Universidade de Aveiro
2021

Francisco Bártolo Ribeiro Pinto

Framework para Sistemas de Gestão Técnica Centralizada

Framework for Centralized Technical Management Systems

Universidade de Aveiro
2021

Francisco Bártolo
Ribeiro Pinto

Framework para Sistemas de Gestão Técnica Centralizada

Framework for Centralized Technical Management Systems

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica do Doutor Paulo Bacelar Reis Pedreiras, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Pedro Nicolau Faria da Fonseca, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho aos meus pais, Ana e José.

agradecimentos / acknowledgements

Agradeço, em primeiro lugar, os meus orientadores, Prof. Paulo Pedreiras e Prof. Pedro Fonseca por toda a ajuda e confiança transmitida durante o projeto, mesmo (e especialmente) quando as coisas não correram como o planeado. Agradeço por serem excelentes profissionais, colocando sempre os interesses dos alunos em primeiro lugar.
To Thomas Georgiadis for his collaboration in this project, and for the good times spent while developing it.
Ao Eng. Luís Silva pela ajuda na parte do SDN/OpenFlow, que me poupou várias horas de trabalho.
À Diferencial, em particular, ao Eng. António Tavares, pela cooperação, conhecimento transmitido e pela disponibilização de equipamentos industriais que permitiram validar o funcionamento do sistema.
Ao Gonçalo Cardoso pela ajuda na crítica e revisão deste documento.
Agradeço também aos meus pais, Ana e José, por me proporcionarem tudo o que sempre precisei. O meu percurso até aqui foi extremamente facilitado, por isso se existem pessoas que têm mérito na conclusão deste curso são os meus pais.
À minha irmã, Inês, por me mostrar que existe muito para além da engenharia.
A toda a minha família, em particular aos meus avós, que mesmo não compreendendo muito bem o que estive a fazer, sempre me ajudaram e incentivaram a continuar.
À minha namorada, Ana, por todo o amor, carinho e compreensão que foi essencial para a realização deste projeto (e não só).
E a todos os meus amigos, em particular ao Gonçalo Cardoso, Hugo Leal, Miguel Carvalhosa, Tânia Ferreira e Diogo Lameiro, por todos os momentos de convívio, jornadas épicas, desabafos, sessões de estudo, entre outros. Foram fundamentais durante todo o meu percurso académico, que termina com esta dissertação.

Palavras Chave

Gestão Técnica Centralizada, *Internet Of Things*, EdgeX Fondry, Protocolos de Comunicação, OpenFlow, *Software-Defined Networking*.

Resumo

Sistemas de Gestão Técnica Centralizada (GTC) permitem o controlo e monitorização de diversos dispositivos instalados em chão de fábrica ou em outros ambientes, como por exemplo, escritórios, prédios ou hospitais. Estes dispositivos podem capturar dados sobre o meio onde estão instalados (sensores) ou até interagir com o mesmo meio (atuadores). Sensores de temperatura, detetores de presença ou medidores de energia podem ser utilizados para controlar sistemas de AVAC, iluminação, ou até disparar um alarme em caso de emergência. Com o aumento de aplicações baseadas em *Internet of Things* (IoT) e *Industrial Internet of Things* (IIoT), torna-se necessária a utilização de sistemas de GTC eficazes. Com estes sistemas, um técnico pode gerir múltiplos dispositivos numa unidade centralizada, sem ter de estar em contacto direto com o dispositivo e, possivelmente, sem ter de se deslocar ao edifício onde os instrumentos se encontram. Existem várias contrariedades quando se projeta um sistema de GTC, sendo uma o facto de os dispositivos que se pretende controlar utilizarem diferentes protocolos para comunicarem entre si. Com isto em mente, no âmbito deste trabalho foi desenvolvida uma *framework* para o desenvolvimento de sistemas de GTC. Esta *framework* permite que todos os dispositivos ligados à unidade central sejam controlados através de uma interface gráfica, de igual forma. Ou seja, o sistema cria uma camada que abstrai o protocolo de comunicação utilizado pelos diversos dispositivos. Neste trabalho foram utilizadas ferramentas como EdgeX Foundry, InfluxDB, Telegraf e Grafana para implementar a *framework*. O funcionamento da *framework* foi validado utilizando dispositivos comerciais (protocolo KNX e Modbus) e um dispositivo desenvolvido de raiz (protocolo MQTT). Adicionalmente, foi implementado um mecanismo de priorização de mensagens consideradas críticas, que utilizem um protocolo IP. Este mecanismo permite que uma determinada largura de banda seja reservada para o protocolo desejado. Para tal, foram utilizados princípios de *Software-Defined Networking* e OpenFlow para implementar um mecanismo que prioriza os pacotes MQTT. Para testar o sistema implementado foram comparados dois *setups*, com e sem o mecanismo de priorização de mensagens.

Keywords

Centralized Technical Management, Internet Of Things, EdgeX Fondry, Communication Protocols, OpenFlow, Software-Defined Networking.

Abstract

Centralized Technical Management (CTM) Systems allow the control and monitoring of various devices installed on the factory floor or in other environments, such as offices, buildings, or hospitals. These devices can capture data about the medium (sensors) or interact with the same medium (actuators). Temperature sensors, presence detectors, or energy meters can be used to control HVAC systems, lighting, or even trigger an alarm in an emergency. With the increasing Internet of Things (IoT) and Industrial Internet of Things (IIoT) applications, it becomes necessary to use effective CTM systems. With these systems, a technician can manage multiple devices in a centralized unit without having to be in direct contact with the device and possibly without having to travel to the building where the instruments are. There are several setbacks when designing a GTC system. One of them is that the devices to be controlled use different protocols to communicate with each other. With this in mind, within the scope of this work, we developed a framework for developing CTM systems. This framework allows all devices connected to the central unit to be controlled through a graphical interface in the same way. That is, the system creates a layer that abstracts the communication protocol used by the various devices. In this work, tools such as EdgeX Foundry, InfluxDB, Telegraf, and Grafana were used to implement a framework. The functioning of the framework was validated using commercial devices (KNX and Modbus protocols) and a device developed from scratch (MQTT protocol). Additionally, a mechanism for prioritizing messages considered critical, which uses an IP protocol, was implemented. This mechanism allows a specific bandwidth to be reserved for the desired protocol. To this end, Software-Defined Network and OpenFlow principles were used to implement a mechanism that prioritizes MQTT packages. Two setups were compared to test the implemented system, with and without the message prioritization mechanism.

Contents

List of Figures

List of Tables

Glossary

AGPL	Affero General Public License
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
BACnet	Building Automation and Control
CAN	Controller Area Network
CLI	Command Line Interface
CoAP	Constrained Application Protocol
CTM	Centralized Technical Management
DBMS	Database Management System
DALI	Digital Addressable Lighting Interface
EtherCAT	Ethernet for Control Automation Technology
ETS	Engineering Tool Software
GPL	GNU General Public License
GUI	Graphical User Interface
HVAC	Heating, Ventilation, and Air Conditioning
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IIoT	Industrial Internet of Things
IDE	Integrated Development Environment
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
LED	Light-Emitting Diode
LGPL	GNU Lesser General Public License
MAC	Media Access Control
MIT	Massachusetts Institute of Technology
MQTT	Message Queuing Telemetry Transport
NS	North-to-South
OPC-UA	Object Linking and Embedding for Process Control - Unified Architecture
OS	Operating System
Profibus	Process Field Bus
PLC	Programmable Logic Controller
QML	Qt Modeling Language
QoS	Quality of service
REST	Representational State Transfer
RTU	Remote terminal unit
SDN	Software-Defined Networking
SDK	Software Development Kit
SN	South-to-North
SSH	Secure Shell Protocol
SQL	Structured Query Language
SSPL	Server Side Public License
TCP	Transmission Control Protocol
TG	Traffic Generator
TLS	Transport Layer Security
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VLAN	Virtual Local Area Network
XML	Extensible Markup Language

CHAPTER 1

Introduction

Centralized Technical Management (CTM) systems play an important role in the development of Internet of Things (IoT) and Industrial Internet of Things (IIoT) applications. They allow a user to monitor and actuate on every device connected to them by creating a digital twin of the physical devices and sensors. A digital twin is a virtual representation of the real devices so that the user only needs to interact with the digital twin to actuate on a real device. This virtualization process enables a convenient and effortless interaction with all the devices. For that, a CTM system must implement an abstraction layer in a way that the system supports devices using different communication protocols.

Furthermore, having all the data gathered from the sensors centralized in one location allows the user, or third-party software like machine learning algorithms, to find and explore possible correlations between the data. However, to implement CTM systems, we must address some concerns. Three major concerns are the dispersion of devices in a complex of buildings, devices using different communication protocols, and timeliness concerns, particularly when the network is stressed.

Usually, buildings like factories or hospitals have multiple devices and sensors. These devices are not typically close to each other, so it is necessary to analyze them one by one when maintenance is required. The maintenance process is time-consuming, and that time increases with the number of devices in the building. Additionally, a proper system can correlate the data gathered from the sensors in different rooms. That correlation could lead to a more efficient use of the available resources, for example, Heating, Ventilation, and Air Conditioning (HVAC) systems. However, this actuation and data analysis can only be done if all devices are connected to a centralized unit.

The physical distance is not the only difficulty to connect all the devices in one building. The fact that these devices use distinct communication protocols is also an adversity. Take, for example, a case where a company wants to control lamps based on motion sensors. If the interface with the lights is done via Digital Addressable Lighting Interface (DALI) [1], it is required to use DALI based motion sensors. Suppose, at some point, the company decides to replace the lamps' control interface to use the KNX protocol [2]. In that case, it is not only necessary to make changes to the lamps but also to the motion sensors since the DALI ones would not work.

Finally, Internet Protocol (IP) protocols, like Transmission Control Protocol (TCP) or User Datagram Protocol (UDP), are broadly used in IoT applications, and for many cases, they work well, despite not having real-time specifications. However, as the number of devices connected to a network increases, so does the network traffic. This increase can lead to saturation periods in the network, leading to more significant delivery times or even packet loss. Although this may not be a problem in some scenarios, it may be critical in other ones. For example, an emergency button that halts a machine in a factory when necessary requires that when an emergency occurs, the stop command must arrive at its destination in a bounded amount of time, independently of the network load conditions. This aspect may force the system designer to choose a specific communication protocol instead of a more convenient one.

To conclude, within the scope of this project, it was designed, implemented, validated, and tested a framework to develop CTM systems that addresses the three concerns described before. The functional requirements of the framework developed are defined in Section 1.1, whereas Section 1.2 and Section 1.3 present the objectives and structure of this dissertation, respectively.

1.1 Functional Requirements

The system was developed in collaboration with the company *Diferencial - Electrotécnica Geral Lda*, which defined the framework's functional requirements to satisfy common customer needs:

- Having all the data collected from both the devices and the user stored in a centralized database.
- Developing a Graphical User Interface (GUI) that allows a user to monitor and actuate on the connected devices. These procedures must be independent of the device and protocol in question.
- Providing the capability of adding to the system multiple devices that use various communication protocols.
- Allowing the GUI to be physically separated from the CTM system and devices.

1.2 Objectives

This dissertation focuses on four specific topics, that are:

- Analyzing possible approaches and mechanisms to implement CTM systems and custom IoT devices.
- Validating the framework developed using commonly used industrial devices, with different communication protocols.
- Evaluating the scalability of the framework implemented.
- Designing and validating a solution to prioritize critical messages in situations where the network is saturated.

1.3 Document Structure

This document is divided into eight chapters, whose content is the following:

- *Chapter 2 - Background*: This chapter introduces the most relevant topics relevant for this dissertation. It starts to review the existing CTM system architectures and relevant tools to develop them. Then it presents communication protocols relevant for IoT and IIoT. Finally, it introduces Software-Defined Networking (SDN) and OpenFlow as a solution to prioritize sensitive messages.
- *Chapter 3 - Methodology*: This chapter is dedicated to describing the methodology used to solve the proposed problem. The chapter justifies selecting the chosen tools and briefly presents the steps to implement, validate, and test the framework.
- *Chapter 4 - Architecture*: This chapter focuses on the system architecture. It presents the structure of the whole system and a brief description of the system components and their interfaces.
- *Chapter 5 - Framework for CTM Systems*: This chapter shows how the system purposed was implemented, including the devices, IoT gateway, database and GUI. The chapter also presents the definition of interfaces between the system components.
- *Chapter 6 - Message Prioritization Mechanism*: This chapter presents the approach chosen to prioritize critical messages. Starting from the components required, then implementing this part of the system, and finally, the definition of two setups for testing purposes.
- *Chapter 7 - Tests, Results, and Discussion*: This chapter is dedicated to the tests performed and their primary outcomes. Then it expounds considerations about the suitability of the framework to develop CTM systems. Finally, it discusses the results obtained.
- *Chapter 8 - Conclusion*: This chapter presents the document closure, the critical analysis about the work developed and main results, and the future work proposal.

CHAPTER 2

Background

This chapter aims to introduce the necessary background to develop and validate a framework for CTM systems. Section 2.1 is dedicated to defining the high-level architecture used in this context, describing the main components of the selected architecture, and presenting possible implementations or toolkits to implement such features.

Furthermore, Section 2.2 presents some communication protocols that can be used to connect devices to a CTM system. The protocols selected are commonly used in different applications, such as industrial environments, buildings, or IoT systems.

Lastly, Section 2.3 presents the required knowledge to implement the feature of prioritizing critical packets over the remaining traffic in an IP network. This section also introduces some topics on SDN, particularly the OpenFlow protocol that was used in the scope of this dissertation.

2.1 CTM Systems

A typical CTM system requires two main components: a gateway and an infrastructure (composed, for instance, by a database and a GUI), as presented in Figure 2.1. The gateway (Subsection 2.1.1) is in control of the interaction with the physical devices, eventually using distinct communication protocols and makes the interface between them and the infrastructure.

On the other hand, the infrastructure is responsible for storing data (either collected from the devices or the users) and implementing the user interface. Its components can be deployed on-premises or remotely, in a cloud system. In both scenarios, a simple infrastructure requires at least two components: a database (Subsection 2.1.2) and a GUI (Subsection 2.1.3). A database is necessary to store the data gathered from the devices to create their digital twins. It is the central component of any IT System's infrastructure. Moreover, the GUI is responsible for displaying the data gathered from the devices and for receiving commands from the user. These commands can be, for example, a *GET* or a *PUT* instruction. The former reads a certain sensor or register asynchronously from the device, whereas the latter actuates on it.

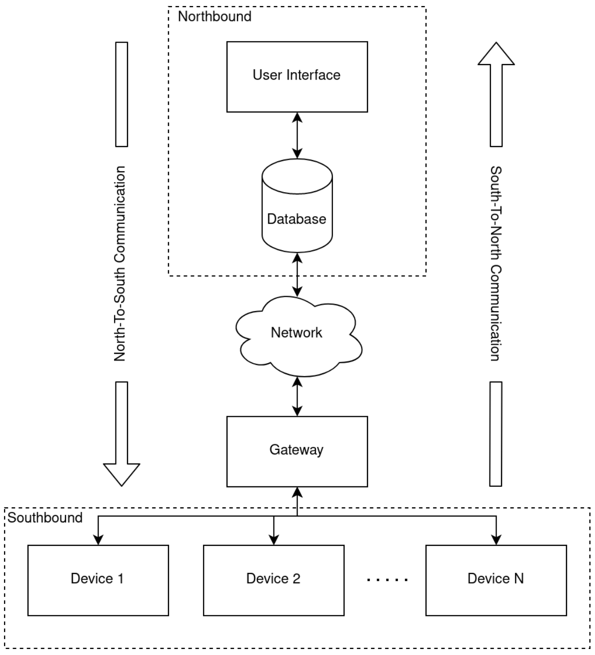


Figure 2.1: Simple CTM System Arquitecture.

2.1.1 Gateway

The gateway is the central element of a CTM system. It is responsible for creating an abstraction layer so that the interaction with all devices is the same, no matter what protocol each device uses. Typically, by convention, the terms *north* and *south* appear referring to the devices and infrastructure, respectively. In between the north and south parts of the system lies the so-called gateway. So, for instance, the phrase *south to north communication* indicates a process where data generated by the device is acquired by the gateway and sent to the infrastructure.

Some gateways offer additional tools such as management, data processing, security features, and different connectors to interact with the northern and southern components of the system. These connectors are provided to accelerate the development process, and they support the most commonly used communication protocols. Additionally, most gateway frameworks provide documentation and Software Development Kits (SDKs) to implement custom connectors.

There are multiple frameworks and toolkits available that meet the requirements to develop a gateway for the CTM system. We analyzed several solutions, particularly *EdgeX Foundry*, *Node-RED*, and *ThingsBoard*, which were studied in more detail and are presented below.

EdgeX Foundry

EdgeX Foundry [3] is an edge IoT middleware platform, being between the cloud systems and the devices that sense and actuate on the physical domain. EdgeX Foundry is a software implementation that leverages cloud-native principles (for instance, loosely-coupled microservices and platform-independence) but was designed to meet specific needs of the IoT edge. It is a vendor-neutral and open-source solution that is distributed under Apache

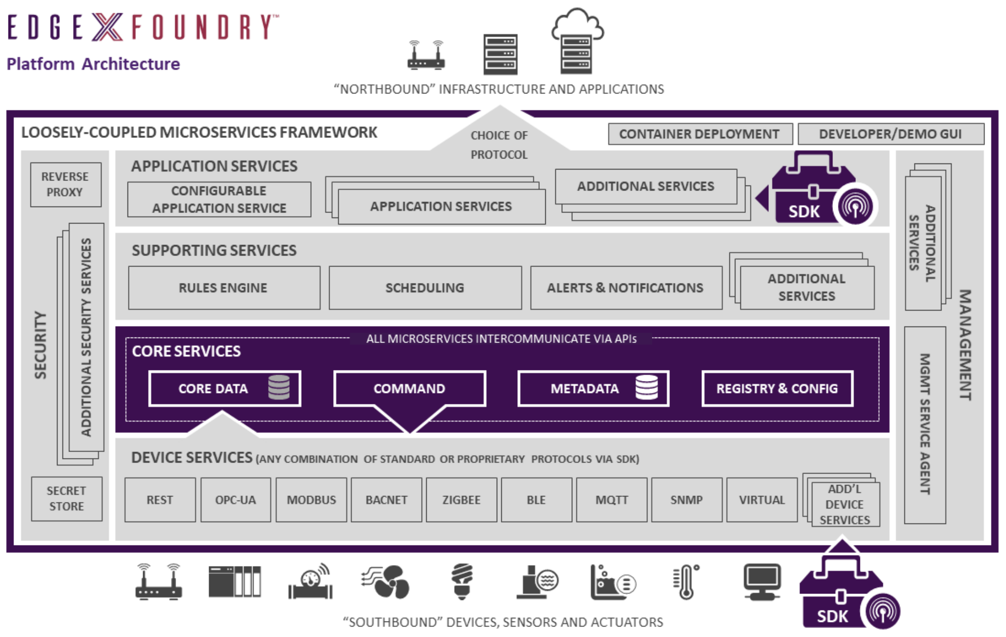


Figure 2.2: EdgeX Foundry Platform Architecture (from [3]).

2.0 license. Dell initially created EdgeX Foundry, but since 2017 it has been one of the projects from the Linux Foundation. This platform was designed to be Operating System (OS) agnostic, so it is compatible with Linux, Windows, and macOS machines.

In short, EdgeX Foundry is a collection of microservices, implemented in docker containers, that are distributed in four main layers (Device, Core, Supporting, and Application Services) and two additional layers (Security and Management Services), as presented in Figure 2.2. The other layers contain microservices to protect the data collected and the devices themselves and allow the administrator to inspect and manage the system at a central point.

Moving to the four main layers and starting from the south, the **device services** offer a group of edge connectors to control the devices and sensors connected to the gateway. The framework has already implemented connectors for protocols like Message Queuing Telemetry Transport (MQTT), Modbus (Remote terminal unit (RTU) and TCP), Object Linking and Embedding for Process Control - Unified Architecture (OPC-UA). In addition, developers can create custom connectors for communication protocols that are not implemented in the framework. Developers may use the Device Service SDK to implement them. At the moment, there are two SDKs one written in Go Lang and the other in C.

Next, the **core services** consist of four microservices - core data, core command, core metadata, and configuration and registry. These services contain the persistent storage with data collected by the devices and specifications about the devices themselves. It also defines the available commands to interact with the different devices and provides the tools to provision new machines.

Above the core services are the **supporting services** layer that includes alert and notification, scheduling, and rules engine microservices. These microservices are specific software programs that perform particular tasks, interacting with the southbound devices through core services and northbound infrastructure through the application services. These tasks include edge analytics duties, triggering actions at specific times, logging systems, and alerting the system manager in case of an emergency.

The fourth and last main layer contains the **application services**. The microservices included in this layer provide mechanisms to export data from the EdgeX Foundry gateway to external endpoints. There can be cloud-based systems (like Azure IoT Hub, AWS IoT, or Google IoT Core) or on-premises applications. At the moment, EdgeX Foundry supports MQTT and Hypertext Transfer Protocol (HTTP) endpoints. The application services provide tools to prepare and groom data before sending it; developers can include these functions in a customizable export functions pipeline and, additionally, use the SDK to create their application service.

There are three different approaches when using the EdgeX Foundry ecosystem: user, developer, and hybrid modes. In the first approach, to start EdgeX services it is required to download and activate the correct docker containers. For that, the EdgeX development team provides docker-compose files that contain instructions to download the docker containers and the order they should start and input parameters for each container. These files speed the deployment process and can be found in [4]. If, on the other hand, one wants to make changes to the platform, it is possible to do so by downloading the source code that is distributed in C and Go Lang. The EdgeX Foundry Documentation, [3] provides the list of software dependencies and the instructions to install, build and run EdgeX from the source code. This solution is called a developer approach. The last approach is called hybrid since it allows the coexistence of dockerized and non-dockerized microservices. This method allows developers to make changes to one microservice to run it in developer mode, while the remaining ones are deployed in the more convenient user mode.

ThingsBoard IoT Gateway

ThingsBoard [5] is an open-source IoT platform with processing, visualization, and management tools to actuate on the data collected from the IoT devices. It was designed by ThingsBoard, Inc. in 2016 and is licensed under Apache 2.0 license.

ThingsBoard IoT gateway, whose simplified architecture is shown in Figure 2.3, aims to connect devices with multiple platforms to the ThingsBoard system. It was developed to run on Linux machines and supports Python 3 releases since version 3.5. To interact with the devices, it supports multiple communication protocols, like MQTT, Controller Area Network (CAN), and OPC-UA. It is also possible to develop a custom connector for communication protocols that are not implemented. This process may be done in Python 3, via the Application Programming Interface (API) and documentation available in [5]. Furthermore, the gateway has an embedded database to save the data gathered, when network or hardware failures occur. For instance, if there was no backup embedded database and the infrastructure database failed, the data gathered during that failure would be lost. Finally, the communication to the ThingsBoard system is carried out via the MQTT protocol.

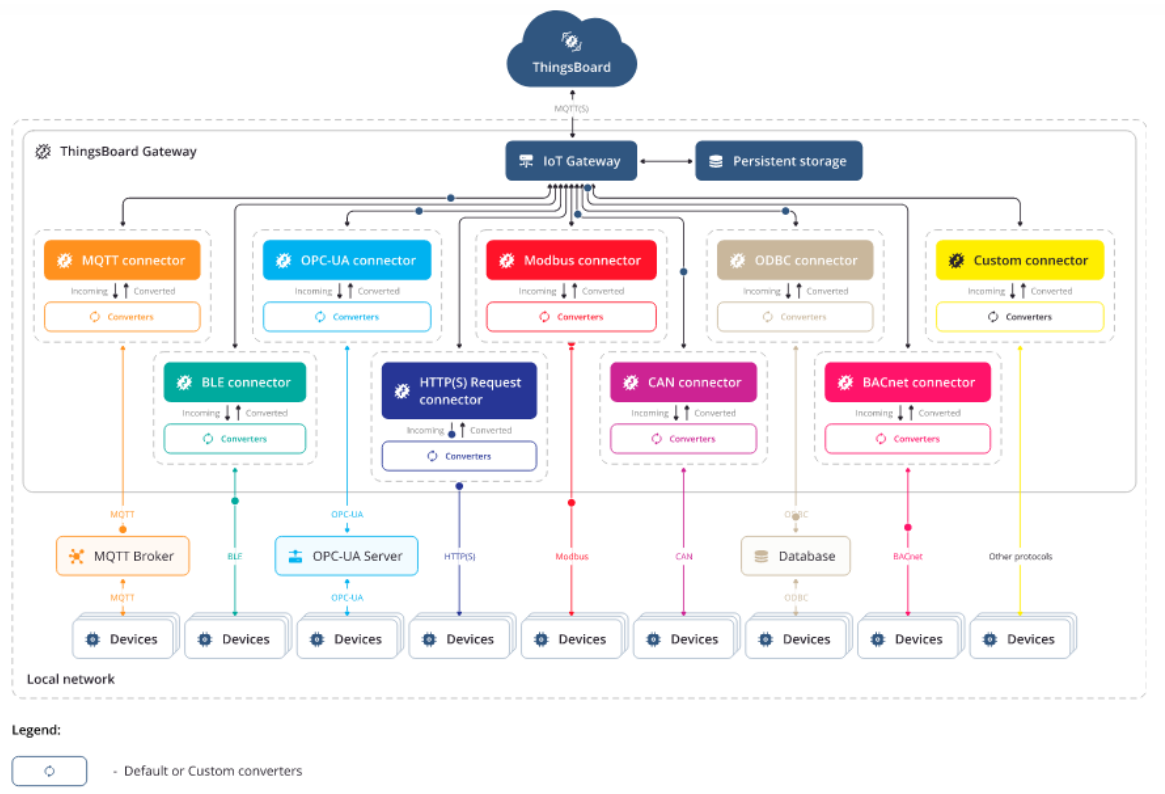


Figure 2.3: ThingsBoard IoT Gateway Architecture (from [5]).

Additional features, like edge analytics, data processing, security, management, and dashboards, are not part of the ThingsBoard IoT gateway but the ThingsBoard system. There are two versions of this software, the community, and professional editions. The latter includes all the features of the former, and it has additional ones. White-labeling, advanced management and administration tools, and the integration with existing IoT platforms are some of the advantages of acquiring the professional edition.

Some use cases of the ThingsBoard platform include energy consumption monitorization, intelligent farming control, and fleet tracking.

Node-RED

Node-RED [6] is a software tool whose objective is to enable communication between APIs, online services and hardware devices, in one place. The Node-RED project was started in 2013 by Nick O'Leary and Dave Conway-Jones from IBM's Emerging Technology Services team, and it became open-source in the same year. From 2016 to the present, it has been hosted by the JS Foundation. Node-RED is distributed under Apache 2.0 license.

In contrast with EdgeX Foundy or ThingsBoard, Node-RED is a flexible multipurpose tool. Hence, it was not developed specifically to develop IoT edge gateways. Nevertheless, it supports multiple communication protocols, such as MQTT, Modbus, and HTTP tools to communicate with external databases, like MySQL, InfluxDB, or Cassandra, and a framework to develop simple user interfaces.

The development of Node-RED-based applications is done via flow-based programming. It

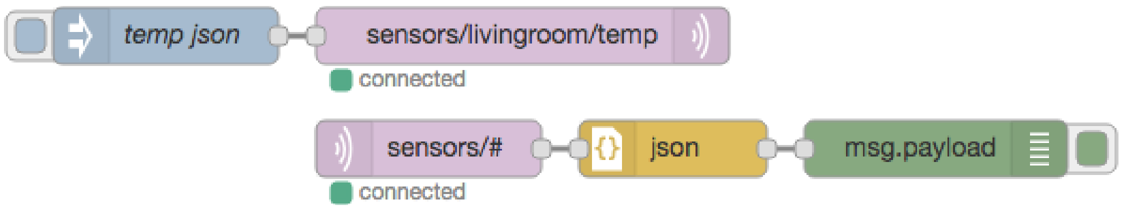


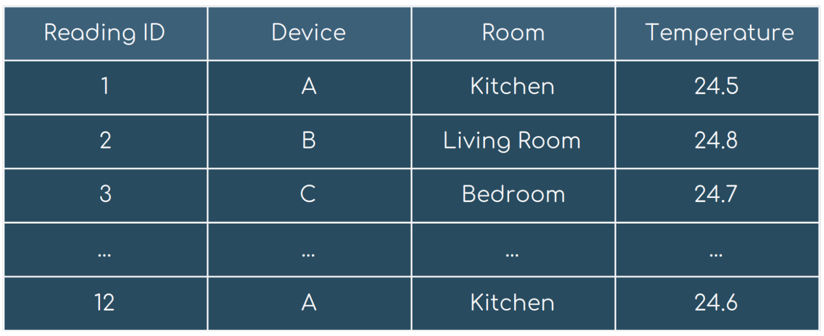
Figure 2.4: Node-RED flow example (from [6]).

is lightweight so that it can run on low-cost hardware, and it runs on multiple platforms and OSs. The applications developed are event-driven and are called flows, consisting of nodes connected to form a pipeline. The nodes are functional blocks that perform specific tasks (like converting the input data to Extensible Markup Language (XML) format or inserting the input data in a database). Developers can implement their algorithms in the nodes using JavaScript. When one node is triggered (for example, when an HTTP message is received or when a predefined amount of time has passed), it outputs a message. That message is sent to the next node, which processes it and sends it to the following node. This happens until the message reaches its endpoint. In the end, the flows developed can be stored conveniently in the JavaScript Object Notation (JSON) file.

Figure 2.4 presents an example of a flow developed in Node-RED to subscribe to the topics "sensors/#". When an MQTT packet is received, it converts the received message to JSON format and then prints the parsed message in the terminal (pipeline's endpoint).

2.1.2 Database

Databases are mechanisms used to store data. The software programs that allow a user or a machine to interact with the database are called Database Management Systems (DBMSs). The most commonly used databases are relational ones. Relational databases are divided into tables, each containing a different set of columns (fields). The columns have a label and data type associated and describe the stored records in each row. Considering the example in Figure 2.5, the fields *Reading ID*, *Device*, *Room*, and *Temperature* describe the four presented records. Structured Query Language (SQL) is the most famous programming language to interact with Relational DBMSs, such as MySQL, SQLite, or PostgreSQL.



Reading ID	Device	Room	Temperature
1	A	Kitchen	24.5
2	B	Living Room	24.8
3	C	Bedroom	24.7
...	...	...	...
12	A	Kitchen	24.6

Figure 2.5: Relational Database Structure Example.

However, the strict and rigid architecture of relational databases is not the best solution for all scenarios. When considering IoT applications, a more flexible design would ease the storage of the data collected by the "things". Since each device produces different data types, the information produced is typically called "unstructured data". To better suit cases with big unstructured data, another kind of database grew in popularity, the non-relational databases. Some solutions were designed to be schema-less and flexible. The relational and non-relational databases are usually called SQL and NoSQL databases, referring to the programming language used to access the data.

NoSQL databases can be designed for different purposes. Some of the most used architectures include wide-column stores (e.g., Cassandra), document stores (e.g., MongoDB), key-value data stores (e.g., Redis), graph stores (e.g., Neo4J), and time-series stores (e.g., InfluxDB).

In IoT applications, two main types of data need to be stored: information about the devices (like configurations, attributes, and interfaces) and the data gathered by them. The latter are time-series data types and represent the vast majority of the data that needs to be stored. This data can be used to analyze the device's status or to feed ML algorithms.

We analyzed different types of NoSQL databases, considering scalability, suitability, and efficiency. In particular, *InfluxDB*, *Apache Cassandra*, and *MongoDB*. The objective of this study was to select the most appropriate solution to incorporate the infrastructure of the CTM system.

InfluxDB

InfluxDB [7] is an open-source time-series database released by InfluxData in 2013. It was developed in Go Lang and is licensed under an Massachusetts Institute of Technology (MIT) license. As the database is time-oriented, it can efficiently store IoT sensor data and real-time analytics. Currently, InfluxDB's latest version is 2.0. InfluxDB can be seen as a toolkit rather than a simple database, since it comes with a user interface, processing, and monitoring tools. Flux is the query and scripting language that allows users or external systems to interact with the DBMS. Its retention policy is convenient for most IoT applications since it allows the automatic deletion of obsolete data.

Apart from InfluxDB itself, InfluxData provides: InfluxDB Cloud, that is InfluxDB as a cloud-based service; InfluxDB Enterprise, a clustered version of InfluxDB to increase availability; InfluxDB Templates, a set of tools to develop monitoring solutions; and Telegraf. Telegraf [8] is a server agent developed in Go Lang that can collect data from multiple sources and write them in various outputs, namely databases. Telegraf was developed in a plugin-based architecture to suit multiple scenarios and has minimal hardware requirements.

Apache Cassandra

Apache Cassandra [9] is an open-source NoSQL database developed by Apache Software Foundation and released in 2008. Currently, the last stable release is 3.11. Cassandra belongs to the Wide-Column store category of NoSQL databases and features a SQL-like querying language, Cassandra Query Language. It is well-known for its high availability and allowing partitioning so that multiple endpoints can access the database faster and prevent the risk of losing data due to hardware failures.

MongoDB

MongoDB [10] is a source-available document-based database. It was released in 2009 by MongoDB Inc. It has a Server Side Public License (SSPL) that is a modified version of GNU General Public License (GPL) 3.0, which requires the services that make use of software licensed with SSPL (like MongoDB) to be open-source under the same license. MongoDB is a NoSQL database that uses JSON-like documents and stands out for having high availability and tolerance to partitioning. In addition, MongoDB provides optimizations for time-series data, for instance the ability to distribute nodes from different clusters based on timestamps.

2.1.3 Graphical User Interface

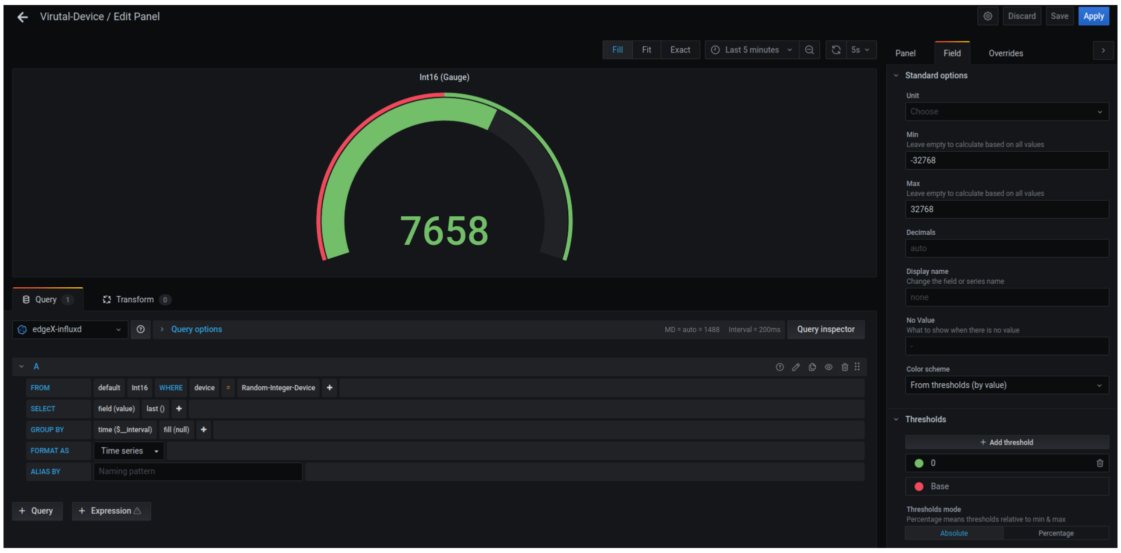
GUIs play a crucial role in the scope of CTM systems, in particular in the infrastructure part. They are the elements responsible for providing data and insights about the data generated by the devices connected to the systems. Moreover, it is the GUI that allows the users to manage and control the devices remotely. So, a user can manage, for example, an entire building or factory from one place, being able to analyze sensor data, like temperature, actuate on devices, like HVAC systems, or inspect possible malfunctioning devices.

When developing user interfaces for CTM systems, the two main technical requirements are graphically presenting the data stored in a database and interacting with the user. For that, two different types of toolkits can be considered: drag-and-drop solutions and widget-based applications. The former prioritizes simplicity and standardization, whereas the latter prioritizes customization. Typically, drag-and-drop-based toolkits offer solutions in which representing data that is stored in a database is a relatively simple task. In addition, they provide various visualization schemes to allow the users to interpret different types of data. Finally, a web browser is responsible for making the interface between the user and the system when using this approach.

On the other hand, widget-based applications provide the necessary tools to implement a fully customizable desktop application. Most toolkits offer templates to increase productivity, but the developer has complete control over how the GUI should look like and should behave. In the scope of this dissertation, different toolkits were considered, and two were analyzed in detail, Grafana (web-based tool) and Qt (application-based tool).

Grafana

Grafana [11] is an open-source web-based monitoring solution created by Grafana Labs in 2014. It is a web application that allows the dashboards to be developed and accessed on multiple platforms. At the moment, it is licensed under an Affero General Public License (AGPL) 3.0 license. Grafana Labs developed the toolkit in Go Lang and Typescript, and at the time of this writing, its latest version is 7.5. Among other characteristics, the integration between the data stored in the most commonly used databases and Grafana is quite simple (namely time-series databases like InfluxDB). It allows multiple and varied ways to present



edgeX-influxd
MD = auto = 1488 Interval = 200ms

A
FROM default Int16 WHERE device = Random-Integer-Device
SELECT field (value) last ()
GROUP BY time ($__interval) fill (null)
FORMAT AS Time series
ALIAS BY Naming pattern

Unit: Choose
Min (Leave empty to calculate based on all values): -32768
Max (Leave empty to calculate based on all values): 32768
Decimals: auto
Display name (Change the field or series name): none
No Value (What to show when there is no value): -
Color scheme: From thresholds (by value)
Thresholds: ● 0; ● Base
Thresholds mode (Percentage means thresholds relative to min & max): Absolute | Percentage

Figure 2.6: Developing a User Interface Panel Using Grafana.

the data. Grafana Labs provide, alongside the open-source version, a cloud-based service of Grafana and an enterprise version with additional features.

After installing the required software, a Grafana dashboard can be built using the web interface provided, as presented in Figure 2.6. Firstly, it is necessary to add a data source, which is typically a database, then developers can add some panels to the dashboard to represent the data. These panels can contain visualization methods like pie charts, scatter plots, heatmaps, and more. Once chosen the visualization method, the data from the database to be presented can be specified using an SQL-like interface. Here the data can not only be selected but also transformed. Additionally, Grafana allows the user to add annotations to their data and define alerts.

Qt

Qt [12] is a cross-platform toolkit to develop GUIs, created in 1995. The Qt Company manages it, and its latest stable release is version 6.1. Qt is distributed with either commercial licenses and open-source licenses, namely, GPL 2.0, GPL 3.0, and GNU Lesser General Public License (LGPL) 3.0. This toolkit was developed in C++ and allowed developers to create GUIs for platforms like Linux, Android, Windows, macOS, as well as embedded platforms.

For development, the framework provides design tools (Qt Design Studio) in which the user can design the visible part of the user interface, as shown in Figure 2.7. Developers can do this design process either with the templates already provided by Qt or by creating custom ones. Once completed the front-end part, the developers can use the cross-platform Integrated Development Environment (IDE) Qt Creator. This framework, among other features, allows developers to interconnect the front-end part of the system with the back-end. This process is done mainly using C++ and Qt Modeling Language (QML). Qt Creator also provided the necessary tools to add pre-built visual components and animations to the interface.

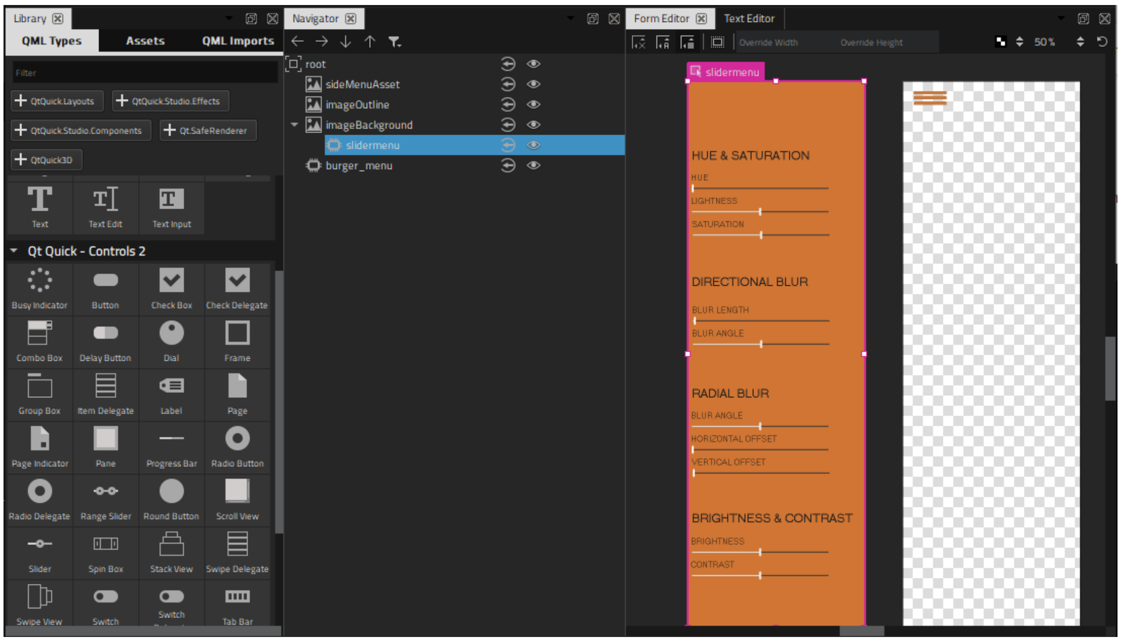


Figure 2.7: Developing a GUI using Qt Design Studio (from [12]).

2.2 Communication Protocols

CTM systems can be applied to multiple scenarios, for example, for buildings, industrial environments, or even in IoT applications. Each application requires its own specific devices, and thus, communication standards.

For building automation systems, protocols like KNX [2], Building Automation and Control (BACnet) [13], and DALI [1] are used. The former two were designed to control and manage a wide variety of devices, like HVAC systems, blinds, lighting, or access control. The latter was designed only for controlling lights.

Standards like Modbus [14], Process Field Bus (Profibus) [15], or Ethernet for Control Automation Technology (EtherCAT) [16] are most commonly adopted regarding industrial automation. These protocols emerge in Fieldbus systems in industrial environments and whose objective is to establish a network to allow industrial devices, such as Programmable Logic Controllers (PLCs), to control factory floor sensors and actuators. In addition, some protocols, like EtherCAT, offer real-time guarantees.

When considering IoT applications, three of the mostly used communication standards are MQTT [17], Advanced Message Queuing Protocol (AMQP) [18], and Constrained Application Protocol (CoAP) [19]. The protocols used in these situations need to be lightweight since they typically need to be implemented in low-cost hardware.

The following three subsections briefly present the KNX, Modbus, and MQTT protocols, which are particularly relevant for the scope of this work.

2.2.1 KNX

KNX [2] is a standard developed mainly for building automation. It supports different transmission media, such as twisted-pair cable, Ethernet, radiofrequency, or power line.

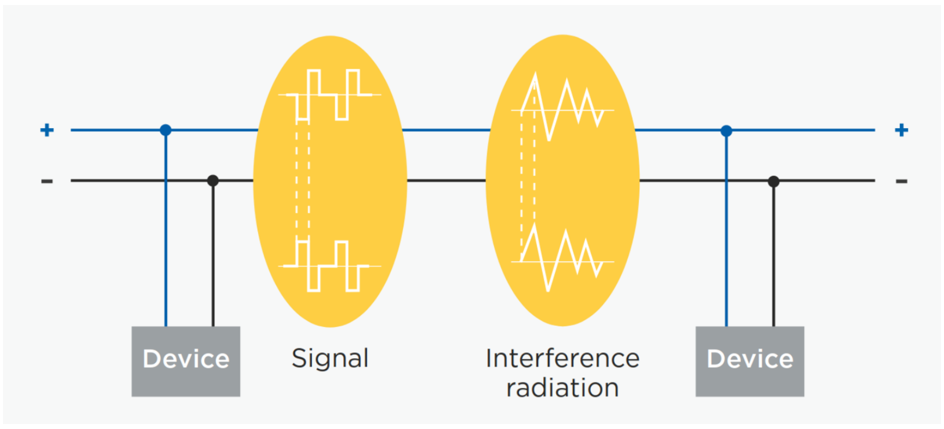


Figure 2.8: KNX TP Data Transfer (from [20]).

.

Compared to other protocols, KNX has the advantage of not requiring a central management device, since the nodes, once connected to the bus and properly configured, can communicate to each other and actuate accordingly. Finally, KNX provides an Engineering Tool Software (ETS) to allow developers to manage the devices and define their actions conveniently.

The most common communication medium used in KNX is the twisted-pair (TP), given its simplicity of installation and cost-effectiveness. The KNX TP requires a particular power supply, since the data transmission is done using the same medium used to power devices, as presented in Figure 2.8. For this reason, although the KNX bus rated voltage is 24 V, the power supplied to the system is 30 V. A 9 V range, from 21 V to 30 V, is given to the devices to allow them to transmit data.

The information is exchanged by the devices, serially and at a rate of 9.6 kbit/s, using telegrams. A telegram is a sequence of eight to twenty-three bytes separated into four fields: Control, Address, Data, and Checksum. The first one is used to define the transmission, the second one carries the sender and receiver addresses, the third contains the telegram payload, and the last one is used for parity checks. [20]

2.2.2 Modbus

Modbus [14] is a communication protocol developed for industrial applications, in particular, to interconnect PLCs. There are many variations of the protocol. Modbus can use different transmission media, transport layer protocol, or error-detection mechanisms in distinct standard versions. Two of the mostly used Modbus versions are Modbus RTU and Modbus TCP/IP. The former uses serial communication and requires an error-detection mechanism, whereas the latter needs to be connected to an IP network and does not require error-detection tools. Additionally, in Modbus versions that use serial communication as the transmission media, the master node is responsible for establishing the communication. In contrast, the slave devices should only respond to the requests. On the other hand, when Internet Protocols are used, both master and slaves can initiate communication.

The data stored in the Modbus salve devices can be divided into four categories: Coils, Discrete Inputs, Input Registers, and Holding Registers. The first two are used to store single-bit values, whereas the registers hold 16-bit values. The difference between input and holding register lies in the fact that the former may only be read, while the latter can be read from and written.

A Modbus RTU frame is composed of four elements: Slave address (1 byte), Function (1 byte), Data (N bytes), and Cyclic Redundancy Check (2 bytes). The slave address identifies the slave. The function codes allow different possible actions, including reading from or writing to a holding register or group of registers. The data field contains the message's payload, for example, the address of the register to read from. Finally, the last field is used for error detection during transmission.

2.2.3 MQTT

MQTT [17] is one of the mostly used standards regarding IoT applications. This protocol is used mainly for being lightweight and for being reliable. The MQTT protocol runs over TCP/IP, and it has two types of agents: a message broker that is a centralized device that receives all the MQTT messages, and the clients that can communicate with each other via the broker.

In MQTT the terms *publish* and *subscribe* arise to the place of *send* and *receive*. A message is said to be published when a client sends it to the MQTT broker. Each MQTT message has a topic associated with it, so for client A to receive messages from client B, it must subscribe to the topic where client B is publishing its messages. Following this example, if client A has subscribed to the topic "temperature" every time client B publishes a message on that same topic, the MQTT Broker sends that message to client A. The same would happen to other clients that had subscribed to the same topic. Figure 2.9 presents graphically the example described before, where client A is one of the devices on the right side of the figure, whereas client B is the one on the left.

Each MQTT connection can specify a Quality of service (QoS). There are three types of QoS - at most once, at least once, and exactly once. The QoS 1 and 2 are implemented based on acknowledge or handshaking mechanisms and consequent retransmissions. On the other hand, QoS 0 takes no additional measures to assure that the message was received correctly.

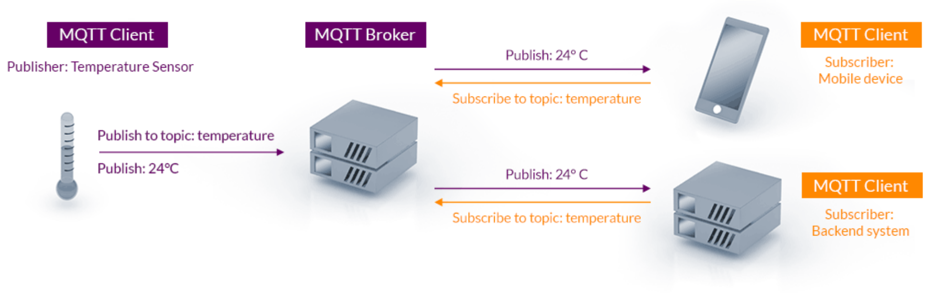

MQTT Client
Publisher: Temperature Sensor
Publish to topic: temperature
Publish: 24°C
MQTT Broker
Publish: 24° C
Subscribe to topic: temperature
Publish: 24° C
Subscribe to topic: temperature
MQTT Client
Subscriber: Mobile device
MQTT Client
Subscriber: Backend system

Figure 2.9: Example of MQTT Communication (from [17]).

.

2.3 Message Prioritization

One concern regarding Industry 4.0 is the fact the existent network infrastructure may not be ready to accommodate the increasing number of IIoT applications. In industrial scenarios, the traffic generated in a network can surpass the maximum values defined by the existing hardware. A situation like this could jeopardize the implementation of IIoT devices for critical scenarios. Existing methods, such as handshaking and re-transmission mechanisms, are not helpful when the network must deliver the data within a certain time interval. A message prioritization mechanism could be a solution to address this issue. For example, a portion of the available bandwidth could be reserved for time-sensitive messages to ensure that they are received on time even if the network is stressed.

This prioritization mechanism can be based on the SDN technology, in particular using the OpenFlow protocol. A brief introduction of these tools is presented in the following subsections.

2.3.1 Software Defined Networking

The Software-Defined Networking paradigm emerged mainly to solve the difficulty of managing complex networks. Traditionally, this process was done device-by-device with the possibility of having devices from different vendors, and thus, heterogeneous configuration interfaces. Due to this issue, complex networks, for instance in industrial environments, were static and almost untouchable once deployed [21].

So, the SDN architecture, presented in Figure 2.10 was designed to separate the data and control planes of a network, and thus allowing a central entity to control the whole network. The decisions made by this central entity, called the controller, are sent to data forwarding devices (e.g., switches). These hardware devices forward packets to a specific egress port, drop, flood, or even change their content. The interface between the data and control planes, i.e., the controller and the devices, is done via a southbound signaling protocol, like OpenFlow. Among other features, SDN provides some QoS mechanisms such as prioritization and bandwidth reservations.

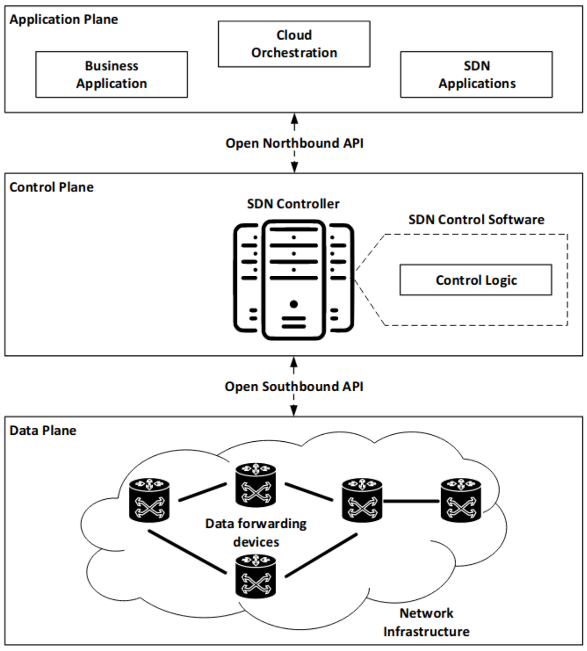


Figure 2.10: Generic SDN architecture (from [21]).

2.3.2 OpenFlow Protocol

OpenFlow is the mostly used standard to interface between data-forwarding devices (like switches) and the controllers. So, accordingly, an OpenFlow-based architecture requires one or more OpenFlow controllers and one or more OpenFlow switches, whose architecture is presented in Figure 2.11.

Apart from the SDN controller-related tasks, an OpenFlow controller implements the required mechanisms to allow the communication and configuration of the OpenFlow switch [21].

In short, an OpenFlow switch consists of an OpenFlow pipeline, a control channel, and a group table. The control channel, composed of one or more OpenFlow channels, provides the interface with the OpenFlow controllers, based on Transport Layer Security (TLS). The OpenFlow pipeline and group table perform the traffic forwarding and look-up. The former is composed of one or more flow tables. These flow tables contain multiple flow entries, each one having a set of match fields and actions. When a received frame matches a specific flow, the measures defined by the flow are applied. For example, network managers can specify these actions to drop the packet, forward it to an output port, or set a specific egress queue. In addition, these queues could have specifications in terms of priorities and bandwidth. Finally, a meter table is an optional feature of an OpenFlow switch that provides QoS services, like rate-limiting [21].

Moreover, there are multiple solutions to develop OpenFlow controllers, for instance, NOX, OpenDaylight, and Ryu. Focusing on the latter, the Ryu SDN Framework is an open-source framework written in Python that allows the development of network management and control applications using a specific API. The OpenFlow versions supported by Ryu are 1.0, 1.2, 1.3,

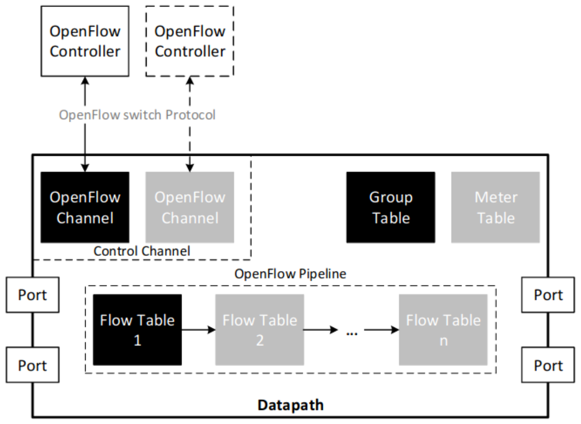


Figure 2.11: OpenFlow switch architecture (from [21]).

1.4, and 1.5. Additionally, templates and examples of controllers are provided as examples. Ryu SDN framework is available under an Apache 2.0 license.

CHAPTER 3

Methodology

This chapter presents the methodology used to elaborate the project that resulted in this dissertation and whose objective was to create and validate a framework to develop CTM systems.

This dissertation followed the work developed by Ferreira in [22], where the author studied possible architectures for developing automation systems. In this preliminary work, it was selected an edge computing-based architecture [23][24], an approach that brings the computing processes closer to the data sources, in this case, the devices. Since it follows the technical requirements and due to its scalability, Ferreira selected this architecture for this project. In short, the selected approach supports two types of data-flows:

- The devices generate data that is sent to the gateway that forwards it to the infrastructure, where data reaches its endpoint, stored in a database, and then is presented to the user via a GUI.
- The user sends commands, via GUI. These commands are stored in the database, processed, and sent to the gateway that forwards them to the end device.

One can take the system's main components from the data-flows. They are the devices, gateway, and infrastructure (database, server agents, and GUI). Next, the devices selected and frameworks used to implement each component are presented in section 3.2. Lastly, in section 3.3 it is discussed how the message prioritization feature was implemented and validated.

3.1 Communication Protocols and Devices

We selected the devices and thus communication protocols used in this project with the help of the company *Diferencial*. It must be noted that there were no concrete functional requirements set for the devices since it was not an objective to develop a specific solution but instead to design a generic framework and evaluate and validate the solution. So, it is advantageous to test different types of devices, communication protocols, and purposes to validate the system and understand its limitations.

Firstly, the communication protocols to implement in the gateway were KNX, Modbus, and MQTT. They were elected because they usually serve different purposes. The former is most commonly used in building automation applications, whereas Modbus is more used to establish communication between devices in an industrial environment. The latter, MQTT, was selected since it can suit multiple purposes and due to its popularity regarding IoT and IIoT applications.

Finally, after choosing the communication standards that the gateway must implement, we selected the devices. As for the protocols, we applied three different approaches. For the Modbus standard, a single commercial device was connected to the gateway. Second, multiple commercial devices were used regarding the KNX protocol, but the KNX bus interface was done via a single KNX device acting as a gateway. Finally, no commercial device was selected regarding MQTT since it was intended to study hardware and software tools to implement an MQTT device.

The MQTT device implementation process and the commercial devices used to validate the framework are presented in detail in Chapter 5, particularly in Section 5.2.

3.2 Framework for CTM Systems

3.2.1 Gateway

When reviewing existing frameworks to implement the gateway, we considered some technical specifications:

- Communication protocols supported and tools to develop custom solutions.
- Data processing features.
- Persistent storage.
- Connections with external databases or endpoints.
- Remote management.
- Scalability.
- User Interface framework.

A similar study was conducted by Ferreira in [22], where *Eclipse Kura*, *macchina.io*, *Node-Red* and *EdgeX Foundry* were compared to perform in a similar scenario. The former two were excluded because of their limited support of communication protocols and their lack of software tools to develop custom device drivers. In the end, EdgeX Foundry ended up being the chosen gateway due to the lack of persistent storage in Node-RED.

In the scope of this dissertation, we analyzed another alternative in detail, the *ThingsBoard IoT Gateway*. This solution offered similar features compared to EdgeX Foundry, with the particularity of providing tools to implement a simple user interface similar to the one supplied by Node-RED. At this point, the tools provided by EdgeX Foundry were considered limited.

The gateway choice started by excluding Eclipse Kura and macchina.io since they still had the same drawbacks described by Ferreira [22]. So, when comparing the remaining three candidates, the point of "scalability" played in favor of EdgeX Foundry. Furthermore, both ThingsBoard and Node-RED offer a flow-based programming solution to implement edge data processing, whereas EdgeX presents a traditional programming solution. However, despite the convenience provided by the flow-based programming approach, we considered that such a solution could compromise large-scale applications. To conclude, although the three candidates meet the required specifications to implement a gateway for a CTM system, EdgeX Foundry was the platform selected.

Once elected the gateway framework, we implemented a gateway based on EdgeX Foundry. The device profile and configuration files were developed and added to the MQTT, Modbus, and KNX devices. This process allows the gateway to know the commands, data types, interfaces, and other specifications of each device connected to it. Finally, using the EdgeX Foundry Device Services SDK, a custom device driver was developed to interact with the KNX gateway device. This implementation process is presented in detail in Chapter 5, particularly in Subsection 5.1.1.

3.2.2 Database and Server Agents

In [22], an in-depth study was conducted to select the database that best suited building automation systems. First, three types of DBMS were considered and based on popularity research, and the author selected the databases MySQL, MongoDB, and InfluxDB. Then, from [25] Rautmare and Bhalerao concluded that when comparing MySQL and MongoDB to store time-series data, the latter slightly outperforms the former. This advantage was mainly observed in insert actions, which are the most used ones when storing time-series data for IoT applications. Finally, in [26], a similar study was conducted, but comparing three NoSQL-based databases, since the authors consider them as the best approach for time-series storage in IoT applications. This study concluded that the performance of InfluxDB was superior, and thus, the most appropriate choice for the context at hand. So, in the work that preceded this one, InfluxDB was the DBMS selected to store the IoT data.

With the work previously developed in mind, we conducted research intending to analyze the solutions that best suited the IoT scenario in terms of performance. The conclusions reached were that the work developed in [26] was the most recent and valid one. So, having that in mind that InfluxDB, MongoDB, and Apache Cassandra would be suitable solutions for the context of this dissertation. The databases were studied not according to their performance since it is already known that InfluxDB outperforms the remaining, but to the additional tools and features they provided.

When reviewing the database's additional tools and features, we studied two in particular detail. Namely, server agents, which are tools to allow data (received from the gateway) to be preprocessed before it is stored in the database, and tools to connect to a GUI based on the data stored. All three candidates had in their ecosystem helpful and well-documented tools. For instance, all DBMSs provided server agent tools to act as a middleman between the data sources and the database. Additionally, all of them offered connectors to commonly used dashboards, such as Prometheus or Grafana. However, the design simplicity and the vast offer of plugins and connectors favored InfluxDB regarding the server agent.

Finally, InfluxDB ended up being the selected database to be the central element of the system's infrastructure. It is, however, essential to note that despite the differences, Apache Cassandra and MongoDB would also be reasonable solutions.

Regarding the implementation process, with the database selected, all the tools started to be installed and set up, then based on InfluxData Telegraf, a server agent was developed. When it was concluded, we established the communication between gateway and database. Lastly, we implemented another server agent, but this time to ensure the communication between the database (from the GUI) and the gateway. This implementation process is described in detail in Chapter 5, particularly in Subsection 5.1.2.

3.2.3 Graphical User Interface

The last main component to be selected was the GUI. We studied two different approaches, web-based and application-based solutions.

The former has a clear advantage over the latter when considering the simplicity of implementation since they provide drag-and-drop tools to quickly represent the data stored in a given database in different schemes. On the other hand, the disadvantage of web-based solutions is related to the interaction with the user. Although multiple alternatives are provided to present the data to the user, there are no tools to allow the user to input data in the system. However, typically these solutions enable the developers to implement their custom-built panels, with which the user can interact. On the other hand, application-based solutions have the advantage of flexibility. Usually, some templates are provided, but the developers have the freedom to implement their GUI as custom as they wish. These can be helpful scenarios with strict specific functional requirements. However, for most applications, the flexibility provided by application-based toolkits does not pay off the effort required to implement the GUI. Additionally, customizing application-based solutions could compromise scalability, whereas standardization associated with web-based solutions facilitates the scaling problem.

Several toolkits were considered, either regarding web-based or application-based solutions. Since the objective was to develop a framework and not a solution to meet specific needs, there was no point in prioritizing customization over standardization. So, we considered web-based approaches, analyzing frameworks like Grafana, freboard.io, and Chronograf. Grafana ended up being the selected one. This choice was mainly due to its offer of visualization solutions, integration with different data sources, visual appearance, and maturity. Given these four main elements, we considered that Grafana was the best solution to develop the system's GUI.

Finally, after electing Grafana as the framework for developing the system's GUI, it was time for implementation. As stated before, presenting the data stored in the database was a simple process since both frameworks provide connectors to ensure this integration. So, it was a matter of selecting the desired data and format to present it. However, Grafana does not provide default panels or plugins to allow the user to input data in the system. So, to allow such interaction, custom input panels were designed and implemented in HTML and JavaScript. With the input panels developed and the data graphically represented, the GUI, and thus the system, was completed. The GUI implementation process is described in detail in Chapter 5, particularly in Subsection 5.1.3.

3.3 Message Prioritization

With the framework for a CTM system developed and validated, our focus went towards preventing losing packets from the devices to the gateway, in particular, critical messages sent via IP-based protocols, such as TCP and UDP. The objective was to allow some message prioritization mechanism based on commercial solutions so that minimum bandwidth is ensured for the devices considered critical. This bandwidth guarantee must be observed even in cases where the network is stressed due to excessive traffic.

We applied SDN principles to add the desired feature to the framework. An OpenFlow switch and a Ryu controller were configured in two setups: traditional switch and switch with bandwidth guarantees for critical messages so that one can validate the system through comparison. We used two SIMATIC IOT2040 [27] gateways to act as MQTT devices since they are compliant with industrial environment requirements. It was an opportunity to study commercial solutions to serve as an IIoT gateway, for instance, for MQTT devices. Finally, we used four machines to generate traffic to simulate stress in the network. The implementation of message prioritization features is described in detail in Chapter 6.

CHAPTER 4

Architecture

This chapter presents in detail the architecture of the framework developed, its main components and the interfaces between them. A simplified version of the system developed is presented in Figure 4.1, from where three main elements stand out: (i) the Framework for CTM systems (Section 4.1); (ii) the devices used to validate the implementation (Section 4.2); and (iii) the network components that implement the message prioritization mechanism (Section 4.3).

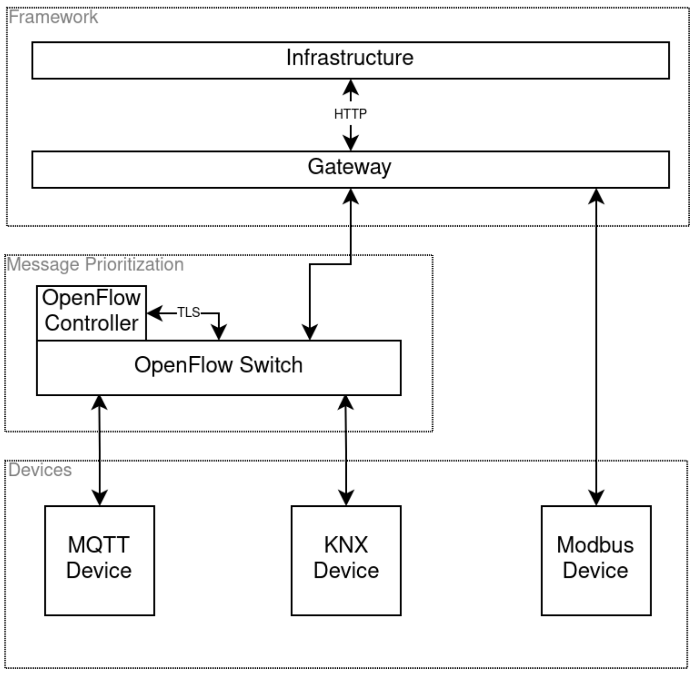


Figure 4.1: System Simplified Architecture.

4.1 Framework for CTM Systems

The framework developed has two core components, gateway and infrastructure, and it is graphically represented in Figure 4.2. The gateway is based on EdgeX Foundry and aimed to be the intermediary between the physical devices and the infrastructure.

On the other hand, the infrastructure is composed of three elements: the database, GUI, and server agents. The database (InfluxDB) and the GUI, developed in Grafana, are responsible for storing the data generated and interacting with the users, respectively. In addition, two server agents are implemented, one for allowing the communication from the gateway to the infrastructure, called South-to-North (SN) server agent, while a North-to-South (NS) server agent was developed to establish the communication from the infrastructure to the gateway. The former server agent is implemented based on Telegraf, whereas the latter is developed in Python 3.

In terms of hardware, all the framework components are implemented and executed in a Linux machine running Ubuntu OS 20.04. However, we developed the framework to make it possible to run the gateway, database, server agents, and GUI in different machines. This is due to the fact that the interface between the modules is performed using the HTTP protocol, more specifically, Representational State Transfer (REST) APIs.

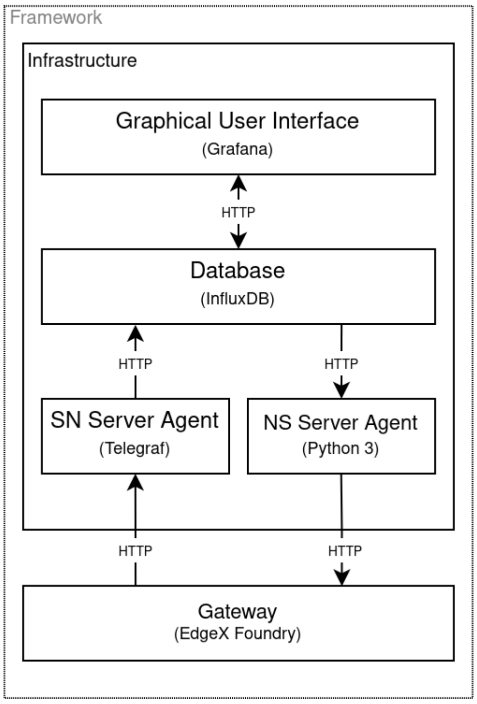


Figure 4.2: Framework Architecture.

4.2 Devices

Three different communication protocols are used to validate the correct operation of the framework, namely KNX, Modbus, and MQTT. Multiple devices, using these communication standards, are connected to the gateway, as shown in Figure 4.3.

The KNX devices used are listed in Table 4.1. These devices are commercial solutions to solve specific tasks regarding building automation. One of these devices is a DALI gateway that makes the interface between DALI lamps and the KNX bus. It is significant to notice that only one device (KNX IP BAOS 774 [28]) is connected to the framework via the EdgeX Foundry gateway. That device makes the interface between the devices connected to the KNX bus and the gateway implemented via TCP. For this reason, in this text, the term *KNX device* appears to refer to all the KNX devices since from the gateway's point-of-view, there is only one KNX device. Following the same principle, the DALI lamps and the respective DALI gateway are considered (for simplicity purposes) a single KNX device.

Moreover, regarding the Modbus protocol, only one device is used. The energy meter EM340 [35], by Carlo Gavazzi, is a commercial solution whose communication interface is done via Modbus/RTU. The device uses a serial communication to connect the device to the gateway.

Lastly, the MQTT device implemented is not a commercial solution but a custom one. It is custom developed using an ESP32 microcontroller. The device built simulates a simple ON/OFF controlled heating system. The device is connected to the same network where the machine running the gateway is connected, via Wi-Fi. Lastly, Eclipse Mosquitto [36] is used as the MQTT broker to enable the device and the gateway to communicate. This broker is executed in the same Linux machine that runs the framework.

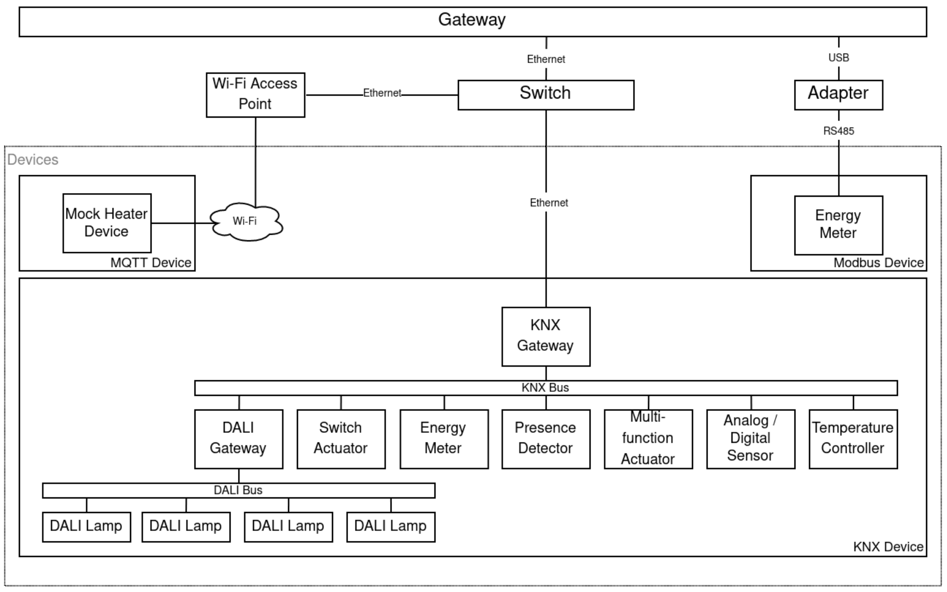


Figure 4.3: Devices Connected to the CTM Sytem.

Device	Reference	Manufaturer
KNX Gateway	KNX IP BAOS 774 [28]	Weinzierl
DALI Gateway	SCN-DALI 64.02 [29]	MDT technologies
Switch Actuator	AMI-0416.02 [30]	MDT technologies
Energy Meter	Kamstrup 382Jx8 [31]	Kamstrup
Presence Detector	Swiss Garde 300 D Presence KNX/KLR [32]	Züblin Elektro
Multi-function Actuator	ACTinBOX Classic - ZN1IO-AB46A [33]	Zennio
Analog/Digital Sensor	QUAD - ZN1IO-4IAD [33]	Zennio
Temperature Controller	SCN-RT4UP.01 [34]	MDT technologies

Table 4.1: List of KNX Devices Used.

4.3 Message Prioritization

The message prioritization mechanism is implemented employing an OpenFlow switch, and an OpenFlow controller, based on the Ryu SDN framework [37].

A different system, depicted in Figure 4.4 is set up specifically for testing the message prioritization mechanisms. As this feature only affects the communication between the devices and the gateway, a simplified version of the framework is used. In short, the system for testing the message prioritization feature has the following characteristics:

- Since the feature implemented is valid for all IP protocols, only the MQTT standard was tested.
- There was no need to run the system's infrastructure since that part of the system does not influence the message prioritization feature.
- The gateway was replaced with a Python program, running on *mock gateway*, that communicated with the mock devices (*device-A* and *device-B*) via the *MQTT broker*.
- Contrarily to the CTM system developed, the MQTT broker and the gateway were executed in different machines approach more realistic scenarios when performing stress tests.
- Three machines (*Traffic Generator (TG)-1*, *TG-2*, and *TG-3*), with the intent of generating load traffic in the network, were used during stress tests.

To validate the message prioritization mechanisms implemented, two setups that use the same hardware resources were considered, namely:

- **Setup 1 - Traditional switch** - In this setup, the system works as a traditional Media Access Control (MAC)-learning switch. The controller creates flow entries to match the switch ports to the Hosts' MAC addresses so that the switch can forward the packets as a common switch. If no flow entry matches the incoming packet, it is flooded to all ports.
- **Setup 2 - SDN switch** - Setup 2 is similar to the first one, but it has the message prioritization feature. It is configured to reserve 5 Mb/s of bandwidth to the MQTT packets, which are considered to contain critical information.

With these two setups, it was possible to compare different scenarios, in terms of network traffic, in a system based on a traditional switch (setup 1) and on a switch that prioritizes time-sensitive data (setup 2).

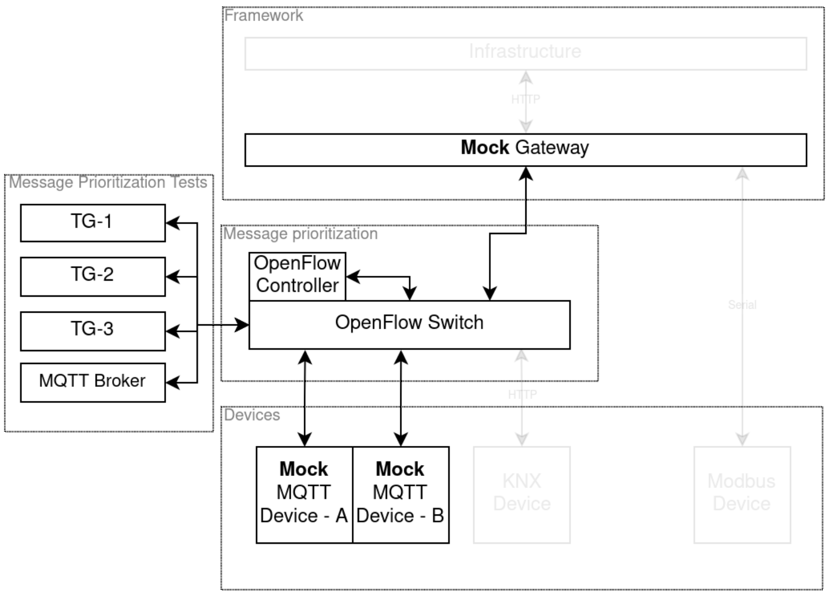
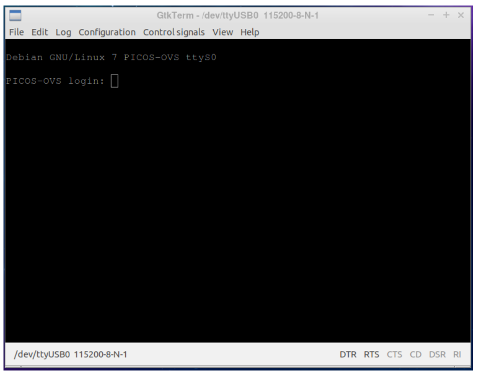


Figure 4.4: System to Test the Message Prioritization.



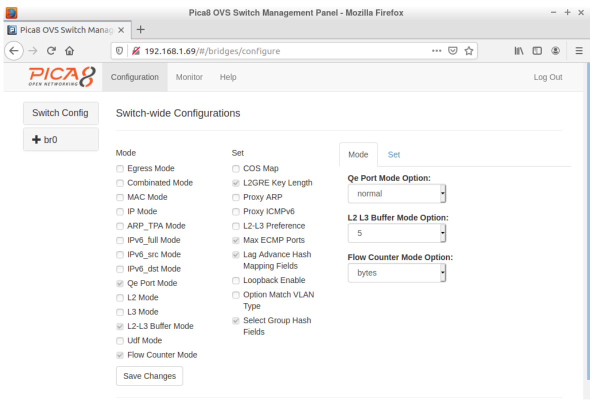
GtkTerm - /dev/ttyUSB0 115200-8-N-1

```
Debian GNU/Linux 7 PICOS-OVS ttyS0

PICOS-OVS login: 
```

/dev/ttyUSB0 115200-8-N-1 DTR RTS CTS CD DSR RI

(a) CLI



Pica8 OVS Switch Management Panel - Mozilla Firefox

192.168.1.69/#/bridges/configure

Switch-wide Configurations

Mode
- ☐ Egress Mode
- ☐ Combinated Mode
- ☐ MAC Mode
- ☐ IP Mode
- ☐ ARP_TPA Mode
- ☐ IPv6_full Mode
- ☐ IPv6_src Mode
- ☐ IPv6_dst Mode
- ✓ Qe Port Mode
- ☐ L2 Mode
- ☐ L3 Mode
- ✓ L2-L3 Buffer Mode
- ☐ Udf Mode
- ✓ Flow Counter Mode

Set
- ☐ COS Map
- ✓ L2GRE Key Length
- ☐ Proxy ARP
- ☐ Proxy ICMPv6
- ☐ L2-L3 Preference
- ✓ Max ECMP Ports
- ✓ Lag Advance Hash Mapping Fields
- ☐ Loopback Enable
- ☐ Option Match VLAN Type
- ✓ Select Group Hash Fields

Qe Port Mode Option: normal

L2 L3 Buffer Mode Option: 5

Flow Counter Mode Option: bytes

(b) Web-GUI

Figure 4.5: Switch Interfaces. (a) CLI and (b) Web-GUI.

At first, we used a commercial off-the-shelf switch to perform the tests, but that solution was discarded due to the hardware gap when comparing the traditional switch with the switch based on OpenFlow. Since this difference could jeopardize the analysis of the results, the latter approach was adopted.

The *OpenFlow switch* used in this project is the Edgecore AS4610-54T [38]. This switch is a commercial data center switch with 48 GE ports. It is compatible with some OS such as OcNOS and PicOS [39]. For this project, the PicOS from Pica8 Inc. is used. Among other features, this OS allows the users to configure the switch via a Command Line Interface (CLI) (Figure 4.5a) or via a Web-based GUI (Figure 4.5b). These configurations can be OpenFlow related, like creating flow entries or meters, or traditional ones, like adding a switch port to a Virtual Local Area Network (VLAN). The CLI can be accessed either via Secure Shell Protocol (SSH) or via Serial Port.

The *mock devices* used are based on two SIMATIC IOT2040 [27] devices. These devices, created by Siemens, were developed to be intelligent gateways for IIoT that work 24/7 in an industrial environment. In a real scenario, the gateways would be acquiring factory floor data, processing it (if necessary), and then sending it to a CTM system. The IOT2040 is compatible with Yocto Linux. Its operating system image and the SDK can be customized and created using Poky. For this project, a kernel was generated based on Yocto Linux 3.1 and the metadata provided by the manufacturer. Two main reasons lead to the choice of these devices instead of the already implemented MQTT device based on an ESP32 microcontroller: (i) It is a commercial solution to implement an MQTT device or gateway in an industrial environment; and (ii) it features two Ethernet interfaces, which means that it could connect directly to the switch via Ethernet, whereas the ESP32 microcontroller would require either a Wi-Fi access point or additional hardware to implement an Ethernet interface.

The *MQTT Broker* was implemented on a Raspberry Pi 4 running the Raspberry Pi OS 5.10.

Finally, for the remaining components, common personal computers were used. The *mock gateway* and *TG-1* run Ubuntu OS 20.04, whereas the *Controller*, *TG-2*, and *TG-3* run Lubuntu OS 19.04, since they did no meet the Ubuntu OS recommended minimum system requirements.

CHAPTER 5

Framework for CTM Systems

This chapter is dedicated to the implementation and validation of the framework developed. Section 5.1 presents how the framework, whose architecture has already been defined, was implemented. In particular, it is discussed how we configured the toolkits and frameworks selected to implement each component. Additionally, it is presented how the system functional blocks communicate with each other. Next, we added some devices to the system to validate the operation of the framework implemented. It is important to refer that the objective was not to produce a CTM system for a specific purpose but to test the framework. The process to include each device in the system, from defining it in the gateway to implementing the GUI to interact with that device, is described in Section 5.2. Moreover, it is described the process of developing a device driver for the gateway (for the KNX device) and the implementation of the MQTT device's firmware.

5.1 Implementation

This section focuses on the implementation of the framework's components, in particular the gateway (Subsection 5.1.1), database and server agents (Subsection 5.1.2), and GUI (Subsection 5.1.3). We used the virtual device provided by EdgeX Foundry to validate primarily the correct operation of each component.

5.1.1 Gateway

The framework EdgeX Foundry was the selected one to implement the gateway. Since there was no need to change or add functionalities to the source, we executed it in user mode. Therefore, it was required to install Docker Engine [40] and Docker Compose [41] (and the respective dependencies) as specified in the Docker documentation, to run the framework in user mode.

In the scope of this dissertation, the gateway was based on the release 1.2 (Geneva) of EdgeX Foundry. There is a GitHub repository [4] that provides YAML [42] docker-compose files to run the gateway. Several files are provided for the Geneva release, varying: (i) the architecture of the machine (x86 of ARM); (ii) the database (Redis or MongoDB); and (iii) security features. For this project, we selected: (i) the version for x86-based computers, according to the machine the system is running on; (ii) Redis database as the persistent storage unit, since MongoDB will be deprecated in future releases; and (iii) without security features, since it was not the objective of this project to study the security services provided by the gateway.

Next, by analyzing the docker-compose file downloaded, it is possible to understand that by default, it defines a Docker network, eleven Docker containers, and the order they should be started. Nine of the docker containers correspond to the main microservices that compose the gateway, whereas the last two correspond to device services, namely REST and Virtual. The former, REST device service, allows third-party applications to push data into the gateway using a pre-defined REST API. On the other hand, the Virtual device service consists of four simulated devices that generate random data. The last device is handy to test the gateway's operation since it pushes data to the gateway and receives commands from the gateway (e.g., to enable or disable randomization). The Docker network is used to allow communication between the different microservices. Finally, the docker-compose file contains the required information to run each microservice successfully, including network specifications, host and container names, environment variables, dependencies, Docker image, and other definitions. The image field contains the path to a Docker Hub repository that contains the image. When the system is started for the first time, using the docker-compose command and the selected docker-compose file, all the microservices are downloaded from their respective Docker Hub repositories.

After ensuring that the system was running with the virtual device services, two additional microservices were added to the docker-compose file, and consequently, the gateway. The first microservice added was a simple user interface used for developers to monitor and manage the instances of the system. One can find the Docker image of this user interface in [43]. The second component added to the gateway was an application service based on the already implemented configurable application service. Figure 5.1 represents the microservices that compose the gateway.

Until this moment, every image was instantiated with its default configurations. However, even using the user approach to run the gateway, it is possible to define custom settings to alter how the containers behave. For that, it is necessary to write configuration files, using TOML [44], mount them using volumes, and change the microservice's entry point to use the custom configuration file.

The configuration files allow developers to change different settings, for instance, registration, database, and service settings. However, in the scope of this project, we focus on the *Writable* and *Binding* section of the configurable application services' configuration file. The former includes a subsection dedicated to specifying the function pipeline used by the application services. In contrast, the latter is used to define the trigger that precedes the execution of that pipeline.

So, using the features provided by the configurable application service, it was created our

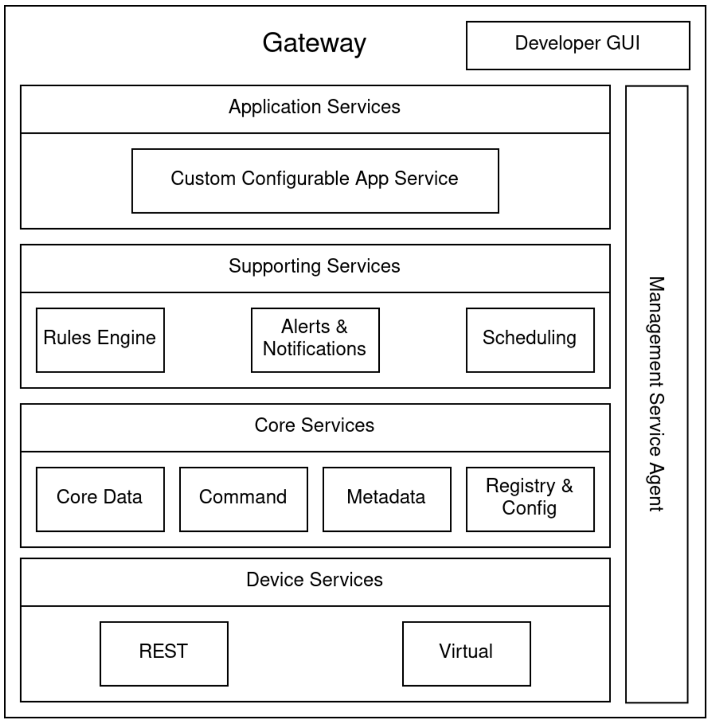


Figure 5.1: Gateway Microservices.

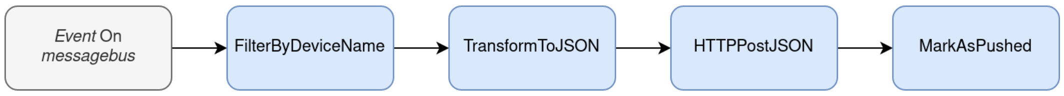


Figure 5.2: Application Service Function Pipeline.

custom application service whose function pipeline is presented in Figure 5.2. It is possible to observe that the pipeline is triggered whenever a message is published to the topic "events" from the gateway message bus. This is the topic used by the core data microservice to publish data received by the devices connected to the gateway. So, in short, the application service pipeline is triggered when the device services collect data from the devices. Apart from the trigger function, four standard functions compose the pipeline: *FilterByDeviceName*, *TransformToJSON*, *HTTPPostJSON* and *MarkAsPushed*. So this pipeline starts by filtering the devices whose data should be exported, then the data is transformed to JSON format, and finally, it is posted to an HTTP endpoint. The list of devices to filter and the HTTP endpoint are specified in the service configuration file. Lastly, the *MarkAsPushed* function signals the gateway that the information has already been exported, meaning that the core data microservice can delete its entry from the persistent storage unit.

In Listing 1 it is possible to analyze an example of a JSON message exported by the gateway. First, it can be noted that the application services can send more than one reading at one time and that two "origin" fields are added. These fields contained Unix timestamps (in nanoseconds) of the moments when the exported message was prepared, and the data was collected from the device.

To summarize, at this point, the gateway implemented was ready to receive commands to

```
{
    "id" : "ea1308ea-b553-4242-923d-dee82ec466df",
    "device" : "Random-Integer-Device",
    "origin" : 1604659537541997035,
    "readings" : [ {
        "id" : "d786fcde-3cfa-4843-8ba7-b3db6ba804a3",
        "origin" : 1604659537379490905,
        "device" : "Random-Integer-Device",
        "name" : "Int8",
        "value" : "41",
        "valueType" : "Int8"
      } ]
}
```

Listing 1: Example of a JSON Message Exported by the Application Services.

actuate on the devices and to send the data gathered by the devices to an HTTP endpoint. The former feature was done by the provided REST API defined in the Core Command Service. In contrast, the latter was accomplished by the developed application service.

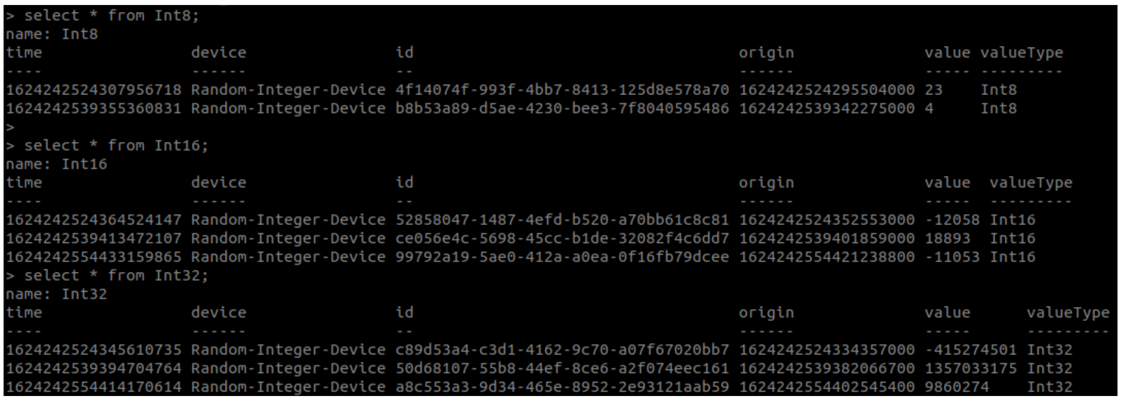
5.1.2 Database and Server Agents

With the gateway implemented, it was time to implement the infrastructure. Its main component is a database, but it also requires server agents and a GUI. The latter is discussed in Subsection 5.1.3 and aims to allow users to establish communication with the gateway.

Firstly, InfluxDB was installed from its source code, alongside its dependencies. The instruction for this installation process can be found in InfluxDB documentation [7]. Then the database structure was designed; InfluxDB uses measurements to separate the data inside the database. So we decided that the information should be divided into measurements according to their device resource name, to facilitate the analysis of multiple devices that gather the same physical quantity. Considering Listing 1, the device resource name is specified in the field "name". In a real-life scenario, such as when using a power meter device, which can be called for example *PowerMeter-1*. The field "device" of the exported JSON message would contain "PowerMeter-1", whereas the "name" field would then contain device resources' names like power, voltage, or current. So, the data would be stored in three different measurements: power, voltage, and current. The structure described can be seen in Figure 5.3 for the example of the virtual device.

InfluxDB provides a REST API to interact with the database. In particular, it is possible to push data into the database via an HTTP post, using the InfluxDB line protocol. The messages that are exported from the gateway use the HTTP protocol, but its content is in JSON format. To solve this issue, Telegraf was used as a server agent to guarantee communication from South to North, also called the SN server agent.

Telegraf has a plugin-based architecture, and a typical Telegraf application is composed of three types of plugins: inputs, processors and aggregators, and outputs. It is necessary to add the desired plugins to a TOML configuration file to run the Telegraf server agent. Those plugins act as function pipelines (similar to the gateway's application services). Figure 5.4 presents the pipeline developed for the Telegaf server agent. It is composed of a single input



```
> select * from Int8;
name: Int8
time                device               id                                   origin              value valueType
----                ------               --                                   ------              ----- ---------
1624242524307956718 Random-Integer-Device 4f14074f-993f-4bb7-8413-125d8e578a70 1624242524295504000 23    Int8
1624242539355360831 Random-Integer-Device b8b53a89-d5ae-4230-bee3-7f8040595486 1624242539342275000 4     Int8
>
> select * from Int16;
name: Int16
time                device               id                                   origin              value  valueType
----                ------               --                                   ------              -----  ---------
1624242524364524147 Random-Integer-Device 52858047-1487-4efd-b520-a70bb61c8c81 1624242524352553000 -12058 Int16
1624242539413472107 Random-Integer-Device ce056e4c-5698-45cc-b1de-32082f4c6dd7 1624242539401859000 18893  Int16
1624242554433159865 Random-Integer-Device 99792a19-5ae0-412a-a0ea-0f16fb79dcee 1624242554421238800 -11053 Int16
> select * from Int32;
name: Int32
time                device               id                                   origin              value      valueType
----                ------               --                                   ------              -----      ---------
1624242524345610735 Random-Integer-Device c89d53a4-c3d1-4162-9c70-a07f67020bb7 1624242524334357000 -415274501 Int32
1624242539394704764 Random-Integer-Device 50d68107-55b8-44ef-8ce6-a2f074eec161 1624242539382066700 1357033175 Int32
1624242554414170614 Random-Integer-Device a8c553a3-9d34-465e-8952-2e93121aab59 1624242554402545400 9860274    Int32
```

Figure 5.3: Database Structure (Example using Virtual Device).

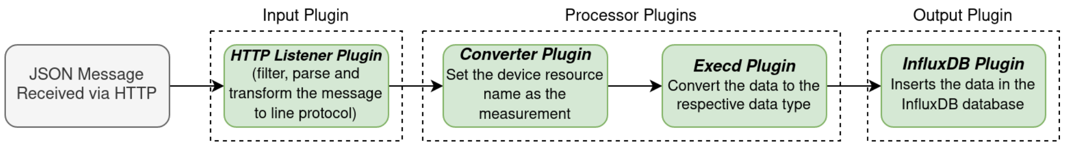

Input Plugin
Processor Plugins
Output Plugin
JSON Message Received via HTTP
HTTP Listener Plugin (filter, parse and transform the message to line protocol)
Converter Plugin Set the device resource name as the measurement
Execd Plugin Convert the data to the respective data type
InfluxDB Plugin Inserts the data in the InfluxDB database

Figure 5.4: Telegraf SN Server Agent Pipeline.

```
Int8 value=41i,valueType="Int8",id="d786fcde-3cfa-4843-8ba7-b3db6ba804a3",origin=160465⌋
↪  9537379490905,device="Random-Integer-Device"
↪  1604659538571697235
```

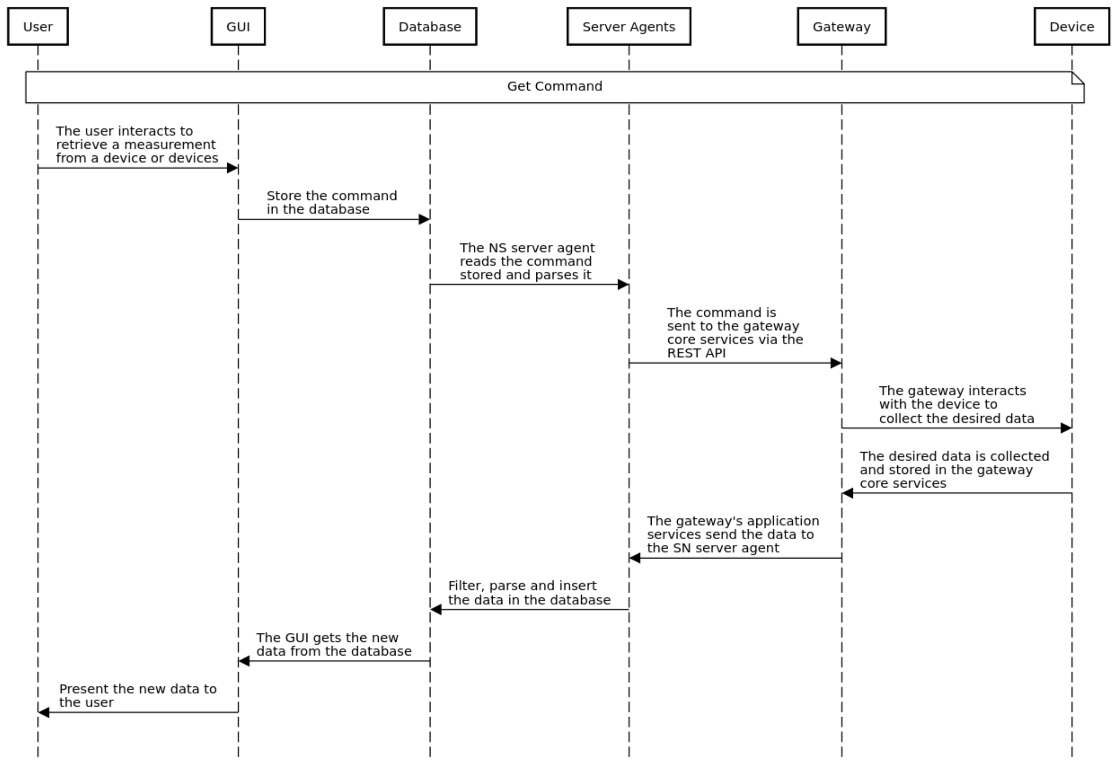
Listing 2: Example of a Line Protocol Message To Insert Data Into the Database.

plugin that is responsible for reading and filtering the JSON data sent by the gateway. This plugin also automatically transforms the information received into the InfluxDB line protocol. Next, the data is transferred to two processor plugins. The first sets the measurement to the device resource name and the second converts the data from string type to the data type specified in the JSON message. Finally, to end the functions pipeline, the processed data is pushed into the database.

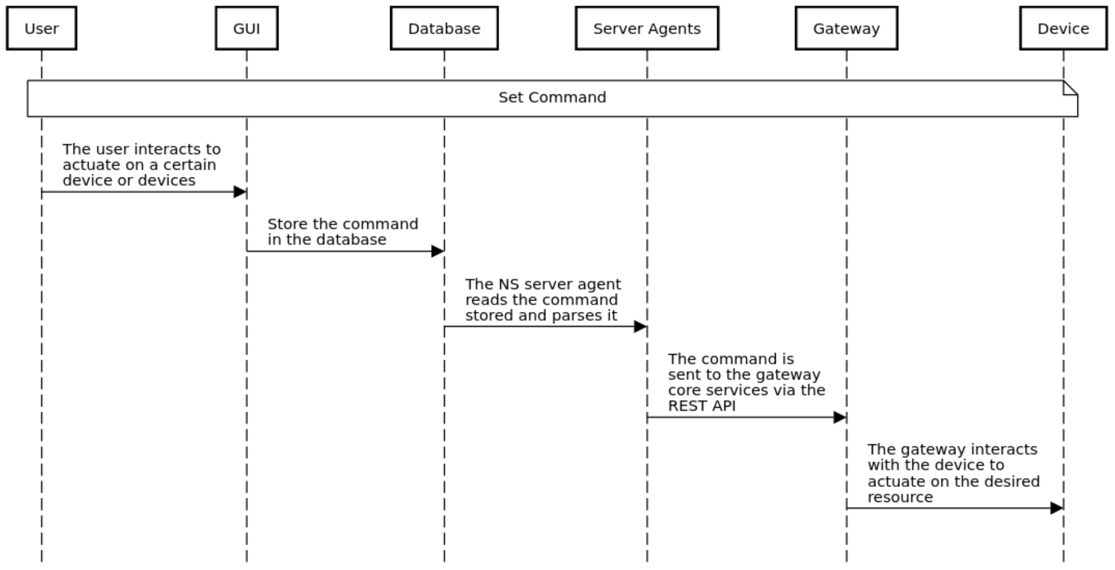
Listing 2 presents, as an example, the message from Listing 1 after being transformed and processed by the server agent. The server agent ensures that this message is sent to the database HTTP endpoint to store the data that it contains.

With this implementation, it is guaranteed that the data from the gateway application services is successfully written in the database (with the correct data type). Next, we implemented the NS server agent. Contrarily to the SN server agents, its objective is to send commands to the gateway (that are then sent to the devices). These commands will come from the GUI, and the interface between the GUI and the gateway is done by the NS server agent.

The system was designed to work as follows: (i) the user interacts with the GUI; (ii) the command from the user is processed and stored in the database; (iii) the NS server agents read that command, process it and send it to the gateway; (iv) the gateway interacts with the devices. This procedure is represented in Figure 5.5. It is also possible to observe that there are two types of possible interactions from the users: *GET* and *PUT*.


User
GUI
Database
Server Agents
Gateway
Device
Get Command
The user interacts to retrieve a measurement from a device or devices
Store the command in the database
The NS server agent reads the command stored and parses it
The command is sent to the gateway core services via the REST API
The gateway interacts with the device to collect the desired data
The desired data is collected and stored in the gateway core services
The gateway's application services send the data to the SN server agent
Filter, parse and insert the data in the database
The GUI gets the new data from the database
Present the new data to the user

(a) Get Command.


User
GUI
Database
Server Agents
Gateway
Device
Set Command
The user interacts to actuate on a certain device or devices
Store the command in the database
The NS server agent reads the command stored and parses it
The command is sent to the gateway core services via the REST API
The gateway interacts with the device to actuate on the desired resource

(b) Put Command.

Figure 5.5: Sequence to Perform (a) Get and (b) Put Commands, from end-to-end.

```
# Get command
user_input_get device="<my_device>",command="<my_command>"

# Put Command
user_input_put
↪  device="<my_device>",command="<my_command>",body="<my_device_resource>:<my_device_resource_value>"
```

Listing 3: Line Protocol Message Structure To Insert User Input.

Subsection 5.1.3 addresses the part of acquiring the commands from the user and storing them in the database. Here we discuss the implementation of the NS server agent that sends commands to the gateway based on the instructions stored in the database. So, it is necessary to define how the data is deposited in the database (by the GUI) before developing the NS server agent. Listing 3 presents the InfluxDB line protocol message structure that enables the GUI to save user commands in the database.

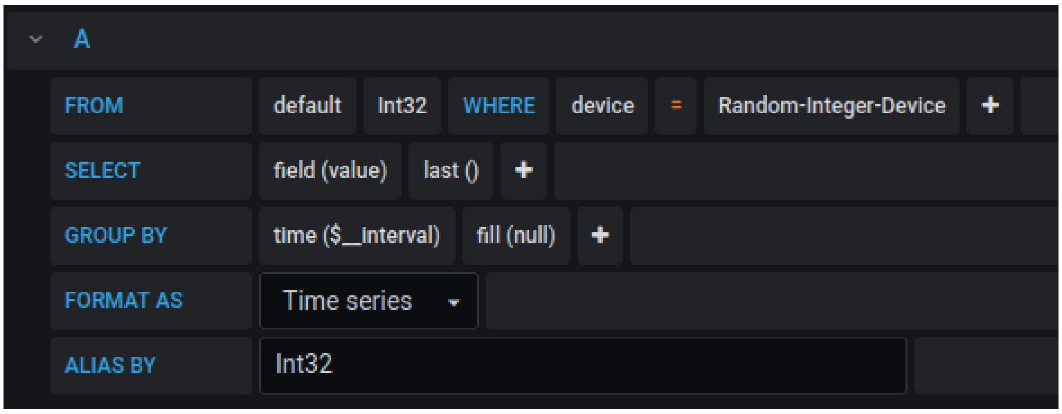
InfluxDB provides a subscription feature [45] that consists of forwarding specific data written in the database to an external endpoint (for example, an HTTP endpoint). This feature was used by the NS server agent, which consists of a web-server, implemented using Python 3, whose operation can be described as follows:

1. Create a subscription to InfluxDB, specifying the subscription name (sub0), database name (database_influx) and the NS server Uniform Resource Locator (URL).
2. Whenever a post request occurs, verify if the body of the request contains either "user_input_get" or "user_input_put".
3. If it does, it means that the user wants to send a command to the gateway. So, the received data (that is in InfluxDB line format) is parsed.
4. Depending on the command type, a put or a get request is prepared based on the device and command parsed.
5. In case of a put request, the body field is once again parsed to transform its "key:value" structure into JSON.
6. Finally, the request is sent to the gateway via its REST API, in particular to *http://edgex-core-command:48082/api/v1/device/name/<my_device>/command/<my_command>* and whose body field is either the message obtained from point 5 or empty, depending on the request type. After the HTTP message is sent, the process repeats from point 1.

One can observe that the response from the HTTP get request is not used because once a gateway command is sent, the data generated by the device triggers the application services pipeline. So, instead of expecting the HTTP response itself, the user waits for the application services to push that same data into the database, that is consequently represented in the GUI.

Finally, the NS server agent was "dockerized" so that all the infrastructure components could be added to the system docker-compose file. The database, SN server agent, and NS server agent were then added to the compose file. Finally, volumes were added to the first two containers to define specific settings in InfluxDB and pass the configuration file (including the plugin pipeline) in the other one.

To conclude, the backend part of the infrastructure was completed. With this, the communication between the gateway and the database was established. Then, two server agents were used to make the interface between these two components. In the end, we added the elements used and developed to the docker-compose file that contained the gateway microservices.



```
A
FROM        default  Int32  WHERE  device  =  Random-Integer-Device  +
SELECT      field (value)  last ()  +
GROUP BY    time ($__interval)  fill (null)  +
FORMAT AS   Time series
ALIAS BY    Int32
```

Figure 5.6: Grafana Query tool.

5.1.3 Graphical User Interface

The last component implemented was the GUI, which was developed using Grafana. At first, Grafana was installed from the source code, following the instructions provided by the documentation [11].

After setting up the platform, it was necessary to specify a data source. In the case of this platform, the source was the database (InfluxDB). This process can be done via the user interface provided by Grafana, and it requires specifying: the DBMS, query language, database URL and other communication settings, authentication configurations, and other details. After configuring the data source, Grafana provides a mechanism to test if it can connect to the database with the specifications.

Once the communication between the data source and GUI is verified, a dashboard was created (for testing purposes). This dashboard aimed to represent the data retrieved from the virtual devices graphically. So we added panels to the dashboard and selected the desired visualizations that can be, for example, graphs, gauges, bar gauges, text, heatmaps, or pie charts. Of course, alongside the visualization mechanism, it is necessary to get the data to represent. For that, Grafana provides a graphical query tool customized for the data source selected (in our case, InfluxDB) to choose the data. An example of that tool can be observed in Figure 5.6.

Notice that "Int32" corresponds to the measurement name that refers to a device resource and not to the data type (that, in this case, are the same). In the figure it is represented a typical query. It is only necessary to change "Int32" and "Random-Integer-Device" to their respective measurement and the device for other scenarios. The device field is particularly necessary when it is intended to filter some device or devices. Take, for example, a situation where multiple devices collect a "temperature" variable. If they share the same device resource name then they will be stored under the same measurement. Thus, filtering is required if the developers only want to show data from specific devices.

After querying the data and selecting the most appropriate visualization tools, the communication from the South to the North part of the system was accomplished. Finally, it was guaranteed that the data gathered (although from virtual devices) was presented to the

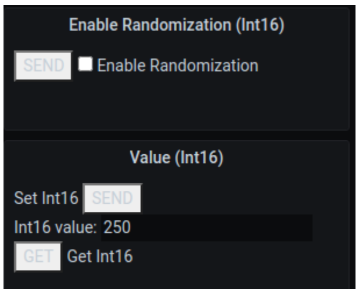


Figure 5.7: Grafana Custom Input Panels.

user via a GUI. It was then time to close the loop by allowing the user to interact with the devices in that same GUI.

The tools to get user inputs provided by Grafana are quite limited. For this reason, *text* panels were used to implement the elements that allow users to interact with the devices. The *text visualization tool* allows the developers to create custom panels using HyperText Markup Language (HTML) and JavaScript. We created three different templates:

- Get command - Allows users to retrieve data from the devices asynchronously.
- Put command with binary input - Allows users to send binary commands to the devices (e.g., turning on or off a heater).
- Put command with text input - Allows users to send text-based commands to the devices (e.g., defining the temperature setpoint of a heater). Users may send integer, float, and binary data types using this template.

Figure 5.7 shows a scenario where the three templates described above are used. The Random-Integer-Device, a virtual device that randomly generates numbers with different data types, has a command (and a device resource) called "Int16". Users may use a get command to collect a new random number. Additionally, two put commands may be used, one to disable randomization and the other to impose a specific value.

To use the template, a developer needs to specify the device and command names. In the case of *PUT* commands, it is also necessary to specify the device resource name that it is intended to change, in the field "body". As defined in Section 5.1.2, the database name used was "database_influx" and the measurements used to store user commands were "user_intput_get" and "user_intput_put". Both database and measurement names should not be changed from the template.

Focusing on the example of the "Int16" command, after configuring the templates correctly, a user should get and set the integer value. When this happens, i.e. when the user presses the "GET" or "SEND" buttons, an HTTP post is sent to the URL *https://influxdb:8086/write?db=database_influx* with the content presented in Table 5.1, respectively. Notice that in the body field of the *PUT* command, the device resource name and value are separated by a colon.

In the end, and following the same principle applied before, we added Grafana to the gateway docker-compose file. The image of Grafana is available in Docker Hub [46]. A volume was mounted to the container with the required provisioning specifications regarding

Action	HTTP Post Command Body (using influxDB line protocol)
GET	user_input_get device="Random-Integer-Device",command="Int16"
PUT	user_input_put device="Random-Integer-Device",command="Int16", body="Int16:250"

Table 5.1: Examples of an HTTP Post With User Input Data.

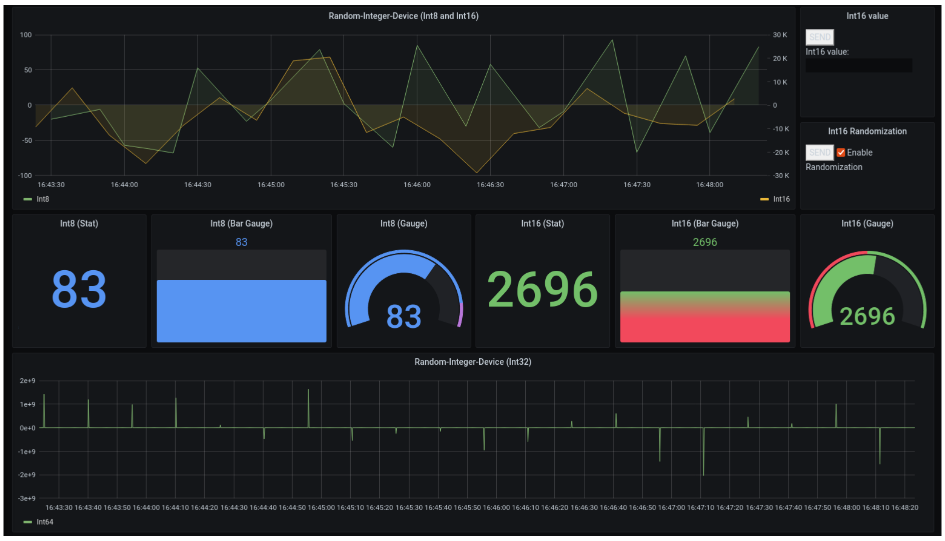


Figure 5.8: Grafana Random-Integer-Device Dashboard.

the data source and the dashboards. With this change, there is no need to configure the connection to the data source since that procedure is done automatically. Moreover, the dashboards developed must be saved (into JSON files) and stored in the path specified in the docker-compose file.

To conclude, the GUI was the last element of the framework implemented. It allows users to graphically analyze the data collected from the virtual devices (stored in the database) and control the devices using the custom panels developed for this purpose. In terms of development, we created three HTML templates to allow developers to add their custom buttons to control the devices. Finally, the GUI was added to the docker-compose file as a microservice to facilitate the implementation of the CTM system. Figure 5.8 presents the dashboard created to monitor and control the virtual devices used for testing purposes.

5.2 Validation

This section focus on the validation of the framework for CTM systems developed. Apart from the REST and virtual device services provided by EdgeX Foundry, three device services (also referred to as device drivers) were added to the gateway. Each one of them was responsible for making the interface with a different communication protocol.

EdgeX Foundry already provides the microservices to connect to MQTT and Modbus (TCP/IP and RTU) devices. However, at the moment of this writing, the KNX protocol was not supported by the gateway. So, the device service SDK was used to develop a device driver for the KNX device. In the three cases, configuration files were created to modify specific device driver settings and mounted as volumes on the docker container of each microservice. This procedure was similar to what was done before, for example, in the configurable application service. The device services' configuration files include, in particular, a path to the device service profiles. These profiles are YAML files that contain information about the physical device connected to the gateway. The device services profiles include:

- General considerations about the device (such as name, manufacturer, model, description).
- *Device Resources* - a list of the device resources, each specifying the data type, units, default values, if it is a resource to read from, write to or both, and other features. This list allows the gateway to implement a digital twin of the physical device, meaning that when a device resource is modified, it affects the physical device and vice-versa.
- *Device Commands* - a list that contains the command names, the device resources used by each command, the operations (get or put) on the device resources, and additional parameters.
- *Core Commands* - a list of core commands that define settings regarding the interface with the devices from outside the gateway. In short, they specify the REST API that allows third-party agents to interact with the physical devices.

5.2.1 Modbus Device

The Modbus device used to validate the system was the EM340 [35] from Carlo Gavazzi. This device is a three-phase energy meter with tariff management and an LCD touchscreen display. It can measure energy, power, voltage, and current values, of three-phase and single-phase loads. The interface with this device can be done via the display or RS485 port. In both, it is possible to retrieve the data collected or manage the device settings.

The load selected to test the device was an electric heater with two operational modes, one that consumes approximately 1 kW and the other that consumes approximately 2 kW. The energy meter was placed in between the load and the mains power. We used an RS485/USB adapter to connect the meter to the gateway. This communication was done using the Modbus communication protocol. Figure 5.9 summarizes the interfaces between the load, mains power, meter, and gateway.

After configuring the physical device, the Modbus device driver was added to the system. EdgeX Foundry provides the image of this microservice in Docker Hub [47]. So, the microservice was added to the Docker compose-file of the framework. A directory containing the configuration file and device profile was mounted to the microservice container.

The configuration file included configurations regarding the device, service settings, and the device profile name. Considering the device settings, and in particular the communication part, it was specified the serial port address, baud rate, data bits, stop bits, parity, and unit ID. The last is helpful in situations where multiple Modbus devices are connected to the gateway. Note that the gateway acts as the Modbus master device. The configuration file also included the definition of auto-events. This allows the gateway to gather information with a defined periodicity automatically. To generate these, it is necessary to specify a device

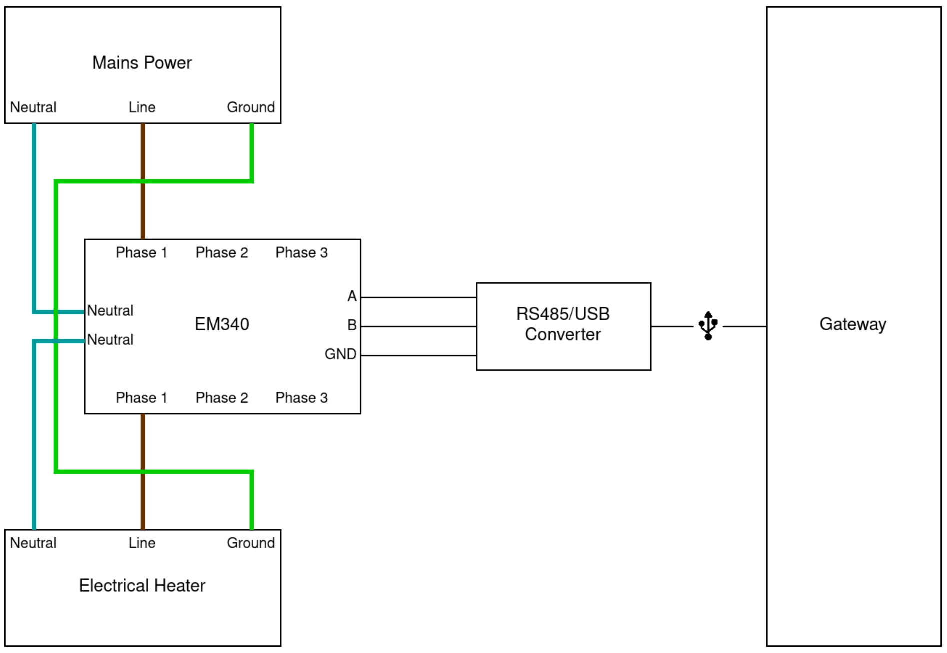


Figure 5.9: Modbus Device Electrical Diagram.

resource and the respective sampling period. In our case, the sampling period of all device resources was 20 seconds.

The device resources, defined in the device profile, were the voltage and current measured in the three phases and the total energy consumption. These were only some of the measurements provided by the meter. Notice that, as it was used a single-phase load, and it was connected to phase 1, only *VoltagePhase1*, *CurrentPhase1*, and *TotalEnergy* device resources provided relevant measurements. The device profile in the device resources list included the address of the holding registers containing the desired data. These addresses and other relevant information regarding the communication with the device are specified by the manufacturer [48].

At this point, the Modbus device service was completely configured to interact with the EM340 device. So, a dashboard was implemented, as shown in Figure 5.10, after validating that the desired data was being inserted in the database. This dashboard uses gauges, bar gauges and a chart to represent the measurements obtained from the device. In addition, for each resource, we added a button to allow the user to get the respective measurement asynchronously.

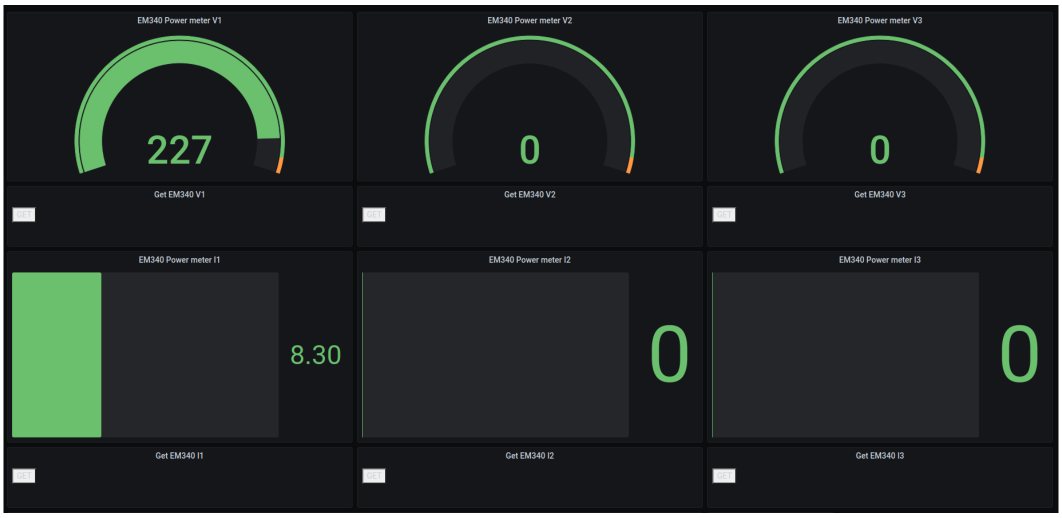


Figure 5.10: Part of the Modbus Device Dashboard.

5.2.2 MQTT Device

To validate the system using the MQTT protocol, it was necessary first to create an MQTT device and then add it to the gateway (similarly to what was done in the Modbus device).

The device, whose circuit is represented in Figure 5.11 was developed based on a ESP32-DevKitC [49], from Espressif. The system implemented simulates a heating system with an ON/OFF controller. The microcontroller reads the temperature from a TC74 [50] sensor and, based on that, actuates on a resistor that produces heat when the current temperature value is below the setpoint. A transistor (IRF840 [51] was used as the power interface between the resistor and the ESP32 board. Additionally, it has two Light-Emitting Diodes (LEDs) and one pushbutton. One LED indicates if the system is actuating on the resistor, and the other indicates whether the heater is turned on or off. The pushbutton was added to allow the user to indicate an emergency. In a real scenario, this button could, for example, halt the heater or even turn off the power box of a building, house, or factory.

The implemented MQTT device has four device resources, namely, *temperature*, *heater setpoint*, *heater state*, and *alarm*. The first resource, temperature, can only be read since it is acquired from the MQTT device. The heater setpoint is the resource that allows the user to select its desired temperature and can be either read from or written to the device. The same happens to the heater state resource, but in this case, it allows the user the turn on or off the heater. This means that the actuation on the resistor requires both the heater to be turned on and the temperature value to be inferior to the heater setpoint value.

The code to implement the heater system was developed in C. It was used a framework ESP-IDF [52] and the MQTT client part of the system was based on the code provided by the same framework and available on GitHub [53]. After establishing MQTT communication with a simple third-party application, a device driver was created to communicate with the TC74 temperature sensor, whose interface was done via I^{2}C. Next, the operation of the temperature sensor was validated, and the LEDs, pushbutton, and power resistor with the

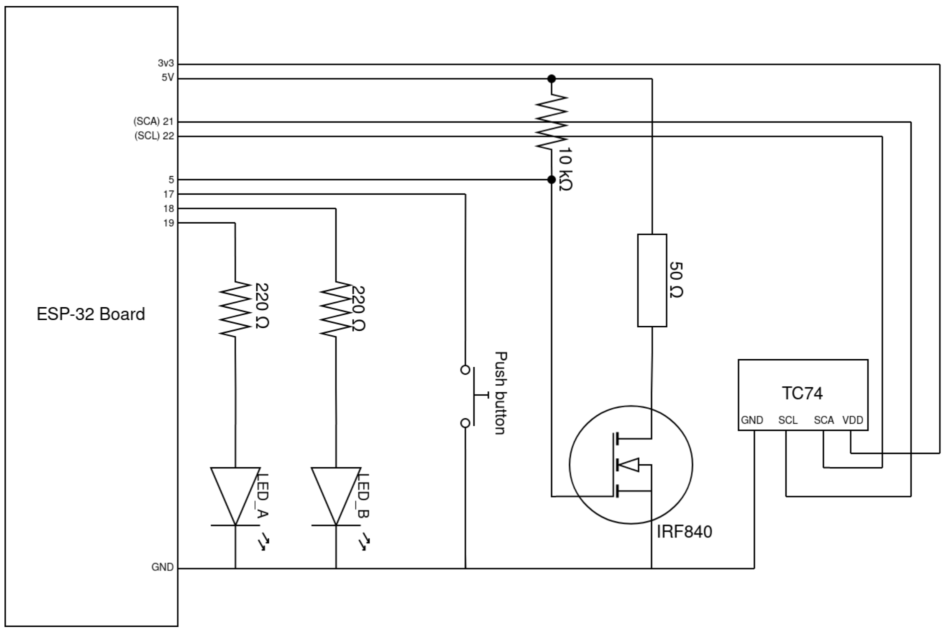


Figure 5.11: MQTT Device Circuit Diagram.

respective transistor were added to the system and then validated. So, at this point, all the required hardware components were set up, the firmware to interact with those components was implemented, and the software to establish MQTT communication was validated, so it was time to join all the components together.

From a high-level point of view, the MQTT device performs the three tasks that follow:

- **Temperature task** - the system periodically reads the temperature from the sensor and actuates on the resistor based on that temperature, the heater setpoint, and the heater state. The last two are defined by the user.
- **Alarm task** - responsible for checking when the alarm button is pushed. If the button is pressed, an MQTT message is sent to the gateway.
- **Communication task** - when an MQTT message using the JSON format is received, it is first parsed. If it is a valid message, i.e., it is a command from the gateway, that command is executed and the MQTT response is sent. The command can be either a get or a put. In the second case, the respective variables are updated and the actuation is executed in the hardware accordingly.

Of course, before starting these tasks, there is a setup process where the whole system is prepared to execute its function. There, the ESP32 board GPIOs are configured. In particular, the pin that connects to the transistor via a pull-up resistor is configured in open-drain mode. This is because its Gate-Source voltage threshold is superior to 3.3 V (the microcontroller operating voltage). So, we used a pull-up resistor to 5 V, and the pin was configured in open-drain mode. This mode protects the microcontroller that, in standard configuration, is only tolerant to voltages up to 3.3 V. The pin that connects to the pushbutton is configured in pull-up mode to avoid using an external resistor for the same effect. The LEDs are configured as standard outputs. In the setup process, the TC74 is also initialized, and the MQTT topic where the gateway publishes its messages is subscribed.

```
# Get and Put commands (published in CommandTopic)

{"cmd":"temperature", "method":"get", "uuid":"60d28d41b8dd790001fc6dcc"}
{"cmd":"setpoint", "setpoint":50, "method":"set", "uuid":"60d28d4fb8dd790001fc6dd3"}

# Responses to the get and put commands (published in ResponseTopic)

{"cmd":"temperature", "method":"get", "uuid":"60d28d41b8dd790001fc6dcc",
↪  "name":"MQTT-Demo-Device", "temperature":25}
{"cmd":"setpoint", "method":"set", "uuid":"60d28d4fb8dd790001fc6dd3",
↪  "name":"MQTT-Demo-Device", "setpoint":50}

# Asynchronous data sent by the MQTT device (published in DataTopic)

{"cmd":"alarm", "method":"get", "name":"MQTT-Demo-Device", "alarm":true}
```

Listing 4: Examples of Packets Exchanged by the MQTT device and the Gateway.

Regarding the communication with the gateway, three topics are used: *CommandTopic*; *ResponseTopic*; and *DataTopic*. The gateway publishes its commands (get or put) in the former topic and expects the response from the MQTT device in the second one. The response from the device is similar for both put and get commands; it adds a "device" field to the message, containing its name, and in the case of the get command, the device sends the value of the desired device resource. Lastly, the *DataTopic* command allows the MQTT device to send data to the gateway asynchronously. This last feature is used by the alarm button as an alternative to polling, to reduce bandwidth usage. Listing 4 presents examples of the three types of messages presented above.

Similar to what was done in the case of the Modbus device, described in Subsection 5.2.1, the device service was based on an image provided by EdgeX Foundry in Docker Hub [54]. It was mounted on the microservice's container a configuration file that contained, in particular, configurations regarding the MQTT protocol (address of the broker, QoS, topics, credentials, among others). To support MQTT communication, we added a broker to the system's docker-compose file. The broker used was the Eclipse Mosquitto, whose image can also be found in Docker Hub [55]. As stated before, the alarm device resource was not periodically read by the system, so an auto-event was configured to execute every minute. This feature works as a "heart bit" to check if this functionality is working. An auto-event every 20 seconds is triggered to acquire the remaining device resources.

Finally, a GUI was implemented to allow the user to interact with the devices. This interface is presented in Figure 5.12. Notice that, in contrast to the Modbus device, the user has now the ability to actuate on the device. This can be done via the text input box (to change the heater setpoint) and the checkbox (to change the heater state).

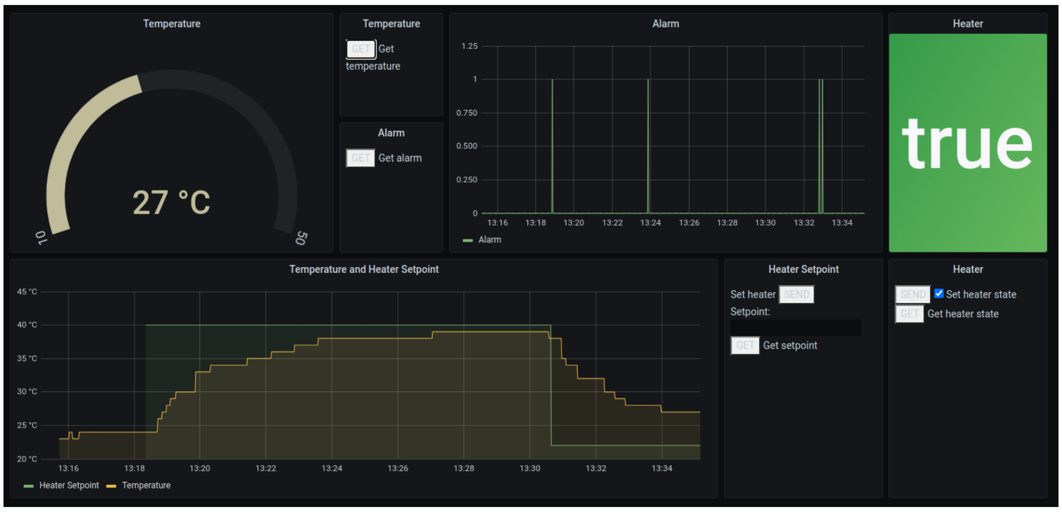


Figure 5.12: MQTT Device Dashboard.

5.2.3 KNX Device

The last protocol added to the framework was KNX. In this case, multiple commercial KNX devices were used. These devices communicate via a bus. A KNX BAOS was used to make the interface between the bus and the gateway. This device provides multiple tools to allow a third-party application to interact with KNX devices.

First, the devices were set up as shown in the simplified schematic presented in Figure 5.13. A KNX power supply was used to provide energy to all devices, except the DALI gateway and the energy meter, via the KNX bus. This means that the bus was used not only to allow communication between devices but also to power them. The ETS5 Professional configuration software tool was used to design and configure the KNX devices in the bus. This tool, which runs on Windows machines, makes the connection with the KNX bus via a USB interface device. In short, and among other functionalities, ETS allows designers and developers to manage the KNX devices and to create interactions between them. For example, it can be used to turn on a DALI Lamp via the DALI gateway when the KNX presence detector detects someone. Additionally, this software tool allows the users to make device-specific configurations, for example, fine-tuning the presence detection thresholds.

In the scope of this project, the objective was not to use the ETS tool as a way to automatize and create rules between the devices. The aim was instead to configure the devices to allow the KNX gateway to access and control all other instruments. After that was done and validated, the KNX device service was developed.

As stated before, EdgeX Foundry does not provide support for the KNX protocol. For this reason, the device service SDK was used to implement the custom microservice. The tools and dependencies required by the kit are specified in the documentation [3]. There are tools to develop device services in Go Lang and C. In the scope of this project, Go Lang was selected due to the fact that most of the gateway components were implemented in this language.

In short, the SDK provides the bare-bones to implement a device service. It already imple-

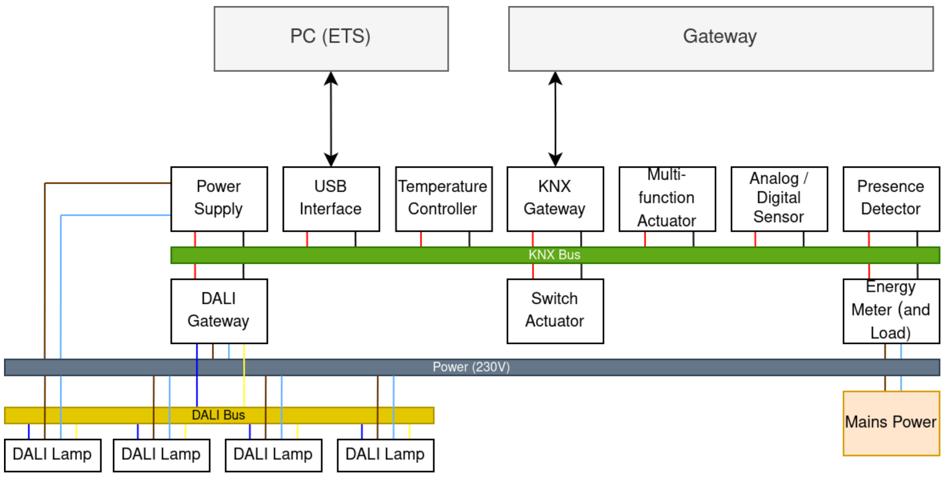


Figure 5.13: Simplified Schematic of the KNX Devices.

ments the component that establishes communication with the Core Services, so developers should focus on two main functions, namely, *HandleWriteCommands*, and *HandleReadCommands*, to, as the names imply, handle read and write commands from the gateway. When these handlers are called, the system establishes a TCP/IP connection with the KNX gateway to either collect a specific measurement from the KNX bus (read command) or actuate on a specific KNX device (write command).

After implementing it, the KNX device service was "dockerized", i.e., a container was created to run that microservice. Then, this container was added to Docker Hub and the docker-compose file of the framework. As done before, a configuration file (and device profile) were mounted as volumes to the container. This allows developers to make specific changes without the necessity to deploy a new container.

Although multiple devices were used, only one device profile was created. This is due to the fact that, from the framework's point of view, there is only one KNX device, and that is the KNX gateway. So, that single device profile, in fact, describes resources from multiple KNX devices.

Finally, a dashboard was created from where it was possible to monitor the device resources and to actuate on the devices intuitively. In Figure 5.14 it is shown part of the dashboard developed, where it is possible to see measurements from the energy meter, switch actuator, and a DALI lamp (via the DALI gateway) devices. Multiple visualization tools were used to represent the data, and different buttons were used to allow the user to actuate on the devices.

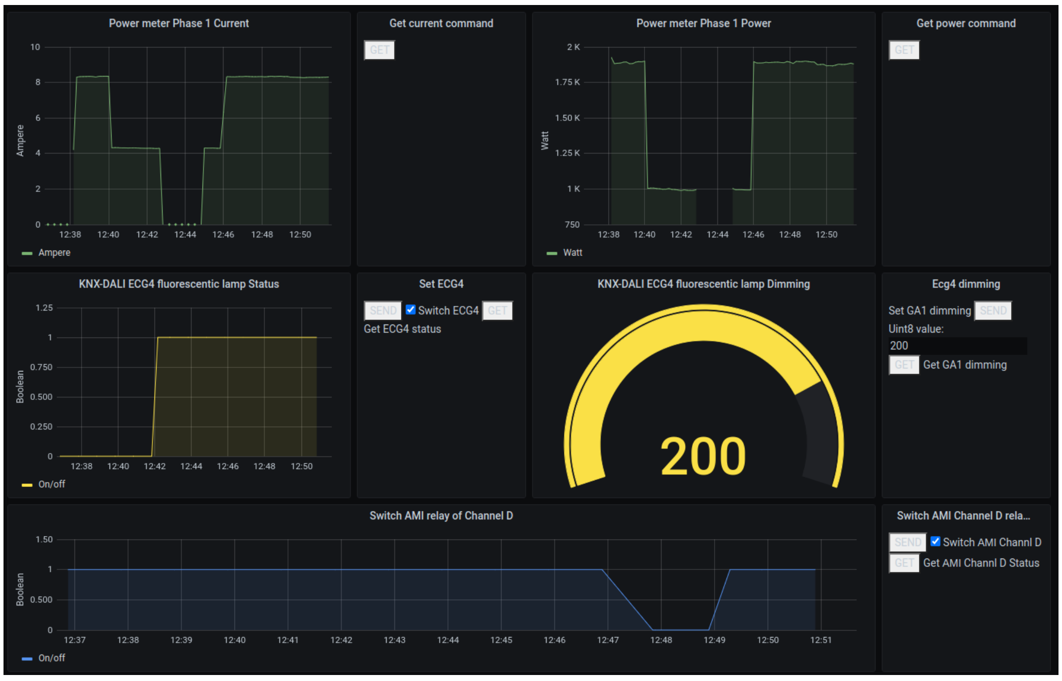


Figure 5.14: Part of the KNX Device Dashboard.

CHAPTER 6

Message Prioritization Mechanism

This chapter presents how the message prioritization feature was implemented and validated. It is relevant to emphasize that this change in the system only affected the communication between the devices and the gateway. Regarding hardware, it consisted of using an OpenFlow switch and an OpenFlow controller instead of a conventional network switch. Then, all the configurations were done in those two elements, which means that neither the devices nor the CTM system framework suffered changes.

Due to the separation mentioned before, it was possible to implement and validate the message prioritization mechanism in a simplified system. The network diagram of that system is graphically represented in Figure 6.1.

The simplified version of the CTM system implemented is composed by five main functional elements: Switch and Controller (Section 6.1), Mock Gateway (Section 6.2), Mock devices (Section 6.3), MQTT Broker (Section 6.4), and TG (Section 6.5).

When comparing to the CTM system framework developed, the gateway based on EdgeX Foundry was replaced by a Python script that receives and processes specific MQTT messages sent by devices. Although the algorithm does not implement a gateway, the term *mock gateway* was used to ease the comparison between the CTM system framework and its simplified version. For the same reason, the phrase *mock device* was used when referring to the MQTT devices used to validate the message prioritization mechanisms.

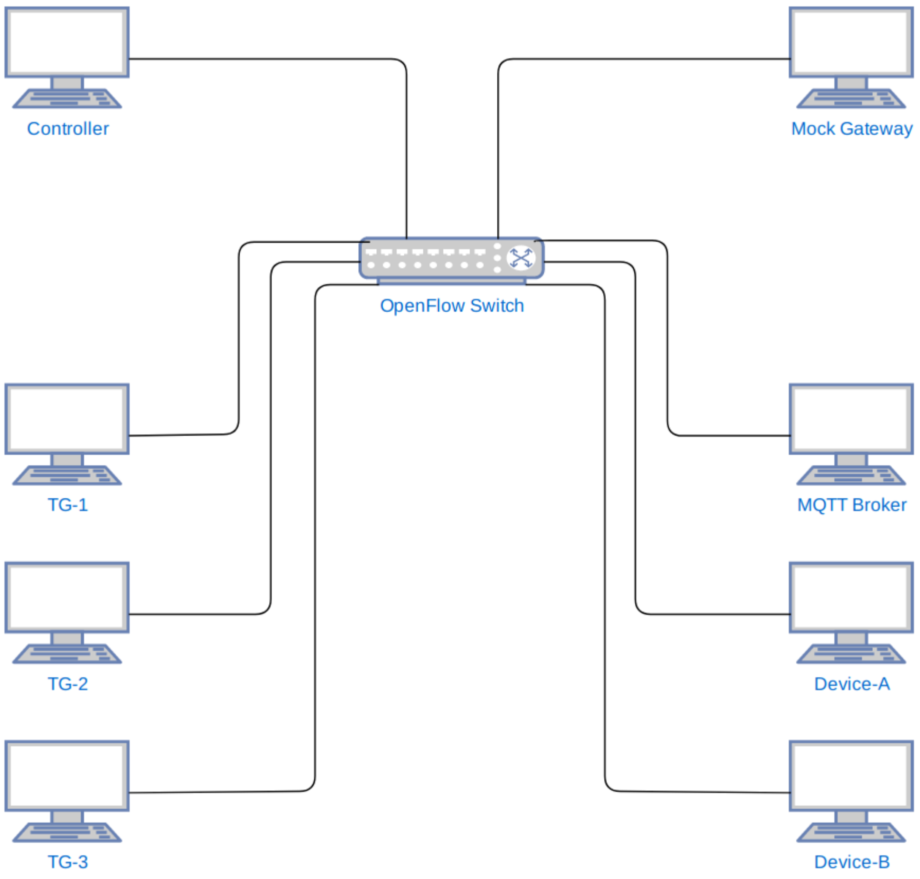


Figure 6.1: Network Diagram to Test the Message Prioritization Mechanisms.

6.1 Switch and Controller

As mentioned before, for testing purposes, we used the same hardware for both setups 1 and 2 that implement, respectively, a conventional and an SDN switch. In the first setup, the role of the switch is to forward incoming packets to the correct destination port. On the other hand, in the second setup, we added the bandwidth reservation feature to guarantee enough bandwidth for the MQTT messages. This feature was added via the switch CLI, whereas the packet forwarding feature is implemented by the controller, as explained below.

Firstly, it is necessary to create a bridge on the switch and add the seven required ports. Then, it is needed to add the controller to the system. One can do these three configurations via the switch CLI. Note that the controller uses a specific switch port meant for management.

It is important to stress that this experiment was meant to test a network with Fast Ethernet speed in Full-Duplex mode. However, some elements like the switch, the mock gateway, and TG-1 have Gigabit Ethernet Interfaces. At first, the switch ports were configured to limit their speed to 100 Mb/s in Full-Duplex mode. However, we observed that although the switch's configurations stated that all the ports were configured in Full-Duplex mode, the links were working in Half-Duplex mode. We configured the switch to perform auto-negotiation (default setting) to solve this issue. So, the machines with Gigabit Ethernet interfaces were configured to advertise their link's speed as 100 Mb/s in Full-Duplex mode. This was done

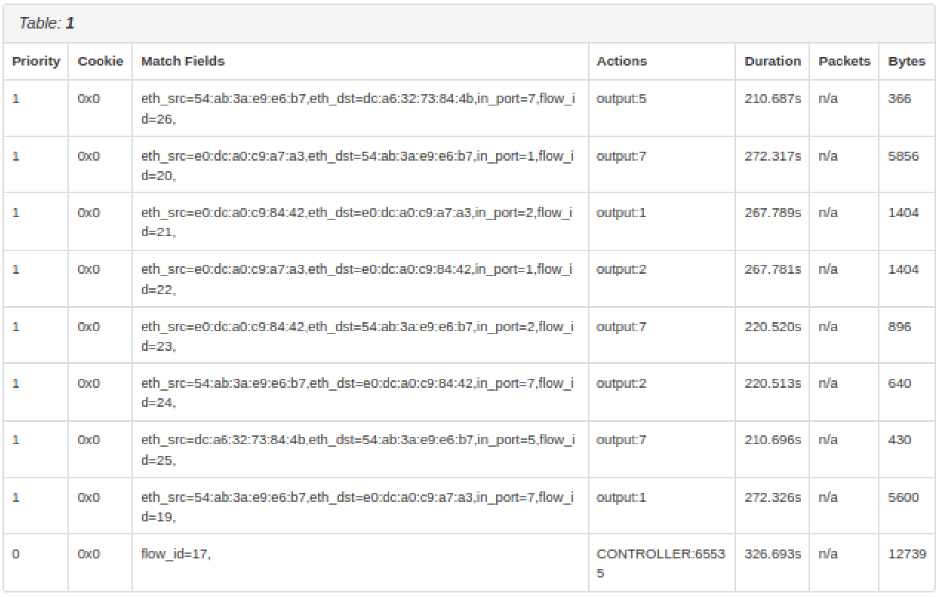


Table: **1**

Priority	Cookie	Match Fields	Actions	Duration	Packets	Bytes
1	0x0	eth_src=54:ab:3a:e9:e6:b7,eth_dst=dc:a6:32:73:84:4b,in_port=7,flow_id=26,	output:5	210.687s	n/a	366
1	0x0	eth_src=e0:dc:a0:c9:a7:a3,eth_dst=54:ab:3a:e9:e6:b7,in_port=1,flow_id=20,	output:7	272.317s	n/a	5856
1	0x0	eth_src=e0:dc:a0:c9:84:42,eth_dst=e0:dc:a0:c9:a7:a3,in_port=2,flow_id=21,	output:1	267.789s	n/a	1404
1	0x0	eth_src=e0:dc:a0:c9:a7:a3,eth_dst=e0:dc:a0:c9:84:42,in_port=1,flow_id=22,	output:2	267.781s	n/a	1404
1	0x0	eth_src=e0:dc:a0:c9:84:42,eth_dst=54:ab:3a:e9:e6:b7,in_port=2,flow_id=23,	output:7	220.520s	n/a	896
1	0x0	eth_src=54:ab:3a:e9:e6:b7,eth_dst=e0:dc:a0:c9:84:42,in_port=7,flow_id=24,	output:2	220.513s	n/a	640
1	0x0	eth_src=dc:a6:32:73:84:4b,eth_dst=54:ab:3a:e9:e6:b7,in_port=5,flow_id=25,	output:7	210.696s	n/a	430
1	0x0	eth_src=54:ab:3a:e9:e6:b7,eth_dst=e0:dc:a0:c9:a7:a3,in_port=7,flow_id=19,	output:1	272.326s	n/a	5600
0	0x0	flow_id=17,	CONTROLLER:65535	326.693s	n/a	12739

Figure 6.2: Flow Table Example to Perform Packet Forwarding.

using the *ethtool* command [56].

For both setups, it was used the Ryu controller, which was installed on the machine responsible for running the controller, and then an application based on *simple_switch_13.py* [57] was executed. This Ryu application acts as a standard MAC-learning switch implemented in OpenFlow. By default, at start-up, it creates a flow entry with low priority (0) to redirect all the packets received to the controller. Then, whenever there is a new communication, it adds the MAC address of the transmitting host if it is not already there. When there is enough information about the hosts (source and destination MAC addresses and the respective ports), a new flow is added to the flow table with more priority (1) than the one created by default. So, for example, if Host-A (port 1) sends Host-B (port 2) a message, if the flow entry: *in_port: 1, eth_dst: Host-B, eth_src: Host-A, output: 2* is present in the flow table, then the packet will be forwarded to the Host-B on port 2 directly (without passing through the controller). If no flow entry matches the received packet, apart from the default one, the message is redirected to the controller that either floods the packet to all the switch ports or, if the packet's source MAC address was the only information left to add a new flow, the packet is then forwarded to the respective output port, and a new flow is added to the flow table. An actual flow table, with four devices connected to the switch, looks like the one presented in Figure 6.2.

The default *simple_switch_13.py* writes the new flow entries to flow table 0. That would work well for setup 1 but not for setup 2, where the objective is to attribute different packets to different queues based on their protocol. Since the selection of sensitive messages must be made before forwarding the packets to the output ports, flow table 0 was reserved for that implementation. This means that, for both setups, table 1 was used for packet forwarding. So, changes were applied to application *simple_switch_13.py* and saved to *qos_simple_switch_13.py* so that its flow entries are recorded in flow table 1 instead of flow table 0.

Finally, for setup 1 (traditional switch) to be implemented entirely, it is necessary to add a flow entry to table 0. That flow will simply forward all packets to table 1 since no message prioritization features are to be implemented. The switch will not work correctly without that flow since the packets always start to be processed in table 0. So, if flow table 0 is empty, the packets are dropped. After adding the flow entry, using the switch CLI setup 1 is completed.

To properly implement setup 2 (SDN switch), it is required to (i) define two QoS queues in all seven ports of the bridge, and (ii) create flow entries to match the packets with the respective queues. Both steps can be done via the switch CLI after the former step is completed. Each port of the bridge should have two queues as follows:

- Queue 0 - Default
- Queue 1 - Minimum bandwidth of 5 Mb/s

Moving to the flow entries, for setup 2, table 0 is not simply used to forward packets to table 1. In fact, its objective is to filter the sensitive MQTT packets and, when detected, set them to go to queue 1 (the one with at least 5 Mb/s), while all the remaining packets are set to go to queue 0 (the default one). After being assigned to the desired egress queues, all messages move to flow table 1, where they will be forwarded to the respective switch port or the controller. So, to implement setup 2, all the previous configurations must be kept apart from flow table 0. The flow entry created before, that dispatches all traffic to table 1, must be deleted. Next, in flow table 1, three flows must be added. Two flows to detect MQTT packets and set them to the priority queue, and the other to assign to the default queue all the remaining packets. Finally, after completing the configurations described, using the switch CLI, setup 2 is successfully configured.

The necessity to add two flows is to prioritize MQTT messages from the transmitting host or hosts to the broker, as well as from the broker to the receiving host or hosts. So, as the broker uses the network port 1883, it is necessary to prioritize TCP/IP messages (given that MQTT runs over TCP/IP) whose network destination or source port, respectively, is 1883. In short, to identify MQTT packets, a flow entry must specify:

- Protocol Type (eth_type) - 0x0800 (IPv4)
- IP Protocol Number (ip_proto) - 0x06 (TCP/IP)
- Network Port (tp_dst or tp_src) - 1883

Figure 6.3 presents the flow tables 0, from setups 1 and 2. Notice that these flow tables have the role of selecting the sensitive packets and setting them to the specific egress queues. The process of forwarding packets to the switch ports is done in table 1, which is the same for both setups. The process of adding the flow entries to table 1 is done automatically by the Ryu controller. An example of this table can be found in Figure 6.2

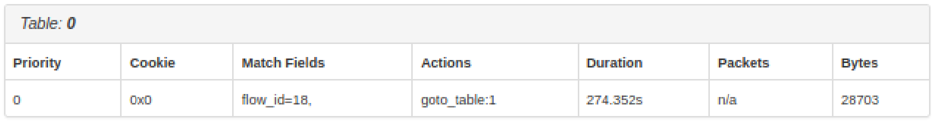


Table: ***0***

Priority	Cookie	Match Fields	Actions	Duration	Packets	Bytes
0	0x0	flow_id=18,	goto_table:1	274.352s	n/a	28703

(a) Setup 1

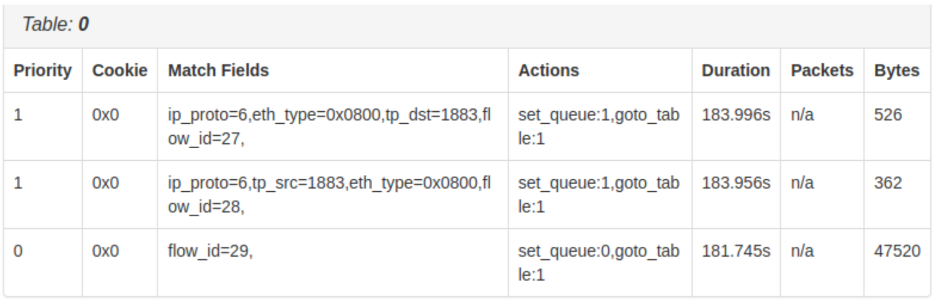


Table: ***0***

Priority	Cookie	Match Fields	Actions	Duration	Packets	Bytes
1	0x0	ip_proto=6,eth_type=0x0800,tp_dst=1883,flow_id=27,	set_queue:1,goto_table:1	183.996s	n/a	526
1	0x0	ip_proto=6,tp_src=1883,eth_type=0x0800,flow_id=28,	set_queue:1,goto_table:1	183.956s	n/a	362
0	0x0	flow_id=29,	set_queue:0,goto_table:1	181.745s	n/a	47520

(b) Setup 2

Figure 6.3: Table 0 Flow Entries From Setup 1 (a) and Setup 2 (b).

6.2 Mock Gateway

The computer that mocks the gateway is responsible for initializing the stress test experiences, receive, process and save the MQTT packets, and analyzing them afterwards. To this end, three Python scripts were written to (i) run the tests; (ii) analyze the raw data, process it, and retrieve relevant data such as jitter and latency estimations; and (iii) analyze the raw data, process it, and retrieve relevant data such as jitter and latency estimations.

The first script was implemented using the library Eclipse Paho [58] to establish MQTT communication. The working principle of this script is:

1. Start the test by establishing communication with the MQTT broker and subscribing to the topic **tests_run**, where the messages from the mock devices will be received.
2. Send the stress test configuration file to the topic **tests_setup**.
3. When a message is received, via the topic **tests_run**: verify that the message is valid (the test_id must match the one specified in the configuration file) and store it along with the arrival timestamp in a text file.
4. The tests are concluded after one of the following two scenarios: all mock devices' last messages are received or after a timeout (in cases of packet loss).

The configuration file uses the JSON format and can be edited in the mock gateway. An example of a configuration file is presented in Listing 5. Note that the only information that is relevant for the mock devices is: *test_id* (unique test identifier), *devices* (list containing the names of the hosts that should participate in the test), *num_messages* (number of messages to send), and *interval_ms* (the period, in ms, to transmit the messages). The configuration file's remaining content refers to traffic generation, and it is only useful for post-processing purposes.

The messages generated by the MQTT mock devices also use the JSON format. They contain the transmitting hostname, the test_id, the timestamp referring to the moment the message was prepared, and the message that is a monotonous sequence of integer numbers

```
{
  "test_id": 25,
  "devices": [
    "device-A",
    "device-B"
  ],
  "num_messages": 36000,
  "interval_ms": 50,
  "trfc_gen_machines": [
    "mock-gateway",
    "tg-1",
    "tg-1",
    "tg-2"
  ],
  "trfc_mbps_per_machine": 40,
  "trfc_protocol": "UDP"
}
```

Listing 5: Example of a Configuration File.

```
{"device": "device-A", "test_id": 25, "timestamp": 1621370507152, "msg": 0}
{"device": "device-B", "test_id": 25, "timestamp": 1621370507153, "msg": 0}
{"device": "device-B", "test_id": 25, "timestamp": 1621370507163, "msg": 1}
{"device": "device-A", "test_id": 25, "timestamp": 1621370507162, "msg": 1}
```

Listing 6: Example of the MQTT messages transmitted by the devices.

```
{"device": "device-A", "test_id": 25, "timestamp": 1621370507152, "msg": 0,
↪  "timestamp_arrival": 1621370507160}
{"device": "device-B", "test_id": 25, "timestamp": 1621370507153, "msg": 0,
↪  "timestamp_arrival": 1621370507164}
{"device": "device-B", "test_id": 25, "timestamp": 1621370507163, "msg": 1,
↪  "timestamp_arrival": 1621370507174}
{"device": "device-A", "test_id": 25, "timestamp": 1621370507162, "msg": 1,
↪  "timestamp_arrival": 1621370507179}
```

Listing 7: Example of the MQTT messages modified by the mock gateway.

from 0 to (num_messages - 1). After receiving a message, the arrival timestamp is added to the message before it is saved to a text file. The test's configuration file is also saved to the text file, alongside the modified messages, so that all the information required for post-processing is saved in a single place. Examples of the messages sent by the mock devices, as well as the modifications made in the mock gateway, can be found in Listing 6 and Listing 7, respectively.

After performing the tests, the scripts developed to analyze the data were executed. The main results obtained are presented in Chapter 7.

6.3 Mock Devices

The mock devices, implemented on the SIMATIC IOT2040 machines, have the role of simulating real MQTT devices or gateways. In an industrial environment, these devices would be acquiring factory floor data and other MQTT devices' data and then sending them to a

CTM system. In this experiment, there were no sensors, so the data was generated on the IOT2040 machines based on the configuration files received.

A Python script was written, also based on the Eclipse Paho library, and its working principle is the following:

1. Establish communication with the MQTT broker and subscribe to the topic **tests_setup** where the mock gateway will eventually send the configuration file.
2. When a configuration file is received, it starts the test if its hostname is specified in the device list.
3. The mock device publishes MQTT messages with the same format as shown in Listing 6, in topic **tests_run**. The number of messages, periodicity, and test_id are specified in the received configuration file.
4. When the test ends, the device waits for the mock gateway to send a new configuration file and repeats the process from point 2.

6.4 MQTT Broker

The MQTT broker runs on a Raspberry Pi 4. The central role of the broker in MQTT is to analyze the received packets' topics and forward them to the hosts that subscribed to that same topic. In this experiment, the MQTT broker used was Eclipse Mosquitto. All the default configurations were kept, meaning that the network port 1883 was used as the transmission and reception points.

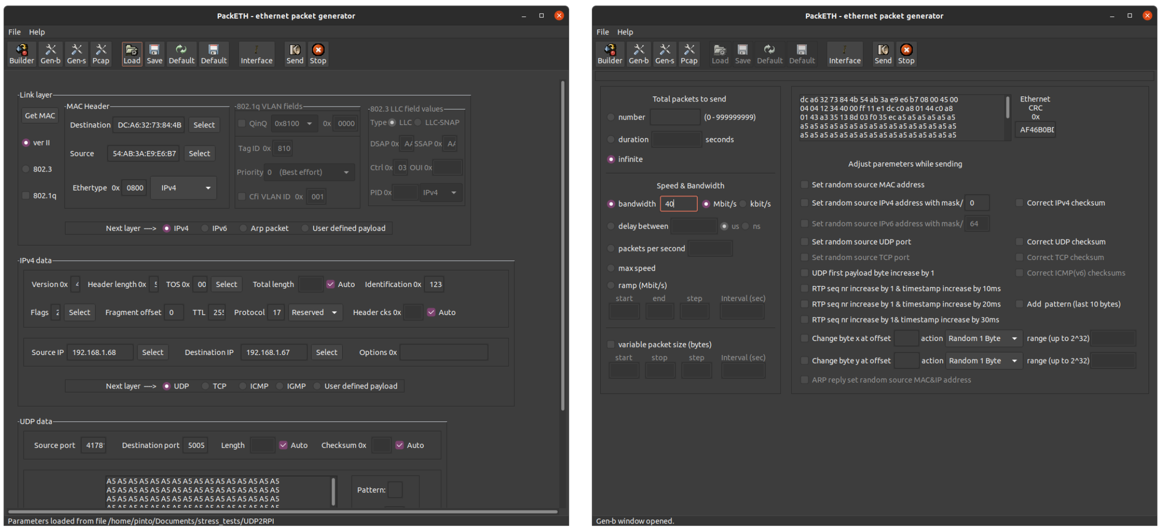
6.5 Traffic Generators

In this experiment, we used three computers whose only purpose was to generate network traffic. The use of these components aimed to mock real traffic in an industrial environment. The software packETH [59] was used to generate traffic. For all the experiments, the target of the packets generated was the Raspberry Pi (MQTT broker). To configure packETH accordingly, it was necessary to specify:

- Source IP address and port.
- Destination IP address and port (Raspberry Pi's IP address and port 1883)
- Protocol (e.g., UDP)
- Message Pattern (e.g., 0xAA)
- Packet Length (e.g., 1000 bytes)

An example of these configurations can be observed in Figure 6.4. Next, to generate traffic in the network, it is necessary to set the bandwidth intended to be occupied by these packets. For this experiment, to simulate a stressed network, the three TG machines and the mock gateway generated UDP traffic at a rate of 160 Mb/s. This means that each traffic generator machine and the mock gateway sent packets to the MQTT broker at a rate of 40 Mb/s each.

A simple UDP server was implemented, in Python, to run on the Raspberry Pi. We used this server for two reasons: (i) for debugging purposes, to analyze if the UDP packets were arriving at the rates defined in packETH; and (ii) to avoid creating more than the desired traffic due to the unreachable port response from the Raspberry Pi.



Link layer

Get MAC

[✓ ver II] 802.3 802.1q

MAC Header

Destination DC:A6:32:73:84:4B Select

Source 54:AB:3A:E9:E6:B7 Select

Ethertype 0x 0800 IPv4

802.1q VLAN fields

QinQ 0x8100 0x 0000

Tag ID 0x 810

Priority 0 (Best effort)

Cfi VLAN ID 0x 001

802.3 LLC field values

Type LLC LLC-SNAP

DSAP 0x AA SSAP 0x AA

Ctrl 0x 03 OUI 0x

PID 0x IPv4

Next layer ---> [✓ IPv4] IPv6 Arp packet User defined payload

IPv4 data

Version 0x 4 Header length 0x 5 TOS 0x 00 Select Total length [✓ Auto] Identification 0x 123

Flags 2 Select Fragment offset 0 TTL 255 Protocol 17 Reserved Header cks 0x [✓ Auto]

Source IP 192.168.1.68 Select Destination IP 192.168.1.67 Select Options 0x

Next layer ---> [✓ UDP] TCP ICMP IGMP User defined payload

UDP data

Source port 4178 Destination port 5005 Length [✓ Auto] Checksum 0x [✓ Auto]

```
A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5
A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5
A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5
```

Pattern:

Parameters loaded from file /home/pinto/Documents/stress_tests/UDP2RPI

Total packets to send

number (0 - 999999999)

duration seconds

[✓ infinite]

Speed & Bandwidth

[✓ bandwidth] 40 [✓ Mbit/s] kbit/s

delay between us ns

packets per second

max speed

ramp (Mbit/s) start end step Interval (sec)

variable packet size (bytes) start stop step Interval (sec)

```
dc a6 32 73 84 4b 54 ab 3a e9 e6 b7 08 00 45 00
04 04 12 34 40 00 ff 11 e1 dc c0 a8 01 44 c0 a8
01 43 a3 35 13 8d 03 f0 35 ec a5 a5 a5 a5 a5 a5
a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5
a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5
```

Ethernet CRC 0x AF46B0BD

Adjust paremeters while sending

Set random source MAC address

Set random source IPv4 address with mask/ 0 Correct IPv4 checksum

Set random source IPv6 address with mask/ 64

Set random source UDP port Correct UDP checksum

Set random source TCP port Correct TCP checksum

UDP first payload byte increase by 1 Correct ICMP(v6) checksums

RTP seq nr increase by 1 & timestamp increase by 10ms

RTP seq nr increase by 1 & timestamp increase by 20ms Add pattern (last 10 bytes)

RTP seq nr increase by 1& timestamp increase by 30ms

Change byte x at offset action Random 1 Byte range (up to 2^32)

Change byte y at offset action Random 1 Byte range (up to 2^32)

ARP reply set random source MAC&IP address

Gen-b window opened.

Figure 6.4: packETH Configuration Example.

CHAPTER 7

Tests, Results, and Discussion

This chapter defines the tests performed to validate the system and discusses the main results obtained from them. In particular, Section 7.1 presents the tests performed regarding the framework developed for CTM systems, these tests aimed to evaluate the framework's scalability. On the other hand, Section 7.2 presents the tests to validate the message prioritization mechanism implemented and its consequent results and discussion.

7.1 Framework for CTM Systems

The framework for CTM systems was validated by assuring that it was possible to (i) analyze the data gathered from the devices in the GUI, and (ii) control the devices via the GUI. In other words, the system ensures end-to-end communication. We repeated this process for different devices, each with a distinct communication protocol (KNX, Modbus, and MQTT). Finally, we tested the framework in terms of its scalability.

The scalability tests conducted focused on the MQTT protocol. However, we did not use the MQTT device previously developed (based on an ESP32 board). Instead, personal computers and a Raspberry Pi were used. This choice was made because there was only one ESP32 board available. In addition, since we intended to test the system's scalability, it would be necessary to have more devices to test different scenarios.

So, in terms of hardware, we used a switch, five personal computers, and a Raspberry Pi 4. The latter used the Raspberry Pi OS 5.10, whereas two personal computers ran Ubuntu OS 20.04 and the remaining Lubuntu OS 19.04. All of these machines ran mock MQTT devices for testing purposes. Additionally, one of the personal computers ran the system, i.e., the gateway, infrastructure, and MQTT broker. This setup is presented in Figure 7.1.

In terms of software, we implemented two types of mock MQTT devices:

- Automatic devices - These devices sent data to the gateway, asynchronously, with a frequency defined in the device.
- Responding devices - These devices respond to request performed by the gateway.

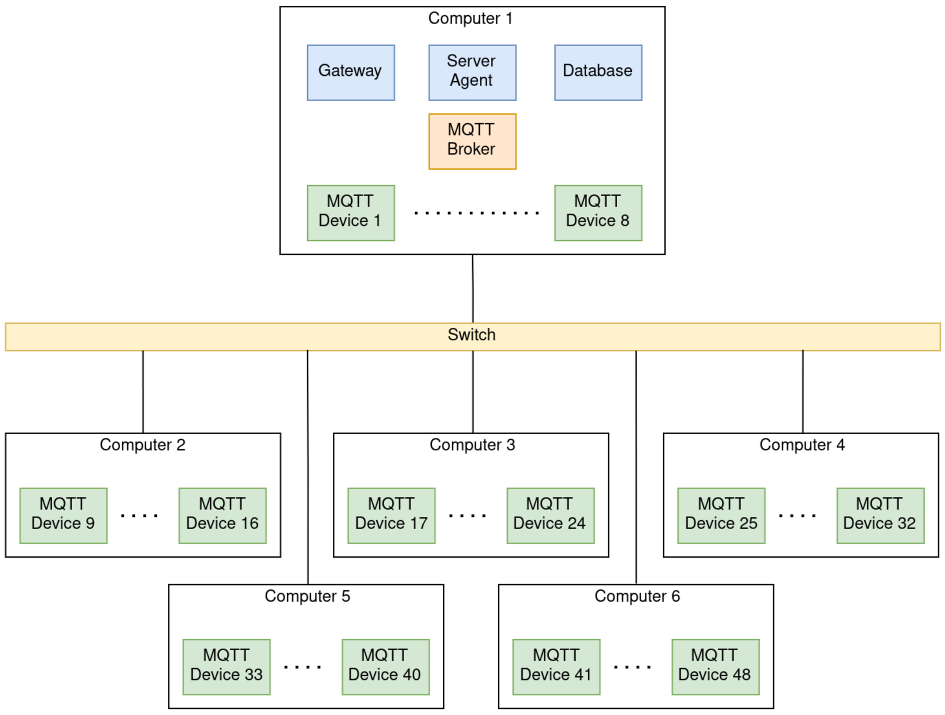


Figure 7.1: Setup to Test the Framework.

These two types allowed to test the three possible ways that the gateway has to gather MQTT data: asynchronous messages sent by the devices, through the gateway's REST API and via auto-events. The last two made use of the *Responding devices* since in these scenarios the gateway is responsible for requesting the data it intends to receive.

These mock devices were developed in Python 3, based on the Eclipse Paho library. They were designed to send to the gateway, in JSON format, three components: a UNIX timestamp of when the device prepared the message, a sequence number, and a test identifier. Thus, both automatic and responding devices sent the same information, either periodically or after a gateway's request, depending on their type. It was also defined a command called "reset" to allow the gateway to assign a specific test ID and reset the sequence number of the devices.

In total, we used six machines to perform the scalability tests. However, to simulate a more realistic situation, each machine executed up to eight devices. This means that eight scenarios were considered, from one device per machine (six devices) to eight devices per machine (forty-eight devices).

From the gateway's point-of-view, we created a single MQTT mock device. Forty-eight resources were added to the device, one for each device. In practice, we used these resources to acquire and store the JSON information sent by each device. An additional "reset" resource was created to allow third-party software to control the tests via the gateway's REST API.

The scalability tests considered four aspects: packet loss, latency, CPU usage, and memory usage. Since the entire system (except the devices) runs on Docker, the performance metrics were obtained using tools provided by Docker (namely, the command *docker stats*). Thus, we studied the CPU and memory usages of the machine that runs the gateway, infrastructure, and MQTT Broker. That machine has 8GB of RAM and an Intel® Core™ i7-7500U CPU.

Finally, we performed several scalability sets of tests to reach the results presented in the following subsection. The results presented were obtained from a set of tests that consisted of acquiring data sent asynchronous, with a period of 1s, from the mock devices. For each test, which took approximately 25 minutes, we used a different number of devices. The system was restarted in between tests. After performing the scalability tests, the system's latency was studied, but this time, using responding devices instead of automatic devices. The objective of this test was to analyze the latency in detail, taking UNIX timestamps in multiple points of the system. These tests consisted of making 1500 consecutive requests to the gateway, via the REST API.

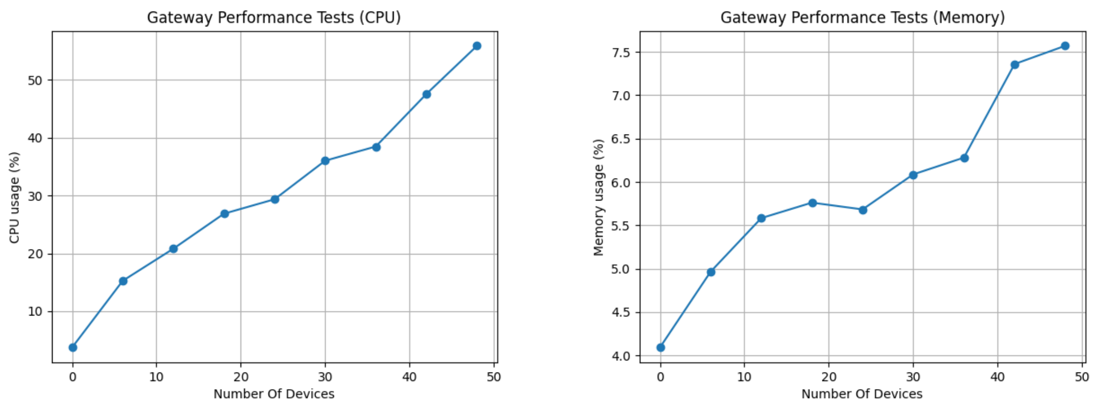
7.1.1 Results

The scalability tests regarding CPU and memory usage of the gateway, according to the number of devices used, are presented in Figure 7.2. As already discussed, the gateway is a collection of multiple layers of services, so Figure 7.3 presents the evolution of the resources' usage of each layer of services. Then, Figure 7.4 shows the same metrics of the last two figures, but regarding the infrastructure services and the MQTT broker.

Figure 7.5 presents the evolution of the latency, from the moment a message is prepared in the devices until it is inserted into the database. Notice that the points represented in the chart correspond to the average latency values of all the devices used in the respective test.

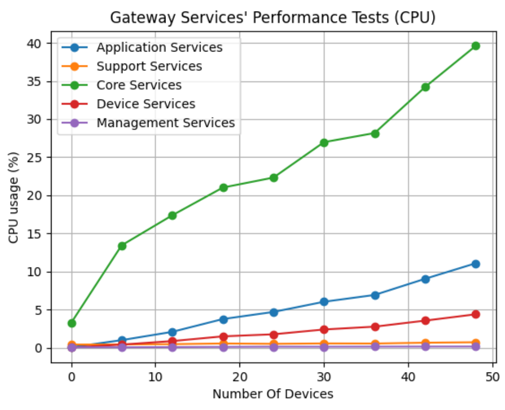
For all the scalability tests performed, no packets were lost, and thus, this metric is not graphically represented in this section.

Finally, we performed a different set of tests to understand better which system components contributed the most to the latency. The results of such tests are presented in Table 7.1.

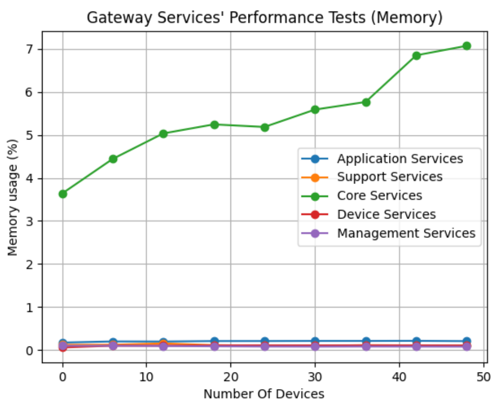


(a) CPU **(b)** Memory

Figure 7.2: Gateway Scalability Results Regarding (a) CPU and (b) Memory Usage.

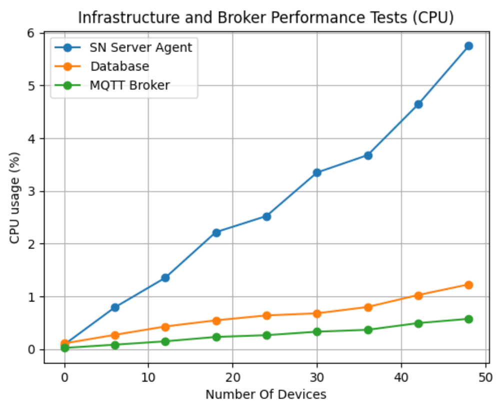


(a) CPU

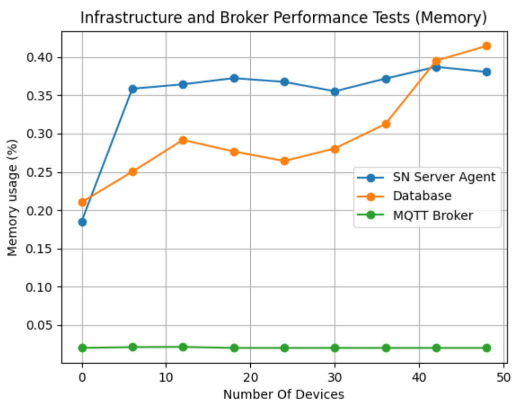


(b) Memory

Figure 7.3: Gateway Services' Scalability Results Regarding (a) CPU and (b) Memory Usage.



(a) CPU



(b) Memory

Figure 7.4: Infrastructure and MQTT Broker Scalability Results Regarding (a) CPU and (b) Memory Usage.

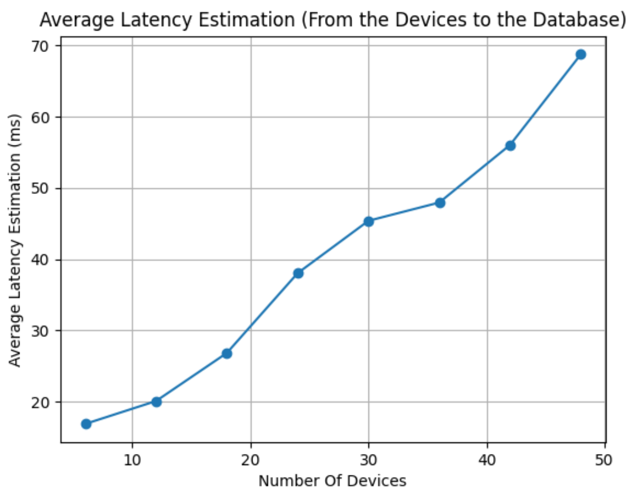


Figure 7.5: Average Latency Estimation Regarding the Number of Devices.

Timestamp Location		Latency	
Start	End	Average (ms)	Standard Deviation (ms)
Request (out)	MQTT Device (in)	46.39	34.40
MQTT Device (out)	Core Service (out)	976.91	100.1
Core Service (out)	Application Service (out)	0.40	2.75
Application Service (out)	NS Server Agent (in)	12.67	3.50
NS Server Agent (out)	Database (in)	8.06	4.21

Table 7.1: Latency From the Request to he Database

7.1.2 Discussion

From Figure 7.2 it is possible to conclude that for up to forty-eight devices, the usage of the resources from the gateway grows nearly linearly, both in terms of CPU and memory. In terms of the gateway's services, it is possible to observe in Figure 7.3 that the vast majority of the resources used by the gateway are consumed in its Core Services, which grow roughly linearly with the number of devices used. In terms of memory, all other services remained insensitive to increasing the number of devices. However, considering CPU usage, the Device Services and Application services also increased with the number of devices.

Regarding the infrastructure, only the SN Server Agent and Database were considered in the scalability tests. The GUI and the NS Server Agent were not affected by the number of devices in the system, but by the users' interaction with the system. Figure 7.4 shows that the Server Agent and Database CPU usage increase, practically, linearly while increasing the number of devices. Nevertheless, the former uses more resources than the latter, even though this usage is relatively reduced compared to the gateway. In terms of memory, the Server Agent appeared to be less sensitive to the number of devices' variation than the database, which increases with the number of devices. However, the memory usage of these infrastructure elements is substantially smaller than the gateway's usage.

Focusing on Figure 7.4 it is also possible to observe that the MQTT Broker's CPU usage increases linearly with the number of devices in the system. Yet, in terms of memory usage, it is insensitive to the same increase.

It is possible to conclude from Figure 7.5 that the latency, from the devices to the database, also increased approximately linearly with the number of devices. Furthermore, we studied this latency and observed that around 80% of it occurred from the gateway's device services to the gateway. Thus, the reaming time was spent from the devices to the gateway's device services.

Finally, in a different set of tests, it was possible to study in detail the latency, but this time from the moment of request to the moment the data is inserted in the database. We observed that for the cases when the gateway's performed a request to the devices (either caused by a command sent via REST API or by an auto-event trigger), the data would take more time to reach the database. After performing the tests, it was possible to conclude that device services were the main responsible for this latency, with an average value of 976.91 ms.

7.2 Message Prioritization Mechanism

Some tests were conducted to check whether the proposed solution solves the problem at hand. We performed those tests under two scenarios: one where the network is not saturated and the other where traffic is randomly generated to stress the network.

If the network is not saturated, no relevant traffic is being generated apart from the MQTT packets, which we consider time-sensitive. On the other hand, to simulate a scenario where the network is stressed, it is necessary to generate random traffic, so the software packETH was used. All four Traffic Generators are programmed to send 1 KB UDP packets, at a rate of 40 Mb/s each, to the Raspberry Pi 4 (MQTT Broker). So, as all network links were set to 100 Mb/s in Full-Duplex mode, those UDP packets will saturate the Raspberry Pi's network link.

Two tests, with both scenarios described above, were performed for each setup. For all tests, each mock device was configured to send 36,000 MQTT packets with a period of 50 ms. These packets were received and saved by the mock gateway and the receiving timestamp, which made it possible to compute the inter-arrival time and latency estimation. It is important to stress that the latency value is just an estimation, since the timestamps are not gathered precisely where the packets are sent and received. That would require specific hardware and since the objective of this project was to compare the implementations rather than getting precise absolute results, this approach was not followed. This aspect must be taken into consideration when analyzing the data in Subsection 7.2.1 and Subsection 7.2.2, i.e., latency estimation values should only be used as a way to compare the two setups implemented.

7.2.1 Results

In this section, the results obtained from the tests described before are presented. In Tables 7.2 and 7.3 it is possible to compare both setup 1 (Traditional Switch) and setup 2 (Intelligent/SDN Switch) under both saturated and non-saturated network scenarios. The average, standard deviation, and percentiles of the inter-arrival time and latency are presented, as well as the percentage of packets lost, to compare both setups. The terms *traditional* and *intelligent* regarding the switch are used to ease the analysis and comprehension of the results. The former refers to a common MAC-learning switch implemented in OpenFlow, whereas the latter refers to the implementation with bandwidth reservation to prioritize critical massages.

In Figures 7.6 and 7.7 the inter-arrival time and latency, respectively, are represented by their histogram. It is possible to make conclusions about the latency and jitter in a non-saturated environment in these figures. In the histogram from Figure 7.6 more than

Network Traffic	Setup	Inter-Arrival Time (ms)		Latency Estimation (ms)		Packet Loss (%)
		Average	Standard deviation	Average	Standard deviation	
Not Saturated	Traditional	50.17	0.71	12.27	1.32	0
	Intelligent	50.17	0.73	9.64	1.18	0
Saturated	Traditional	51.54	635.35	3842.30	7397.18	2.54
	Intelligent	50.17	3.18	13.63	7.19	0

Table 7.2: Inter-Arrival Time and Latency Estimation Main Results.

Network Traffic	Setup	Inter-Arrival Time Percentiles (ms)							Latency Estimation Percentiles (ms)						
		Min	1	10	50	90	99	Max	Min	1	10	50	90	99	Max
Not Saturated	Traditional	0	50	50	50	51	51	91	10	10	11	12	14	14	53
	Intelligent	0	50	50	50	51	51	91	8	8	8	10	11	11	51
Saturated	Traditional	0	0	0	0	13	892	92945	26	46	234	967	11025	41785	56485
	Intelligent	0	50	50	50	51	51	834	12	12	13	13	15	15	805

Table 7.3: Inter-Arrival Time and Latency Estimation Percentiles.

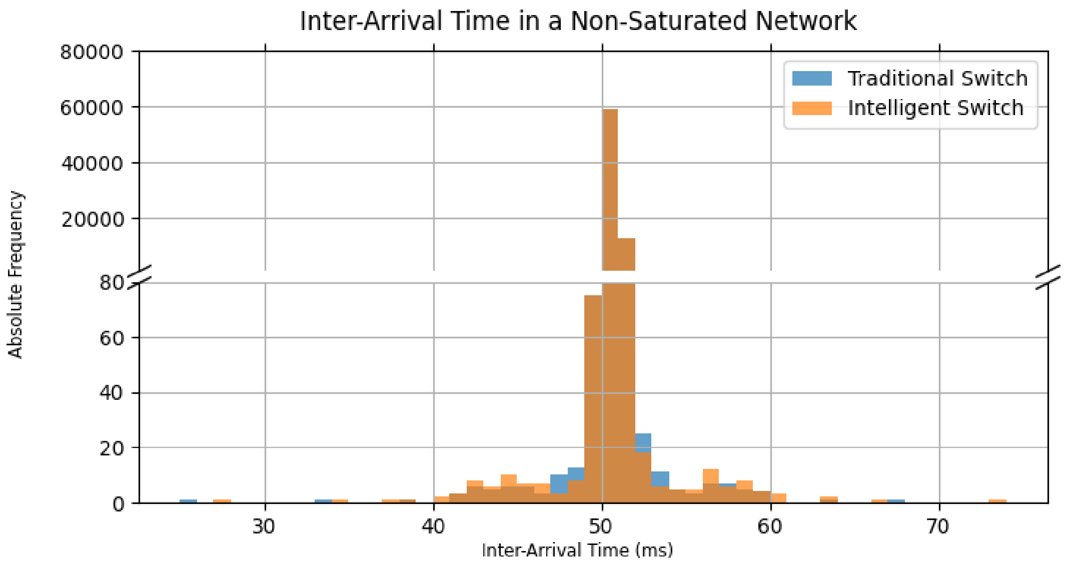


Figure 7.6: Inter-Arrival Time Histogram in a Non-Saturated Network.

99.97% of the data is represented. Whereas in the one from Figure 7.7 at least 99.99% of the information is conveyed.

In Figures 7.8 and 7.9, the inter-arrival time and latency, respectively, are represented in histogram form, but for this case, the network was stressed when the data was retrieved. It is possible to make conclusions about the latency and jitter in a saturated network in these figures. In the histogram from Figure 7.8 more than 99.62% of the data is represented. Whereas in the histograms from Figures 7.9a and 7.9b at least 97.87% and 52.26% of the data, respectively, are represented.

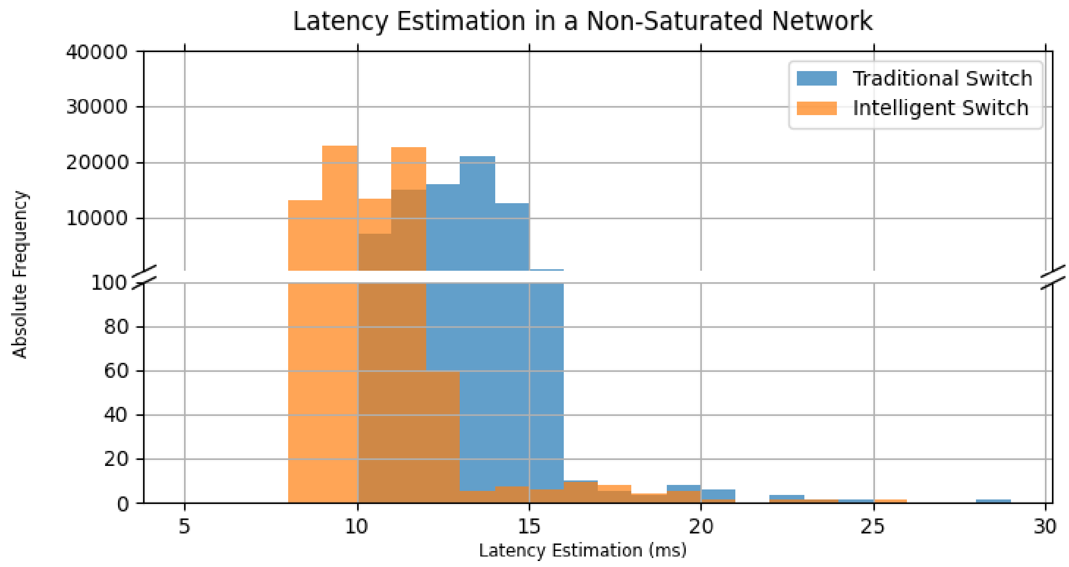


Figure 7.7: Latency Estimation Histogram in a Non-Saturated Network.

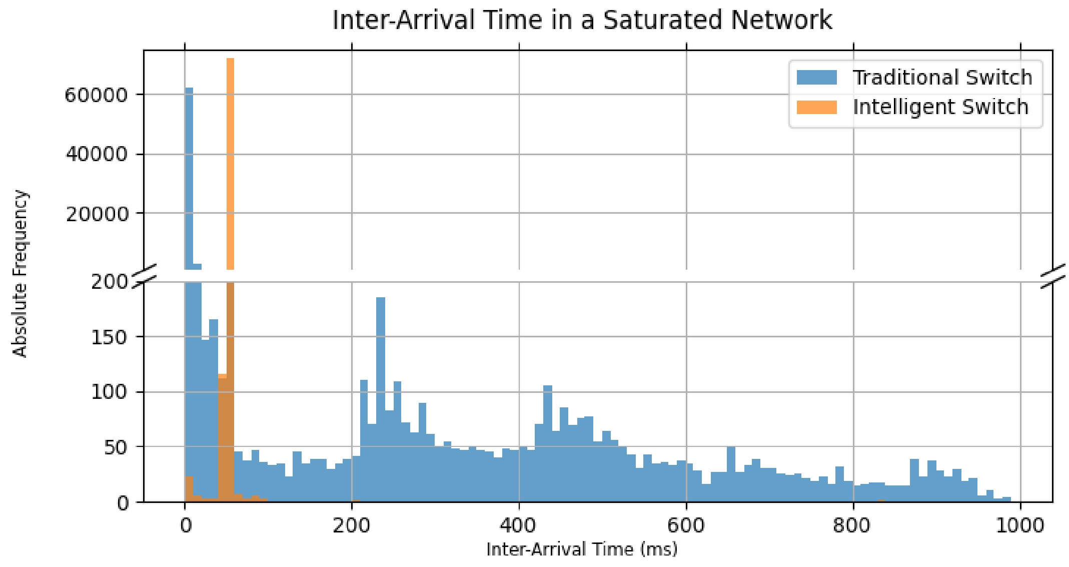
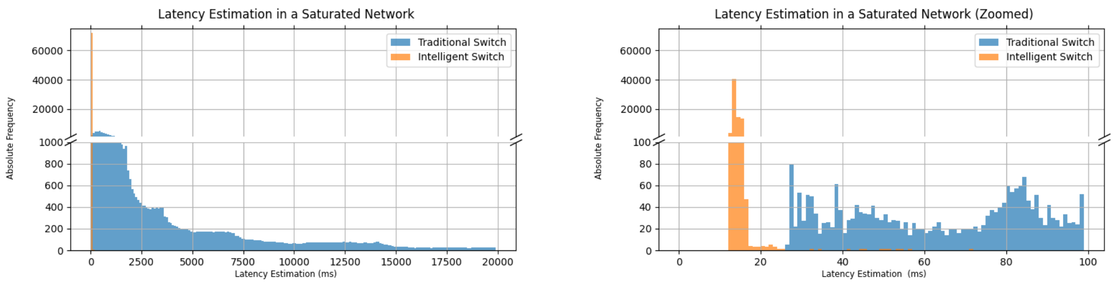


Figure 7.8: Inter-Arrival Time Histogram in a Saturated Network.



(a) **(b)**

Figure 7.9: Latency Estimation Histogram in a Saturated Network. (a) Not Zoomed (b) Zoomed.

7.2.2 Discussion

From Tables 7.2 and 7.3, as well as from Figures 7.6 and 7.7, it is possible to observe that when the network is not stressed, both setups 1 (Traditional switch) and 2 (Intelligent switch) have similar behavior in terms of jitter, latency, and packet loss.

On the other hand, it is possible to note that same does not happen when the network is saturated. In that case, using the traditional switch setup, 2.54% of the transmitted MQTT packets are not received, whereas when using the intelligent switch setup, no packets are lost. The latency estimation average value decreases approximately 99.64% when 5% of the bandwidth is reserved for the time-sensitive messages.

Under the same circumstances, the inter-arrival average time decreases approximately 2.66%, but its standard deviation drops around 99.50%. The decrease in the average inter-arrival time is not as significant as in the latency estimation, but when considering the percentiles (and maximum values), one can observe that at least 90% of the packets have a jitter value inferior to 13 ms. In addition, in 1% of them, the inter-arrival time was between 892 ms and 92945 ms (between 17.84 and 1858.9 times the ideal theoretical value), which lead to the conclusion that the mock gateway received most of the packets in bursts after being retained in the switch for too long, thus the relatively small jitter average value and the considerable latency average value.

Under the same saturation conditions, when we used the intelligent switch setup, the results were different, as shown in the histograms in Figures 7.8 and Figures 7.9. In this case, no packets were lost, and the latency estimation and inter-arrival time values, over time, were identical to the case where the network was not saturated. It was observed approximately no difference in the inter-arrival average time and a difference of around 9.98% in the latency estimation average value when comparing the intelligent switch under stress conditions and the traditional switch under normal conditions. This concludes that reserving 5% bandwidth for sensitive messages allows those messages to be exchanged as there was no stress at all in the network, even when there is.

Lastly, it is possible to conclude that a message prioritization mechanism is feasible using a network based on SDN/OpenFlow principles. With this implementation, the MQTT packets were prioritized over the remaining without being necessary to change the devices, the protocol, or the gateway, only requiring a commercial SDN switch and a proper configuration. So, although the tests were performed only using the MQTT protocol, it is concluded that using a similar approach, other IP-based protocols could benefit from the same strategy.

CHAPTER 8

Conclusion

This dissertation focused on describing the design, development, validation, and testing processes of a framework for CTM systems. At first, architectures for CTM were studied. With the architecture selected, we analyzed and compared several toolkits and frameworks to implement the system components. The system implemented may be divided into three elements: the framework (composed by the gateway, database, and GUI), message prioritization mechanism (implemented using the OpenFlow switch and controller), and the devices used to validate the system operation.

The framework's gateway was implemented based on EdgeX Foundry, which provided the required tools to connect to different devices with different communication protocols. The gateway application services were configured to export the data collected from the devices to a pre-determined HTTP endpoint in a JSON format. This endpoint was the SN server agent, implemented using Telegraf. That server agent was responsible for inserting the data from the gateway into the database (InfluxDB). Once the data was stored in the database, it could be presented to the user via the GUI based on Grafana. This SN communication was validated using the virtual devices provided by EdgeX Foundry. The same devices were used to test the NS connection that started in the GUI where the users send commands to the devices. These commands are stored in the database and consequently analyzed by the NS server agent. This server agent, developed in Python 3, is responsible for parsing the commands and sending them to the gateway via the REST API provided by EdgeX Foundy framework. Finally, the gateway processes and sends the commands to the devices using the respective communication protocol.

Furthermore, we added three devices to the framework: Modbus, KNX, and MQTT devices. The former was a single device connected to the gateway via Serial communication. Then, the *KNX device* is, actually, a group of eight KNX devices connected to a gateway that makes the interface between the KNX bus and the framework gateway. Finally, the MQTT device was implemented from scratch based on an ESP32 board. With these devices, we validated the system operation since it was possible to monitor and control them via the GUI.

Finally, we evaluated the framework on its scalability. We used up to forty-eight mock MQTT devices to study how the system behaves according to the number of devices connected to it. The results showed that the system's usage of resources (namely CPU and Memory) grew close to linearly with the number of devices. It was also possible to observe that the gateway's core services were the principal responsible for resource consumption. Moreover, although the latency (between the device and database) increased close to linearly with the number of devices used, no packets were lost. Finally, in a different set of tests, we analyzed the latency in detail. From there, it was possible to conclude that when the gateway's REST API or auto-event triggers are used to acquire data from the devices, the latency is more significant when comparing it to the case where data is generated asynchronously. Additionally, we observed that most of the time is spent between the gateway's device services and the application services in these periods.

Lastly, the message prioritization mechanism was implemented applying SDN/OpenFlow principles. The agent selected the critical messages based on specific fields of the packets and assigned them to a queue with a minimum of 5 Mb/s. In contrast, all the remaining packets were set to the default queue. This feature was tested in scenarios with diverse traffic conditions in the network, using two setups. One without the prioritization agent (traditional switch) and the other with the prioritization agent (SDN switch). The tests were performed considering that the MQTT protocol carried the time-sensitive information. The tests conducted under stress conditions show that the packet loss and high latency values observed, on MQTT packets, in the traditional switch setup were not verified when we implemented the message prioritization mechanism. It is, however, essential to note that we applied no additional measures to the remaining network traffic, which means that those messages were affected by the global network's stress conditions.

To conclude, as this work followed the work *Development of a Centralized Building Management System* [22], the future work suggestions presented there were taken in consideration. We accomplished four of the five suggestions during this dissertation. Additionally, the framework meets all the functional requirements agreed with the company *Diferencial* at the beginning of the project.

8.1 Future Work

Despite being functional, the framework developed could benefit from additional features that can be considered when carrying out the work described in this dissertation. Some of the aspects that can be explored include:

- Implement and test security features in the framework.
- Evaluate the framework under real-life scenarios, such as an industrial environment with multiple nodes and communication standards.
- Extend the message prioritization mechanism to other protocols.
- Implement a message prioritization agent that selects messages based on their content (e.g., a specific MQTT topic).

- Automate the message prioritization mechanism by applying a dynamic reservation of bandwidth. This way, it would not be necessary to define the QoS queues and their bandwidth boundaries statically.

References

[1] *Digital Illumination Interface Alliance.* [Online]. Available: https://www.dali-alliance.org/dali/.

[2] *KNX – Communication Protocol for Building Automation.* [Online]. Available: https://www.wago.com/us/knx.

[3] *EdgeX Foundry Documentation.* [Online]. Available: https://docs.edgexfoundry.org/1.2/.

[4] *EdgeX Foundry / Geneva Release Compose Files.* [Online]. Available: https://github.com/edgexfoundry/edgex-compose/tree/geneva.

[5] *ThingsBoard - Open-source IoT Platform.* [Online]. Available: https://thingsboard.io/.

[6] *Node-RED.* [Online]. Available: https://nodered.org/.

[7] *InfluxDB Time Series Platform.* [Online]. Available: https://www.influxdata.com/products/influxdb/.

[8] *Telegraf Open Source Server Agent.* [Online]. Available: https://www.influxdata.com/time-series-platform/telegraf/.

[9] *Apache Cassandra.* [Online]. Available: https://cassandra.apache.org/.

[10] *MongoDB.* [Online]. Available: https://www.mongodb.com/.

[11] *Grafana: The open observability platform.* [Online]. Available: https://grafana.com/.

[12] *Qt | Cross-platform software development for embedded and desktop.* [Online]. Available: https://www.qt.io/.

[13] *BACnet.* [Online]. Available: http://www.bacnet.org/.

[14] *Modbus Specifications and Implementation Guides.* [Online]. Available: https://www.modbus.org/specs.php.

[15] *Profibus.* [Online]. Available: https://www.profibus.com/.

[16] *EtherCAT.* [Online]. Available: https://www.ethercat.org/default.htm.

[17] *MQTT - The Standard for IoT Messaging.* [Online]. Available: https://mqtt.org/.

[18] *AMQP.* [Online]. Available: https://www.amqp.org/.

[19] *CoAP — Constrained Application Protocol.* [Online]. Available: https://coap.technology/.

[20] "KNX BASICS Smart home and building solutions. Global. Secure. Connected. KNX.ORG," Tech. Rep. [Online]. Available: https://www.knx.org/wAssets/docs/downloads/Marketing/Flyers/KNX-Basics/KNX-Basics_en.pdf.

[21] L. E. M. d. Silva, "A Real-Time Software-Defined Networking Framework for Cyber-Physical Production Systems," Ph.D. dissertation, Universidade de Aveiro, Sep. 2019. [Online]. Available: https://ria.ua.pt/handle/10773/29182.

[22] J. M. S. Ferreira, "Development of a Centralized Building Management System," Universidade de Aveiro, Tech. Rep., Feb. 2021. [Online]. Available: https://ria.ua.pt/handle/10773/31332.

[23] *Edge computing architecture: Overview - IBM Cloud Architecture Center.* [Online]. Available: https://www.ibm.com/cloud/architecture/architectures/edge-computing.

[24] *What is edge computing? | Cloudflare.* [Online]. Available: https://www.cloudflare.com/learning/serverless/glossary/what-is-edge-computing/.

[25] S. Rautmare and D. M. Bhalerao, "MySQL and NoSQL database comparison for IoT application," in *2016 IEEE International Conference on Advances in Computer Applications, ICACA 2016*, Institute of Electrical and Electronics Engineers Inc., Mar. 2017, pp. 235–238, ISBN: 9781509037704. DOI: 10.1109/ICACA.2016.7887957.

[26] S. Di Martino, L. Fiadone, A. Peron, V. N. Vitale, and A. Riccabone, "Industrial Internet of Things: Persistence for Time Series with NoSQL Databases," in *Proceedings - 2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2019*, Institute of Electrical and Electronics Engineers Inc., Jun. 2019, pp. 340–345, ISBN: 9781728106762. DOI: 10.1109/WETICE.2019.00076.

[27] *SIMATIC IOT2000 | SIMATIC IOT gateways | Siemens Global.* [Online]. Available: https://new.siemens.com/global/en/products/automation/pc-based/iot-gateways/iot2000.html.

[28] *KNX IP BAOS 774.* [Online]. Available: https://www.weinzierl.de/index.php/en/all-knx/knx-devices-en/knx-ip-baos-774-en?gclid=Cj0KCQjw_8mHBhClARIsABfFgpiW8woyOVFwgl2cSqx0V-tluwwsbpK8wPsJ0aZI1LlTI6fy5-PUnUYaAlZYEALw_wcB.

[29] *Dali Gateway 64.* [Online]. Available: https://www.mdt.de/EN_Dali_Gateway.html.

[30] *Switch Actuators with current measurement.* [Online]. Available: https://www.mdt.de/EN_Switch_Actuators_AMS_AMI.html.

[31] "Kamstrup 382 Datasheet," Tech. Rep. [Online]. Available: https://products.kamstrup.com/documents/512b6c0158f0e.pdf.

[32] "Swiss Garde 300 D Datasheet," Tech. Rep. [Online]. Available: https://descargas.futurasmus-knxgroup.org/doc/en/zublin/10633/sg300_d_presence_knx_up.pdf.

[33] *ACTinBOX CLASSIC HYBRID Multifunction Actuator 4Out10A 6In.* [Online]. Available: https://www.zennio.com/products/actuators/actinboxclassichybrid.

[34] *Temperature Controller.* [Online]. Available: https://www.mdt.de/EN_Temperature_Controllers.html.

[35] "EM340 Datasheet," Tech. Rep. [Online]. Available: http://productselection.net/Pdf/UK/EM340DS.pdf.

[36] *Eclipse Mosquitto.* [Online]. Available: https://mosquitto.org/.

[37] *Ryu SDN Framework.* [Online]. Available: https://ryu-sdn.org/.

[38] *Edgecore Networks Products | AS4610-54T.* [Online]. Available: https://www.edge-core.com/productsInfo.php?cls=&cls2=&cls3=46&id=21.

[39] *PICOS SDN Edition - Pica8.* [Online]. Available: https://www.pica8.com/product/#sdn-edition.

[40] *Install Docker Engine | Docker Documentation.* [Online]. Available: https://docs.docker.com/engine/install/.

[41] *Install Docker Compose | Docker Documentation.* [Online]. Available: https://docs.docker.com/compose/install/.

[42] *The Official YAML Web Site.* [Online]. Available: https://yaml.org/.

[43] *edgexfoundry/docker-edgex-ui-go.* [Online]. Available: https://hub.docker.com/r/edgexfoundry/docker-edgex-ui-go.

[44] *TOML: Tom's Obvious Minimal Language.* [Online]. Available: https://toml.io/en/.

[45] *Manage subscriptions in InfluxDB | InfluxDB OSS 1.8 Documentation.* [Online]. Available: https://docs.influxdata.com/influxdb/v1.8/administration/subscription-management/.

[46] *Grafana Docker image.* [Online]. Available: https://registry.hub.docker.com/r/grafana/grafana.

[47] *edgexfoundry/docker-device-modbus-go.* [Online]. Available: https://hub.docker.com/r/edgexfoundry/docker-device-modbus-go.

[48] “EM300 Series and ET300 Series COMMUNICATION PROTOCOL,” Tech. Rep. [Online]. Available: https://www.enika.eu/data/files/produkty/energy%20m/CP/3-phase%20Modbus%20serial%20protocol%20EM300%20and%20ET300-rev_2_13.pdf.

[49] *ESP32-DevKitC Board I Espressif.* [Online]. Available: https://www.espressif.com/en/products/devkits/esp32-devkitc.

[50] “TC74 Datasheet,” Tech. Rep. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/21462D.pdf.

[51] “IRF840 Power MOSFET transistor,” Tech. Rep. [Online]. Available: www.vishay.com/doc?91000.

[52] *ESP-IDF Programming Guide Documentation.* [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/index.html.

[53] *esp-idf/examples/protocols/mqtt/tcp at master · GitHub.* [Online]. Available: https://github.com/espressif/esp-idf/tree/master/examples/protocols/mqtt/tcp.

[54] *edgexfoundry/docker-device-mqtt-go.* [Online]. Available: https://hub.docker.com/r/edgexfoundry/docker-device-mqtt-go.

[55] *eclipse-mosquitto.* [Online]. Available: https://hub.docker.com/_/eclipse-mosquitto.

[56] *ethtool - Linux man page.* [Online]. Available: https://linux.die.net/man/8/ethtool.

[57] *ryu/simple_switch_13.py | GitHub.* [Online]. Available: https://github.com/faucetsdn/ryu/blob/master/ryu/app/simple_switch_13.py.

[58] *Eclipse Paho | The Eclipse Foundation.* [Online]. Available: https://www.eclipse.org/paho/.

[59] *packETH.* [Online]. Available: http://packeth.sourceforge.net/packeth/Home.html.