said before, the middleware sends both the sensor values and the monitor results to the dashboard. These are two types of different messages, so they must be treated differently. To program this in an effective way in the middleware, a behavioural pattern was used, the *Template Method*.

Template Method defines the skeleton of an algorithm in the superclass and lets the subclasses override specific steps of the algorithm without changing the structure. This design pattern is used when there are several classes that contain almost identical algorithms [35].

```

1. public abstract class ThingsboardConnector {
2.     public void publishToThingsboard(String username, String json);
3.
4.     public String getUsername(Map<String, String> deviceAccessToken, String key) ;
5.
6.     public abstract void sendMessage(Message message);
7.
8.     public abstract String getJson();
9. }
10.

```

Table 4-E ThingsBoard connector abstract class

In this case, for example the process of publishing always works in the same way. The MQTT subscription is established with an specific username and a JSON is sent. But there are some

differences when sending the message. For example, when a sensor message arrives, the message is forwarded directly to the device; but the forwarding of the results works differently. Each node of the graph has its own result, so the middleware must do multiple communications from one message.

4.4.4. Service Builders

Builder is a creational design pattern to construct complex objects step by step. The pattern organizes the construction of an object into a set of steps. In this case it is used to create services. Each service requires a different implementation to build the object, so several different builder classes that implement the same set of building steps are created. At this project, the building steps are called in a specific order directly from the client code.

The Builder pattern is used for different reasons. The main reason to use it in this product is because the code must be able to create different representations of a product (i.e., multiple services). The builder interface defines all the construction steps, and the concrete builders implement these steps to construct particular representations of the product.

This creational pattern has several advantages. As constructors are composed of steps, they can be executed step by step, deferred or recursively, the constructor code is reusable, and each step has a single responsibility (Single Responsibility Principle). The only disadvantage is that the overall complexity of the code increases since the pattern requires creating multiple new classes [36].

This is the structure of the builders of the project:

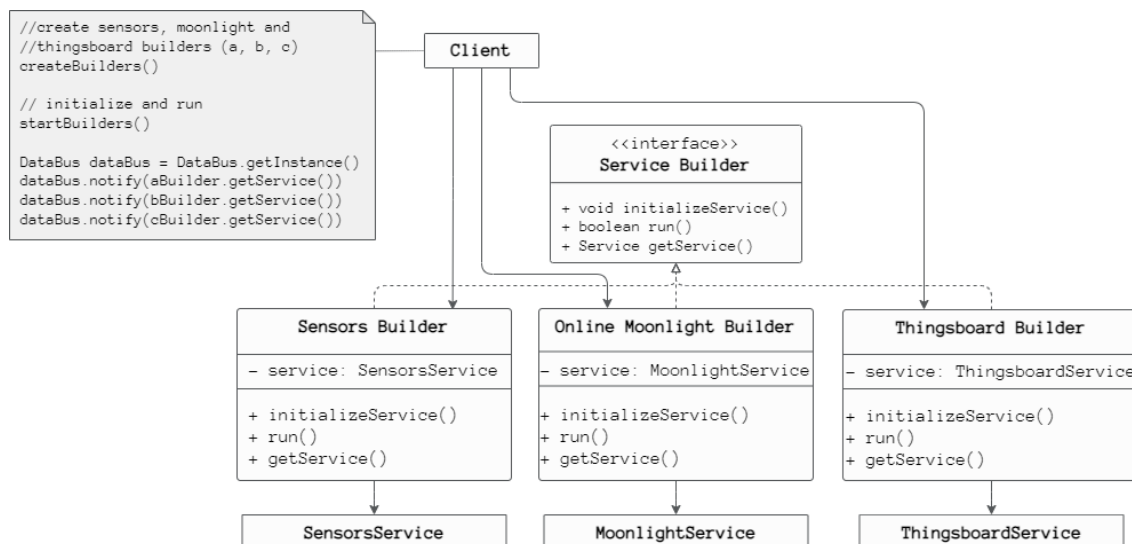


Figure 4-J Builder design pattern

The client creates the service builders and these service builders create the services providing them with the customized formulas, spatial models, brokers etc. After the services are created, they are sent to the data bus so they can communicate with each other.

4.5. Data Bus (Horizontal messages)

The data bus created works like an ESB (Enterprise Service Bus). This is an architectural pattern where a centralized software component performs integrations between applications. In this project, the middleware enables the communication between the implemented services. The created software has some components which are responsible of transforming the data and handling the connectivity and the messaging.

By implementing it in the SOA architecture, the services don't need to directly connect to any service. Without this each service would need to perform the necessary data transformation to meet each of the service interfaces and the directly connect, that is a lot of work and creates a significant maintenance challenge in the future. That's why this pattern is considered as an essential component of SOA [33].

The data bus used is a singleton. A singleton is a design pattern that ensures that a class has only one instance and provides a global access point to this instance. Ensures that a class has just a single instance, the data bus is a shared resource, and all the services will have the same object. Then, provides a global access point to that instance, the object can be accessed from anywhere in the program. These two characteristics make this pattern appropriate to use it in the data bus, however, it brings have some problems. For example, when writing "*new DataBus()*", the programmer would expect to have a new fresh object, but would receive the one that is already created. Also it can be unsafe since it can be overwritten by anyone. To solve this issues, the DataBus has the default constructor private to prohibit the use of new; and the object is saved in a static field [26]. This is the structure of the singleton data base:

```

1. public class DataBus {
2.     private static DataBus instance;
3.
4.     private final List<Service> services;
5.
6.     private DataBus() {
7.         services = new ArrayList<>();
8.     }
9.
10.    public static DataBus getInstance() {
11.        if(instance == null) {
12.            instance = new DataBus();
13.        }
14.
15.        return instance;
16.    }
17.
18.    public void offer(Message m) {
19.        instance.services.forEach(s -> s.receive(m));
20.    }
21.
22.    public void notify(Service s) {
23.        instance.services.add(s);
24.    }
25.
26. }
27.

```

Table 4-F Singleton Data Bus class

In summary, the data bus offers a centralized (thanks to the singleton) communication and integration between the services of the developed software. The services connect to the data bus, so the needed time for the integration is reduced and provides a scalable solution. However, this approach can be considered as a bottleneck in the future, all the services are connected to the same object, sending messages continuously, can provoke some delays.

4.6. Client

...

4.7. Physical system

The physical system is a network of spatially dispersed and dedicated sensors that record the physical conditions of the environment and forward the collected data to the middleware. The network relies in wireless connectivity (BLE and WiFi). The network has one direction, from the sensors, that work as an output, to the monitor.

This network was created for the Office use case. There are two types of sensors involved in the network. First, the Thingy52 collects environmental data using temperature, humidity and air quality (i.e., CO₂ and TVOC) sensors. For the communication, the Thingy does not have Wi-Fi connection, so the network needs another device to be able to forward the collected data. Therefore, the next sensor is the ESP32, this works as an intermediary between the Thingy and the middleware.

To be more specific, the Thingy52 is a broadcaster, the device's purpose is to transfer data to a device in a regular basis. To advertise the environmental values to the nearby ESPs, Bluetooth beacon (**¡Error! No se encuentra el origen de la referencia.**) is used. With this technology, the ESP, a compatible receiving device, picks up the transmitting packets with the message. Therefore, on one side, the ESP is an observer, it repeatedly scans the advertising packages that are being broadcast by the Thingy. And in the other side, it works as a publisher. ESP is connected to the Internet and sends the received message to the MQTT broker, where the middleware is subscribed.

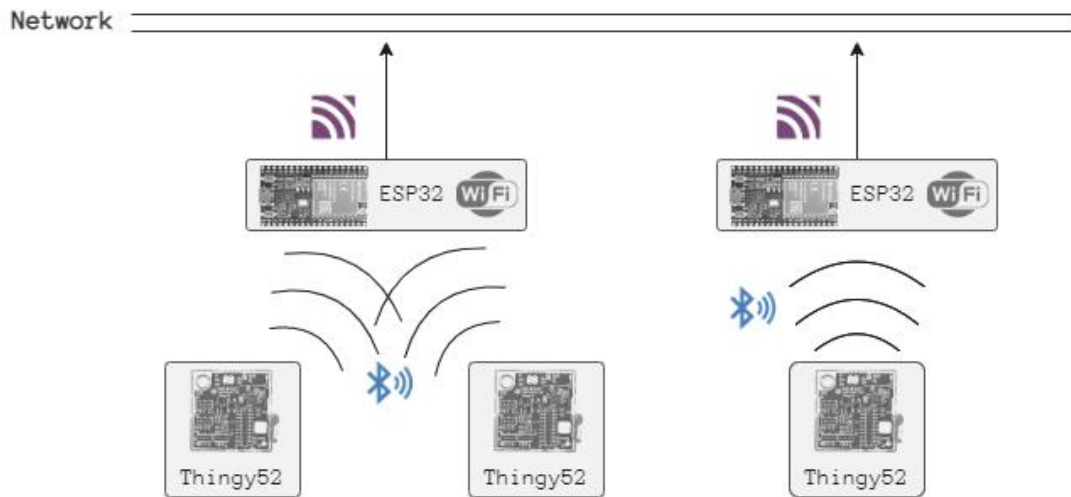


Figure 4-K Physical system

4.7.1. Thingy52

Thingy52 is a prototyping platform to create demos without starting from scratch. To build, flash and run the applications, the Zephyr development environment is used. The Thingy52 needs to collect the environment information, so first, the sensors must be activated in the Kernel configuration file (named *prj.conf*). In this file there are specified the application-specific values and the board-specific settings [38].

The configuration must activate the following sensors and application settings: air quality sensors, temperature, humidity, real-time-clock and Bluetooth. For the temperature, humidity and air quality sensors, I2C driver must be enabled because they share a direct dependency. An I2C (Inter-Integrated Circuit) chip driver controls the process of talking to individual physical devices that, such as the ones just mentioned [39].

Additionally, the use of RTT Viewer is enabled. This allows to see the print lines of the device and check that everything works correctly.

```

1. # Activate the sensors
2. # Most of the on-board devices are accessed through I2C
3. CONFIG_I2C=y
4. CONFIG_SENSOR=y
5.
6. # Air quality sensors
7. CONFIG_CCS811=y
8.
9. # Temperature and humidity sensors
10. CONFIG_HTS221=y
11. CONFIG_HTS221_TRIGGER_NONE=y
12. CONFIG_CBPRINTF_FP_SUPPORT=y
13.
14. # enable rtc timer interface
15. CONFIG_NRF_RTC_TIMER=y
16. CONFIG_NRF_RTC_TIMER_USER_CHAN_COUNT=1
17.
18. # Enable Bluetooth
19. CONFIG_BT=y
20. CONFIG_BT_PERIPHERAL=y

```

```

21. CONFIG_BT_DEVICE_NAME_DYNAMIC=y
22. CONFIG_BT_DEVICE_NAME="000"
23.
24. # To see the printk-s with JLinkRTTViewer
25. CONFIG_USE_SEGGER_RTT=y
26. CONFIG_RTT_CONSOLE=y
27. CONFIG_UART_CONSOLE=n
28.

```

Table 4-G Thingy52 prj.conf, Kernel file

Regarding the Bluetooth beacon, the Thingy52 advertises continuously their characteristics. It has the ID as the Bluetooth device name, to identify to which Thingy the ESP is connecting to. For the ESP to read the Thingy's values, the reading permissions must be conceded.

```

1. BT_GATT_SERVICE_DEFINE(
2.     read_sensor_service, BT_GATT_PRIMARY_SERVICE(&my_service_uuid),
3.     BT_GATT_CHARACTERISTIC(
4.         &my_characteristics_uuid.uuid,
5.         BT_GATT_CHRC_READ,           // service-characteristics is readable
6.         BT_GATT_PERM_READ,          // read and write permissions
7.         &read_callback, &write_callback, (void *)1));
8.

```

Table 4-H Thingy read permissions

With the setup of the Table 4-H, the connected devices will be able to read the values of the environment. Once the ESP has discovered a readable characteristic and after establishing the connection between the two devices, the observer makes a request to read the broadcaster's values and the `read_callback` function is called.

```

1. static ssize_t read_callback(struct bt_conn *conn,
2.                             const struct bt_gatt_attr *attr, void *buf,
3.                             uint16_t len, uint16_t offset) {
4.
5.     const char *message = getCurrentValues();
6.
7.     return bt_gatt_attr_read(conn, attr, buf, len, offset, message, strlen(message));
8. }
9.

```

Table 4-I Thingy read callback

This function it gets the current environment values and sends them to the device that made the request.

4.7.2. ESP32

ESP32 is the device that connects the Thingy52 with the middleware. This chip has integrated Wi-Fi and dual-mode Bluetooth and is a development board for Arduino. The ESP needs to send the data to the middleware; therefore, it needs to connect to the Wi-Fi and be able to use MQTT. To achieve this, two libraries are used, *WiFi.h* and *PubSubClient.h*.

The former enables connection using the Arduino WiFi shield. With this library is possible to connect either to open or encrypted networks. This library is compatible with all architectures so it can be used in the ESP or all the other Arduino boards [40]. The latter is a client library for MQTT messaging. MQTT is a lightweight messaging protocol, ideal for the ESP and other small devices. This library allows to send and receive MQTT messages, but in this project is only used to send messages [41].

With steps from above, the ESP can establish a connection with the middleware. For the connection with the Thingys, BLE is used. Bluetooth market has a lot of demand. In 2021 they were 4.7 billion Bluetooth device shipments [42]. To find the Thingys around so many Bluetooth devices UUIDs (Universal Unique Identifier) are used. The Generic Attribute Profile (GATT) establishes in detail how to exchange all profile and user data over a BLE connection. GATT has two attributes, service and characteristics. For the primary service discovery and characteristic discovery, ESP scans all the BLE devices nearby, but it connects just to the ones that has the required UUIDs [43].

Table 4-J BLE connection characteristics

Thingy52

```
static struct bt_uuid_128 my_service_uuid =
BT_UUID_INIT_128(
BT_UUID_128_ENCODE(0x10362e64, 0x3e14,
0x11ec, 0x9bbc, 0x0242ac130002));

static struct bt_uuid_128
my_characteristics_uuid = BT_UUID_INIT_128(
BT_UUID_128_ENCODE(0xa3bfe44d, 0x30c3,
0x4a29, 0xacf9, 0x3414fc8972d0));
```

ESP32

```
static BLEUUID serviceUUID("10362e64-3e14-
11ec-9bbc-0242ac130002");

static BLEUUID charUUID("a3bfe44d-30c3-
4a29-acf9-3414fc8972d0");
```

Moreover, the ESP can connect with multiple devices at a time. So, one ESP can handle the messages of up to three Thingy52 devices. Finally, the ESP forwards the messages received in a JSON file every time it makes a request.

5. Problems and solutions

In this section the problems found during the development are explained, and afterwards, the implemented solutions are shown.

5.1. Dealing with null values

The CPS must be able to overcome the systems uncertainty related to the absence of information. The sensors send information continuously but not all the sensors send the data at the same time. Figure 5-A represents the problem of the missing values. In the example from bellow, the second sensor started later than the first one. Moreover, there are gaps between each sensor's messages and finally, the time interval between the values can change.

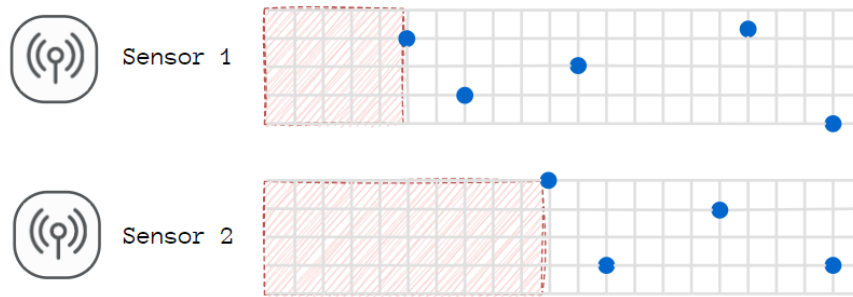


Figure 5-A Missing values problem

First, the values from the beginning are discarded (i.e., the red are in the Figure 5-B). In this way, Moonlight does not receive null values for the second sensor and it does not throw any error when monitoring. To fill the gaps between each message, *TimeChain.java* class is used. TimeChains work similarly to a LinkedList. Each sensor has its own TimeChain (i.e., *TimeChain<Double, V>*, Double being the format of the time and V the generic type for the values to be analyzed) to record their values. The gaps between each message are filled as it appears in the next image.

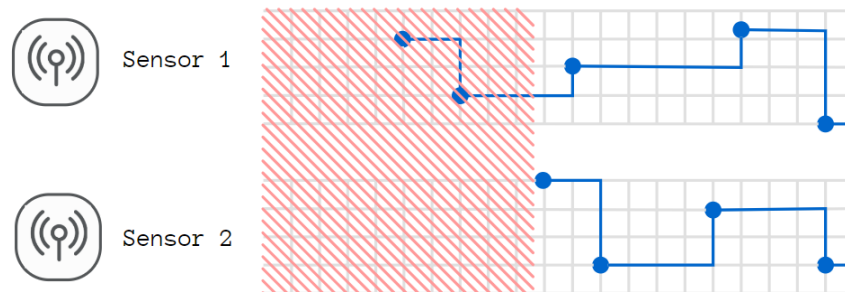


Figure 5-B Missing values solution

With this approach, all the null values are filled, and the problem is solved. The monitor does not receive any null value, so it can monitor the system correctly. Additionally, if a sensor stops

sending messages, it does not crash the monitor and it continues evaluating the rest of the system.

5.2. Managing the buffer

With the solution from above, the missing and null values are solved. However, to monitor the system Moonlight needs all the sensor's values at the same time. More specifically, the developed monitor needs the data to be *"TimeChain<Double, List<V>>"* type. Moreover, the Moonlight uses the online method to monitor the state of the system. This method observes just the new values, without repeating the already monitored values. Therefore, there must be a way to empty the buffer without just dropping all the sensors values. If all the values were dropped there would be null values and the problem explained in the previous chapter it would appear again (i.e., the problem represented with the red area in the figures Figure 5-A and Figure 5-B).

To summarize, the buffer must have a way to drop the monitored values in a proper way and save the values in a list *"List<TimeChain<Double, V>>"* and convert it to *"TimeChain<Double, List<V>>"* for Moonlight to be finally able to monitor.

To solve these

1. Time Chain Splitter
2. Join

5.3. Sensors time synchronization

Global reference time is a very important criteria in CPS components. The global reference time helps to ensure the real-time communication performance is achieved. With a global reference time, there is an assurance that the communication between physical and cyber world will work properly [11]. Unfortunately, the Thingy52s do not have access to the internet and is not possible to use the global time. As a time reference the Thingys use their internal RTC (Real Time Clock). This module measures the passage of time, provides a generic, low power timer on the low-frequency clock source (LFCLK). The timer of this device makes 32000 ticks per second [44]. In the Table 5-A appears the function to get the time, which returns the milliseconds passed since the sensor is turned on.

```

1. static const int64_t increment = 32000;
2.
3. unsigned long long getRealTimeValue(){
4.     uint64_t time;
5.     time = z_nrf_rtc_timer_read();
6.     return time / (increment / 100);
7. }
8.

```

Table 5-A Thingy get time function

Therefore, the problem is that at the same global time, the value of the clocks can be very different. Although if the sensors were switched on in similar times, the thingy has a grand precision when reading the past milliseconds and the temporal integrity of the monitor would be violated. Because the monitor requires to receive the values in monotonic time order.

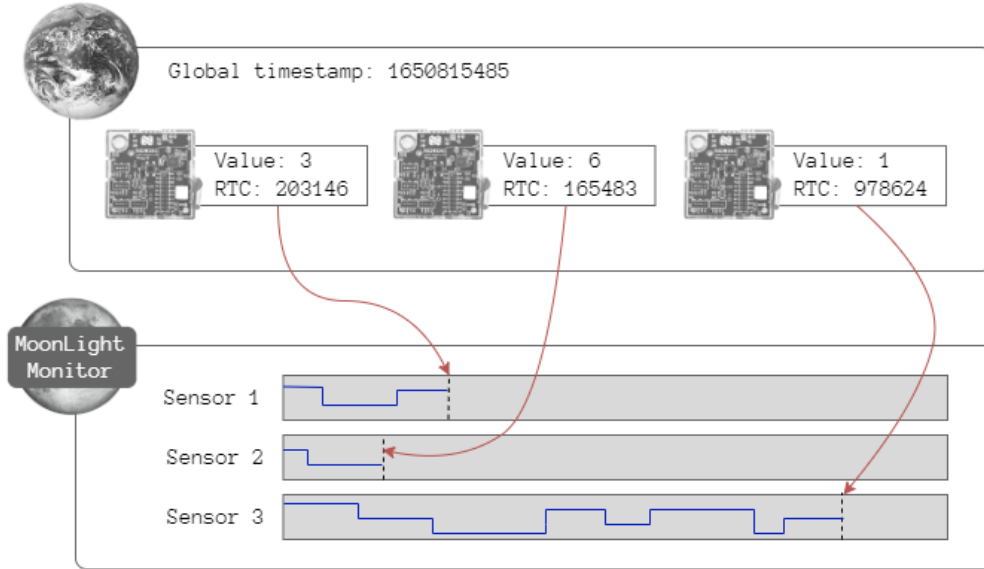


Figure 5-C Time synchronization problem

Figure 5-C represents the problem stated. To solve this problem a new process is added to the Middleware to synchronize all the sensors together. Clock synchronization is a topic in computer science and engineering that aims to coordinate otherwise independent clocks. Each sensor in the system shares its local time with the Middleware.

In distributed systems usually the clocks of the devices are adjusted by using an external clock reference or by sharing their clock and adjusting depending on the other nodes [45]. In this project the solution does not involve changing the sensors inner clock information. The sensors memory and capacity are limited, so the middleware gives it a solution without involving more communication to the network.

When the sensors messages start arriving to the middleware a time list is filled with the times. Figure 5-D is an example to explain how the synchronization works. When a message arrives from a device the time is saved in the device id's position. This is done until all the spaces of the list are filled, in this case the spatial model size is three, so when the times of the sensors 0, 1 and 2 are received the sensors are considered synchronized.

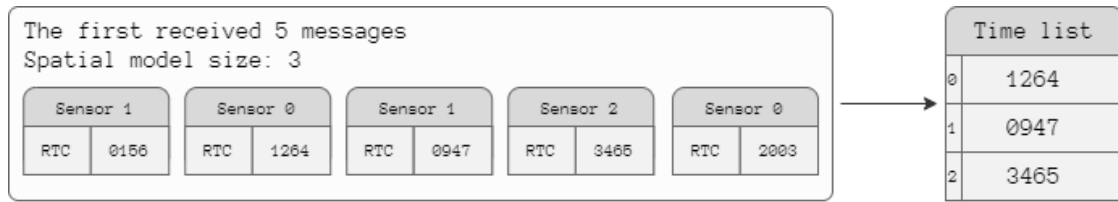


Figure 5-D Register of the time of the messages

In this way, taking as a reference this example, the middleware would take the time 1264 as time zero for the first sensor and so on. This is not an optimal solution; the ordering of events and the synchronization are wide camps by their own. Besides, in a distributed system there is a message transmission delay, but in this solution, it is not considered. The project is not focused here, so the solution explained in this section is taken as a good approximation for this project.

5.4. ESP32 problems

5.4.1. Connections

5.4.2. Memory

5.5. Not solved problems

5.5.1. Sensors error

ESP MQTT connection not always work

ESP can lost connection to the sensors

5.6. Other problems

- MoonlightRecord had some problems: Escape + online monitor + MoonlightRecord = infinite loop

The null values didn't throw errors, wrong error handling.

I reported these issues -> Ennio created Tuple.class, that had everything MoonlightRecord was wrong but with these errors fixed: I just used Tuple from that moment on.

- I was having problems with Windows + Zephyr project -> I used Linux for the coding of the sensors

6. Use cases

6.1. Office use case

7. Results

...

7.1. Efficiency

7.2. Robustness

//sonarcloud

Error handling

Maintainability

8. Economic memory

Gastuak

The majority of the cost of a software project is in long-term maintenance. [clean code liburua]

Irabaziak

9. Conclusions and future lines

9.1. Conclusions

a. Reflexiones técnicas: relacionadas con los objetivos del proyecto b. Reflexión sobre las implicaciones sociales, de salud y seguridad, medioambientales, económicas e industriales c. Reflexión sobre la aplicación de conocimientos relativos a cuestiones económicas, organizativos de gestión (gestión del riesgo y del cambio) en el contexto industrial y comercial.

/!\ Agenda 2030

9.2. Future lines

10. Personal evaluation

11. Gradu Bukaerako Lanaren laburpena (Basque summary)

Atal honetan sarrera, ondorioak eta etorkizuneko ildoak atalen laburpen bat egingo da euskaraz.

11.1. Sarrera

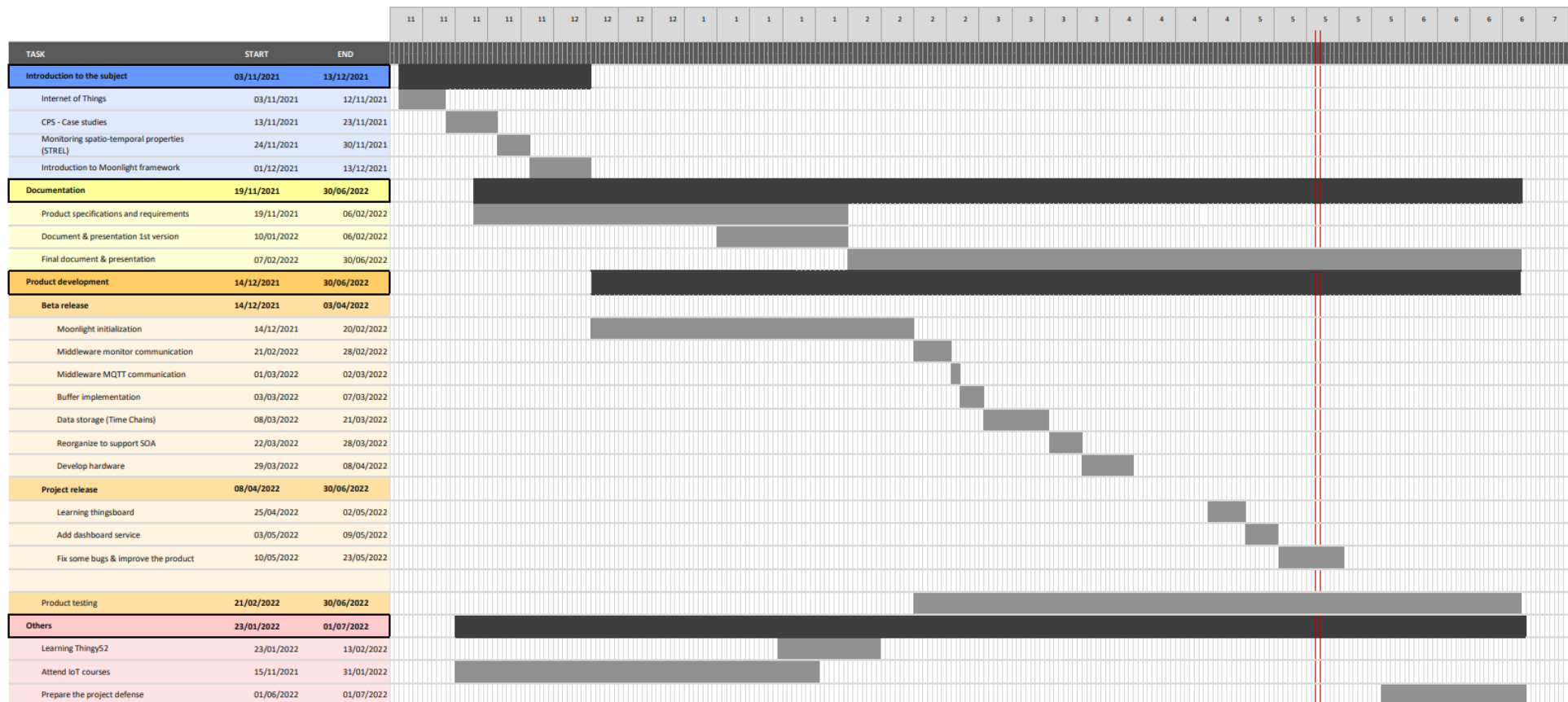
11.2. Ondorioak

11.3. Etorkizuneko ildoak

12. Appendix A

Gantt chart

update



13. Bibliography

- [1] T. Insights, «Statista,» December 2020. [En línea]. Available: <https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/>.
- [2] «<https://www.oracle.com/internet-of-things/what-is-iot/>,» Oracle. [En línea].
- [3] L. Nenzi, E. Bartocci, L. Bortolussi, M. Loreti y E. Visconti, «Monitoring Spatio-Temporal Properties (Invited Tutorial),» de *Runtime Verification*, Springer International Publishing, 2020.
- [4] D. Ratasich, F. Khalid, F. Geissler, R. Grosu, M. Shafique y E. Bartocci, «A Roadmap Toward the Resilient Internet of Things for Cyber-Physical Systems,» *IEEE Access*, vol. 7, pp. 13260-13283, 2019.
- [5] H. Gill, US National Science Foundation, 2006.
- [6] C. Tsigkanos, M. M. Bersani, P. A. Frangoudis y S. Dustdar, «Edge-based Runtime Verification for the Internet of Things,» *IEEE Transactions on Services Computing*, 2021.
- [7] M. Illarramendi, L. Etxeberria, X. Elkorobarrutia, J. Perez, F. Larrinaga y G. Sagardui, «MDE based IoT Service to enhance the safety of controllers at runtime,» 2019.
- [8] «<https://www.tuwien.at/en/tu-wien/about-tu-wien/>,» TU Wien. [En línea].
- [9] «<https://ti.tuwien.ac.at/cps/>,» TU Wien. [En línea].
- [10] K. Schwab, «The Fourth Industrial Revolution: what it means, how to respond,» World Economic Forum, 2016. [En línea]. Available: <https://www.weforum.org/agenda/2016/01/the-fourth-industrial-revolution-what-it-means-and-how-to-respond/>.
- [11] J. Jamaludin y J. M. Rohani, «Cyber-Physical System (CPS): State of the Art,» *2018 International Conference on Computing, Electronic and Electrical Engineering (ICE Cube)*, 2018.
- [12] G. Garatu, «Que son los Sistemas Ciber-Físicos (CPS) en la nueva Industria 4.0,» 2017. [En línea]. Available: <https://grupogaratu.com/que-son-sistemas-ciber-fisicos-cps/>.
- [13] U. Zurutuza, G. Papa, E. Jantunen y M. Albano, *The MANTIS Book. Cyber Physical System Based Proactive Collaborative Maintenance*, River Publishers, 2018.
- [14] E. Bartocci, L. Bortolussi, M. Loreti, L. Nenzi y S. Silvetti, «MoonLight: A Lightweight Tool for Monitoring Spatio-Temporal Properties,» *ArXiv*, vol. abs/2104.14333, 2021.
- [15] E. Bartocci, *Internet of Things VU*, I. f. C. Engineering, Ed.
- [16] C. L. Talcott, *Cyber-Physical Systems and Events*, 2008.

- [17] E. Bartocci, L. Bortolussi, M. Loreti y L. Nenzi, «Monitoring mobile and spatially distributed cyber-physical systems,» 2017.
- [18] J. Simmonds, S. Ben-David y M. Chechik, «Monitoring and Recovery of Web Service Applications,» *Computing*, vol. 95, pp. 250-288, 2010.
- [19] L. Nenzi, L. Bortolussi y M. Loreti, «jSSTL - A Tool to Monitor Spatio-Temporal Properties,» 2017.
- [20] M. Ma, E. Bartocci, E. Lifland, J. Stankovic y L. Feng, «SaSTL: Spatial Aggregation Signal Temporal Logic for Runtime Monitoring in Smart Cities,» 2020.
- [21] E. Visconti, E. Bartocci, M. Loreti y L. Nenzi, «Online Monitoring of Spatio-Temporal Properties for Imprecise Signals,» 2021.
- [22] A. Rivas, A. Pérez García Osvaldo y J. Hernández-Palancar, «Estado del arte sobre aplicaciones del middleware ROS,» 2016.
- [23] «MQTT,» [En línea]. Available: <https://mqtt.org/>.
- [24] «Bluetooth Low Energy,» Wikipedia, [En línea]. Available: https://en.wikipedia.org/wiki/Bluetooth_Low_Energy.
- [25] M. ABET, «Bluetooth vs Bluetooth Low Energy, what's the difference?,» *ela innovation*, 3 12 2021. [En línea]. Available: <https://elainnovation.com/en/bluetooth-vs-bluetooth-low-energy-whats-the-difference/>.
- [26] «What Is Wi-Fi?,» Cisco, [En línea]. Available: <https://www.cisco.com/c/en/us/products/wireless/what-is-wifi.html>.
- [27] S. Sendra, M. Garcia-Pineda, C. Turro y J. Lloret, «WLAN IEEE 802.11 a/b/g/n indoor coverage and interference performance study,» *International Journal on Advances in Networks and Services*, 2011.
- [28] «<https://thingsboard.io/docs/getting-started-guides/what-is-thingsboard/>,» ThingsBoard. [En línea].
- [29] «<https://www.az-delivery.de/en/products/esp32-developmentboard>,» az-delivery. [En línea].
- [30] C.-G. Guo, X.-L. Li y J. Zhu, «2010 Second International Conference on Networks Security, Wireless Communications and Trusted Computing,» *A Generic Model for Software Monitoring Techniques and Tools*, 2010.
- [31] «Fusion Middleware Concepts Guide,» Oracle, [En línea]. Available: https://docs.oracle.com/cd/E21764_01/core.1111/e10103/intro.htm#ASCON110.

- [32] «What is middleware?,» Red Hat, 2018. [En línea]. Available: <https://www.redhat.com/en/topics/middleware/what-is-middleware>.
- [33] «SOA (Service-Oriented Architecture),» IBM, 2021. [En línea]. Available: <https://www.ibm.com/cloud/learn/soa>.
- [34] «Gson,» GitHub, [En línea]. Available: <https://github.com/google/gson>.
- [35] A. Shvets, «Template Method,» Refactoring guru, [En línea]. Available: <https://refactoring.guru/design-patterns/template-method>.
- [36] A. Shvets, «Builder,» Refactoring guru, [En línea]. Available: <https://refactoring.guru/design-patterns/builder>.
- [37] A. Shvets, «Singleton,» Refactoring guru, [En línea]. Available: <https://refactoring.guru/design-patterns/singleton>.
- [38] «Application Development,» Zephyr Project, [En línea]. Available: <https://docs.zephyrproject.org/latest/develop/application/index.html>.
- [39] G. Kroah-Hartman, «I2C Drivers, Part II,» Linux Journal, 02 2004. [En línea]. Available: <https://www.linuxjournal.com/article/7252>.
- [40] «WiFi,» Arduino, [En línea]. Available: <https://www.arduino.cc/reference/en/libraries/wifi/>.
- [41] «PubSubClient,» Arduino, [En línea]. Available: <https://www.arduino.cc/reference/en/libraries/pubsubclient/>.
- [42] J. Marcel, «New Wireless Trends and,» Bluetooth, 04 2022. [En línea]. Available: <https://www.bluetooth.com/blog/new-trends-and-forecasts-for-the-next-5-years/>.
- [43] C. C. A. R. D. Kevin Townsend, «Chapter 4. GATT (Services and Characteristics),» de *Getting Started with Bluetooth Low Energy*, O'Reilly Media, Inc., 2014.
- [44] «RTC — Real-time counter,» Nordic semiconductor, Updated 2021. [En línea]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.nrf52832.ps.v1.1%2Frtc.html&cp=4_2_0_24&anchor=concept_rvn_vkj_sr.
- [45] M. A. Yasvi, «Synchronization in Distributed Systems,» Geeks for Geeks, 2020. [En línea]. Available: <https://www.geeksforgeeks.org/synchronization-in-distributed-systems/>.
- [46] M. Illarramendi, «Runtime Observable and Adaptable UML State Machine-Based Software Components Generation and Verification: Models@Run.Time Approach,» 2019.

* The icons used in some images are from www.flaticon.com