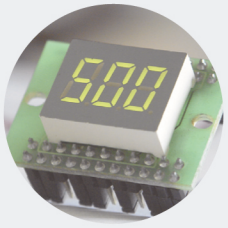
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES






Comunicação de Dados Baseada em MQTT para Sistemas Embebidos

ÁLVARO CARVALHO COHEN LOPES CARDOSO
julho de 2024

POLITÉCNICO
DO PORTO

Politécnico do Porto
Instituto Superior de Engenharia do Porto

MQTT-Based Data Communication for Embedded Applications

Álvaro Carvalho Cohen Lopes Cardoso

Master in Electrical and Computer Engineering
Specialization Area of Automation and Systems

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto

July, 2024

This dissertation partially satisfies the requirements of the Thesis/Dissertation course of the program Master in Electrical and Computer Engineering, Specialization Area of Automation and Systems.

Candidate: Álvaro Carvalho Cohen Lopes Cardoso, No. 1190341, 1190341@isep.ipp.pt

Scientific Guidance: Paula Viana, pmv@isep.ipp.pt

Company: Stratio, Lda.

Advisor: Rafael Chelim, rafaelchelim@stratioautomotive.com

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
Rua Dr. António Bernardino de Almeida, 431, 4200–072 Porto

July, 2024

To Emília and Mariana, the two women of my life.

Acknowledgements

Writing a master's thesis is a monumental task that requires countless hours of research, numerous brainstorming sessions, and, admittedly, a fair share of headaches. Thankfully I was lucky enough to have the support of a large number of wonderful people who helped me through this endeavor.

First and foremost, I would like to express my gratitude to my supervisor, Rafael Chelim. His tireless support, constant availability, and guidance were very important in the completion of this project. I would also like to extend my heartfelt thanks to Professor Paula Viana for her advice and unwavering support.

To my mother, Emília, your patience and willingness to help in any way possible meant the world to me. To my girl, Mariana, your encouraging words and the brilliant idea for the box plot graph were amazing contributions to this thesis.

Lastly, I would like to extend my sincere thanks to the entire Stratio team. Each one of you made me feel welcomed and supported throughout this journey. Your positive energy and camaraderie were a source of great comfort and motivation, and I have nothing but the highest praise for all of you.

Thank you all!

Abstract

The rise of the Internet of Things (IoT) brings an increase in the number of devices that are connected to the internet, which results in a greater need for efficient, reliable, and fast communication protocols.

This thesis explores the development and implementation of an MQTT-based communication system for embedded applications. It embraces a clear challenge: an embedded system that currently only prints messages to a serial line and does not have network transmission capabilities. By implementing MQTT, this system can communicate with other devices, which has a big impact on the workflow, as new and easier ways of debugging possible errors, understanding the state of the system, and even publishing information to the system are unlocked. This firmware was developed in a FreeRTOS environment.

In a structured way, this project began by testing the chosen hardware, with the goal of understanding its ability to perform the intended tasks. This was achieved by building a small Proof of Concept (PoC) that sends simple MQTT messages through the network. Several software validation strategies were implemented to ensure a robust code.

Custom MQTT brokers were also developed in efforts to achieve optimal performance and good compatibility with the embedded environment.

The results gathered from analyzing the message transmission times, demonstrate that balancing speed, consistency, and network congestion is a crucial step toward having a more robust and efficient system.

Keywords: MQTT, FreeRTOS, Communication.

Resumo

A ascensão da Internet das Coisas (IoT) traz um aumento no número de dispositivos conectados à internet, resultando numa maior necessidade de protocolos de comunicação eficientes, confiáveis e rápidos.

Esta tese explora o desenvolvimento e a implementação de um sistema de comunicação baseado em MQTT para aplicações embebidas. Enfrenta um desafio claro: um sistema embebido que atualmente apenas imprime mensagens numa linha série e não possui capacidades de transmissão de rede. Ao implementar o MQTT, este sistema pode comunicar com outros dispositivos, o que tem um grande impacto no fluxo de trabalho, desbloqueando novas e mais fáceis maneiras de apurar possíveis erros, entender o estado do sistema e até mesmo publicar informações no sistema. Este firmware foi desenvolvido num ambiente FreeRTOS, aproveitando a sua extensa documentação.

De forma estruturada, este projeto começou por testar o hardware escolhido, com o objetivo de entender a sua capacidade de realizar as tarefas pretendidas. Isso foi alcançado construindo uma prova de conceito que envia mensagens MQTT simples através da rede. Várias estratégias de validação de software foram implementadas para garantir um código robusto.

Brokers de MQTT personalizados também foram desenvolvidos, num esforço para alcançar um bom desempenho e boa compatibilidade com o ambiente embebido.

Os resultados obtidos a partir da análise dos tempos de transmissão das mensagens demonstram que equilibrar velocidade, consistência e congestionamento da rede é um passo crucial para ter um sistema mais robusto e eficiente.

Palavras-Chave: MQTT, FreeRTOS, Comunicação.

Contents

List of Figures

List of Tables

Listings

List of Acronyms

APT	Advanced Persistent Threat
ARP	Address Resolution Protocol
AWS	Amazon Web Services
CA	Authority Certificates
CC	Client Certificates
CD	Continuous Delivery
CI	Continuous Integration
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
CSV	Comma-Separated Values
DDoS	Distributed Denial of Service
DPA	Differential Power Analyses
GNSS	Global Navigation Satellite Systems
GPOS	General Purpose Operating System
GPRS	General Packet Radio Services
GPS	Global Positioning System
GSM	Global System for Mobile Comunication
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-Process Communication
JSON	JavaScript Object Notation

LAN	Local Area Network
LoRa	Long Range
LPWAN	Low-Power Wide-Area Network
LTE	Long Term Evolution
LTE-M	Long Term Evolution (M2M)
M2M	Machine to Machine
MAC	Media Access Control
MitM	Man-in-the-Middle
ML	Machine Learning
MQTT	Message Queuing Telemetry Transport
NB-IoT	Narrowband IoT
OS	Operating System
PoC	Proof of Concept
QoS	Quality of Service
RSSI	Received Signal Strength Indicator
RTOS	Real-Time Operating System
SCA	Side-Channel Attack
SIL	Software In the Loop
SPA	Simple Power Analyses
SSL	Secure Socket Layer
STD	Standard Deviation
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VPN	Virtual Private Network
WSN	Wireless Sensor Network
XML	Extensible Markup Language

Chapter 1

Introduction

The data transmitted via the Internet has been growing steadily throughout the years [1]. This is caused by an exponential growth in the number of devices that are connected to the internet, which in turn, significantly increases the amount of generated, transmitted, and processed data. Hence, it is important to have robust and adaptable infrastructure, from modems to communication protocols, to efficiently handle the massive data flows and ensure connectivity.

Internet of Things (IoT) refers to a network of different devices, that are equipped with sensors and connected to the internet, communicating and exchanging information with each other. It covers a wide range of applications, from intelligent domestic devices to road vehicles that can transmit important data. This project is bound to be integrated into an environment similar to the latter, as it aims to create the necessary conditions to enable an embedded system within a vehicle to transmit information to distant clients.

1.1 Contextualization

This project is designed to be integrated within a wider system developed by Stratio. They are a Portuguese company, who are at the forefront of predictive maintenance technologies. Their focus is on developing databoxes [2] that are installed on-road vehicles (mainly buses) to gather information about the vehicle's state - for example, the condition of the brakes, the engine, and so on. The objective is to check for any faults before they even occur preemptively. These Databoxes are permanently

connected to the network so that the clients can have all of this information remotely. Each Databox can be considered as an embedded system that works repeatedly to gather information about the vehicle's condition and send it to a server, using the HTTP protocol. These databoxes also have internal logging systems, which generate logs about the state of the device (for example, there are logs that denote when the device connects to the internet or if any task had a problem). These logs are essential to give information to the developers about how the device is working or to check for any error logs in case it malfunctions.

The main goal of the project is to create a way to transmit these logs to a server, via the internet, to make sure developers have ways to diagnose any potential issues tampering with the system. It is worth mentioning that the already existing connection to the network is not configured to send these logs, being solely responsible for transmitting the data that the databox gathers about the vehicle's state. This is because, the HTTP communication is meant to provide information to Stratio's clients, and the logs, which are only accessible via a hardware connection capable of reading the serial line, are supposed to be private (not available to the clients). Therefore, this project aims to give the system a whole new way of communicating. The strive for efficiency, reliability, and adaptability was paramount to the choice of technologies such as Message Queuing Telemetry Transport (a reputed communication protocol) and FreeRTOS (a well-known and frequently used real-time operating system). Figure 1.1 illustrates a representation of the intended system for this project, showcasing several vehicles equipped with Stratio's databoxes, transmitting the generated logs.

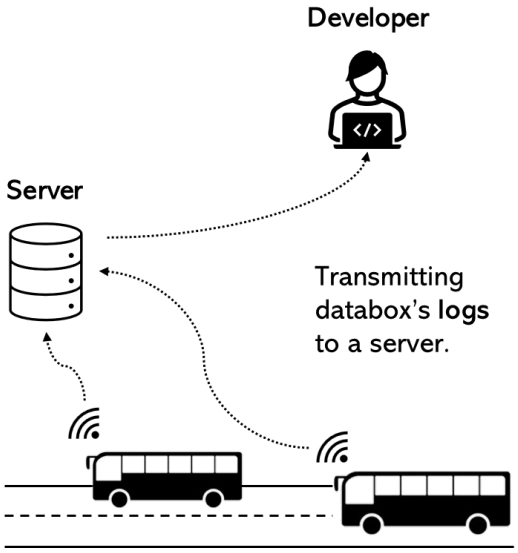


Figure 1.1: General Project Overview.

By overcoming the challenges inherent to the transmission of messages, this project aims to not only improve the workflow but also to provide a robust and

scalable solution for future developments. As technical aspects and implementation strategies are approached, the following chapters delve deeper into the adopted methodologies and the results achieved, which show the impact of this project in a real-world implementation.

1.2 Stratio

Stratio [2] is a Portuguese company dedicated to building around the goal of achieving a zero-downtime future, where transportation is more reliable, sustainable, and accessible to all.

They provide transport companies with mechanisms that allow real-time visibility over their vehicles. What before was simple guesswork, now is a source of operational intelligence. Using data and AI, Stratio offers predictive insights that help their clients identify faults before they occur, as well as minimize downtime (and the associated costs). This maximizes vehicle availability and reduces unexpected service interruptions, which is good for both the transport companies and their clients.

Stratio's approach contributes to accessible and sustainable transport.

1.3 Problem Formulation

Each Stratio databox generates logs that detail how the system is doing, which are important for developers to understand the state of the system. Before the implementation of this project, these messages were only transmitted via the serial port. This limitation impacted negatively the efficiency of the workflow, as it made it harder to analyze or debug possible problems that occurred when the developers did not have physical access to the equipment. This project aims to introduce a system by which an MQTT connection is established between the databox and a server, which would allow these logs to be transmitted, and easily accessed, even when developers do not have physical access to the device. This brings many benefits to the system, as it improves workflow, all the while unlocking new ways to interact with the system.

1.3.1 Objectives

Multiple objectives can be outlined for this project. These are:

- **Development of a Communication System**: Implement a way for the embedded system to communicate with the network and publish its logs.

- **Development of a Pilot Project**: Make sure that the chosen hardware can handle such communications, while also familiarizing with the involved technologies. This can be achieved through a Proof of Concept (PoC).
- **Software Validation**: Use systems like a Continuous Integration/Continuous Delivery (CI/CD) pipeline to validate the software and assure the quality of the final project.
- **Obtaining Results**: After the project is working, it is important to gather information about its performance, and how changes to the project affect these results.
- **Study of IoT Security Liabilities and LPWAN technologies**: Even though the implementation does not heavily focus on these aspects, they serve as support for Stratio's future work and future improvements for this project.

1.3.2 Expected Results

The main expected result of the project is the improvement of the workflow around the embedded system. Monitoring and debugging are expected to be vastly improved from the introduction and implementation of the project.

In more specific terms, it is expected that the chosen hardware is capable of performing such tasks and that the communication happens as smoothly as possible.

The outcome should be robust software that answers the problems enumerated, and that is simple, yet efficient.

1.4 Document Structure

This document is organized as follows:

- **Chapter 2**: This chapter delves into some of the most relevant technologies and methods related to the IoT and that might be relevant to the understanding of the software/hardware choices for the project. Security in IoT, message protocols and Real-Time Operating System (RTOS) are a few of the topics discussed.
- **Chapter 3**: This chapter describes the implementation strategies of the project itself. It covers the setup of the broker infrastructure, configuration, deployment, and interactions with the broker both as a "localhost" and via a dedicated server. Software architecture is the main focus.
- **Chapter 4**: This chapter presents the results gathered from the tests performed. It also includes a detailed analysis and explanation of said results.

- **Chapter 5**: This final chapter focuses on the conclusions drawn from the entire project as well as an assessment of future improvements that could be achieved.

Chapter 2

State of The Art

The focus of this State of the Art is to review all the involved technologies, assessing how they are being used in different industries.

The first subchapter delves into what the IoT is and its inherent security issues. Crucial security improvements are discussed and examples of real-life applications are studied. After, the focus changes to the messaging protocols. Of all the ones available, 3 of them are picked to analyze and study the differences that make each one unique and useful for different tasks. An overview of RTOSs follows, depicting three of the main ones currently used. A comparative analysis is done to understand which ones fit which type of application. The last topic of this state of the art is Wireless Connectivity, highlighting its crucial role in modern communication, with Low-Power Wide-Area Network (LPWAN) technologies catering to diverse IoT requirements and modems, facilitating efficient and reliable connectivity.

2.1 Internet of Things

In its most basic form, IoT can be described as the concept of connecting everyday objects to the internet, allowing them to communicate with each other and with centralized systems. By the end of the current year (2024), an estimation of 19.2 billion connected IoT devices worldwide is projected [1], highlighting the growth these technologies are experiencing.

2.1.1 IoT Security

The globalization of IoT technology comes with an increased vulnerability of the devices and the network itself, as there are a great number of gateways from where to perform malicious activities. Malicious internal and external attackers and corrupt manufacturers are some of the threat agents that exist in the IoT world.

An attack can be defined as an attempt to destroy, expose, alter, disable, steal, or gain any unauthorized access to an asset [3]. There are many different types of possible attacks in IoT, that can affect different parts of the data transmission process or even data integrity. It is, therefore, important to build a robust system that can be as immune as possible to threats of any kind.

This sub-chapter focuses on analyzing threats and attacks that have been reported by various entities, while also understanding the reasons they succeeded and possible solutions to stop them from happening again.

As mentioned, several different types of attacks can happen in IoT, and some of the most common are [4]:

- **DDoS** – Distributed Denial of Service
- **APT** – Advanced Persistent Threat
- **MitM** –Man-in-the-Middle
- **SCA** – Side-Channel Attack

Distributed Denial of Service (DDoS)

A Denial of Service attack is the attempted prevention of the legitimate use of a service [5]. Typically, the perpetrator sends several packages that overload some key resource, therefore impeding the victim from using said service [6]. When these attempts are implemented as a coordinated action from several different locations, a Distributed DoS (DDoS) attack is being carried out.

One of the biggest attacks that used the DDoS approach was the Mirai Botnet in 2016. It targeted essentially IoT devices, such as printers and routers, and it was able to infect over 600,000 devices [7].

The Mirai Botnet attack started by trying to scan several thousand IPv4 (Internet Protocol version 4) addresses by sending them Transmission Control Protocol (TCP) probes, looking for devices that had weak or default credentials which are typically the result of users who neglect to change the default usernames and passwords that come with the devices. Once it found one, it would jump into the next step of the attack which was to force several usernames and passwords, in attempts to perform an unauthorized login. When successful, two things happened: the username and the password were stored in a server and malware was downloaded to

the victim; the now infected device would join others in Mirai's botnet[1] in attempts to launch a coordinated DDoS attack of key targets [7]. These attacks flooded the targeted websites and online services with a massive volume of traffic, making them inaccessible and causing severe disruption.

There are various ways to prevent this kind of attack and safeguard the devices connected to any network. One of the most important safety measures to have in mind is to always change the default username and password to unique and strong credentials, and regularly update their firmware to address any new vulnerabilities.

Z. Ahmed *et al* [8] propose blockchain-based protection. A structure was created to organize IP addresses of hosts in different security lists. These lists were shared between the devices through the Ethereum blockchain (benefiting from its extremely secure features, like cryptographic hashing and decentralized mechanisms) so that all devices were aware of possible threats. They also implemented a maximum amount of failed login attempts to try to neutralize Mirai's forced unauthorized logins.

Ruchi Vishwakarma *et al* [9] presented a honeypot-based approach that uses Machine Learning (ML) techniques for malware detection. A honeypot functions as a cybersecurity decoy. It simulates vulnerabilities and lures in attackers, capturing critical information, including IP addresses, Media Access Control (MAC) addresses, port numbers, targeted devices, and malware details. In their proposal, a honeypot system was developed and the data it collected was used to train ML software, in hopes of automating and preventing this kind of attack.

Advanced Persistent Threat (APT)

According to NIST (National Institute of Standards and Technology), an APT is defined as "*an adversary that possesses sophisticated levels of expertise and significant resources that allow it to create opportunities to achieve its objectives by using multiple attack vectors*" [10]. While DDoS aims to try to crash a certain service, APT aims to steal information and spy on the victim.

K. Huang *et al* [11] split APT attacks into 4 major phases. The *Preparedness* phase involves extensive research, aiming to identify specific targets (IoT devices or networks). This includes the gathering of potential target information as well as vulnerabilities. It ends in creating ways to gain access to the target by methods such as e-mail phishing to trick users into revealing credentials. *Infiltration* follows up, where the e-mails are delivered to the targeted individuals, and once opened, a backdoor is installed that connects the attacker and the host. *Lateral Movement* consists of installing other backdoors on other nodes in the network to propagate access through the company. Finally, *Data Exfiltration* where the attacker manages

[1] Botnet refers to a collection of internet-connected devices or computers that have been compromised by malicious software.

to extract sensitive information from the infected network. This attack can take a long time, but it is often worth it because of its careful and seamless action.

The NetTraveller group is a cyber-espionage group responsible for more than 350 high-profile victims from all over the world since 2004 [12]. Their method typically consists of phishing campaigns that target specific individuals, trying to persuade them to click on a compromised Uniform Resource Locator (URL). These URLs lead to executable files and other infected files that try to exploit Microsoft Office vulnerabilities. Kaspersky, who wrote a detailed report on the NetTraveller group, shared some mitigation information, such as indicators of compromise (network traffic red flags and C&C domains), malware names, and several cryptographic hashes [13].

Man-in-the-Middle (MitM)

A MitM attack occurs when communication between two systems (typically IoT devices) is breached. This kind of attack encompasses several different approaches but one of the most common is eavesdropping where the attacker passively listens to network communications to gain access to private information. This can be achieved, for example, with Address Resolution Protocol (ARP) cache poisoning [4].

The ARP protocol matches a given IP address to a physical machine address (MAC Address). When a node needs to send a package, it will forward an ARP request to the network containing the receiver's IP address. The node that is identified with said IP address replies with its MAC Address. However, one of the weaknesses of this protocol is the fact that it can receive ARP replies without having made a request. This opens the door for MitM attacks as, by changing the ARP cache, messages might be forwarded through a malicious node [14]. This node can then read all of the information passing through.

D. Kim *et al* [15] propose a new mechanism to prevent ARP poisoning-based MitM attacks. It is based on two different concepts: long-term IP/MAC mapping table and voting. ARP caches typically have a two-minute timeout, which means that after 2 minutes, there will have to be an ARP request to get the MAC Address. The proposed model intends to implement a long-term IP/MAC mapping table (with a default timeout of 60 min) that is used to store the addresses for much longer, reducing the need for a bigger number of requests that might receive malicious replies. If an ARP request or reply contains an IP Address that is registered in the long-term table, but whose MAC is different from the one registered in the table, several ARP requests will be sent to the registered MAC Address. If at least one replies, that means that the MAC Address present in the long-term list is still valid and the one in the ARP request or reply is likely malicious. If none replies, then the new MAC is accepted in both lists. The voting-based conflict resolution method is proposed when a new machine joins a Local Area Network (LAN) with no priorIP/MAC mapping information. It collects information from multiple hosts

about the IP/MAC mapping of a new machine and uses a voting process to determine the most reliable MAC address for the given IP address.

A. Said [16] summarizes a set of good countermeasures for MitM attacks. It mentions that the usage of protocols such as Transport Layer Security (TLS) is crucial, as it encrypts the communication channels between IoT devices and their networks, ensuring data confidentiality.

One other way to encrypt the communication flow is by using Virtual Private Network (VPN), as it creates a secure tunnel that prevents unauthorized access or interception by potential attackers [17].

Side-Channel Attack (SCA)

An SCA is based on analyzing physical information. Its objective is to steal cryptographic information, and does that by analyzing data such as execution time, power consumption, electromagnetic emission, and photonic emission from devices that are processing cryptographic algorithms. One way for attackers to measure power consumption is to put a resistor in series with the power or ground pins, and then measure the voltage drop [18].

B. Sim *et al* [19] investigated the different approaches of an SCA, detailing processes like Simple Power Analyses (SPA) and Differential Power Analyses (DPA).

The SPA approach is based on the fact that instant power consumption is directly linked to a task being carried out by the IoT device, which means that different tasks might have different instant power consumption that can then be traced back and used to steal cryptographic data. It does not require multiple power measurements over time. The double and add algorithm is used in several cryptographic contexts. In this case, an SCA attack could be fruitful as the cryptographic key can be revealed if the attacker can tell the difference in instant power consumption between the double and add actions. J. Fan *et al* [20] discern several ways of countering this threat, one of which being the creation of *dummy* operations like *dummy* point additions, that would cloud any real operations. These are decoy actions designed to mimic the power consumption patterns of actual cryptographic operations like point additions in the double and add algorithm.

DPA is a more advanced and powerful side-channel attack technique that relies on collecting multiple power consumption measurements at different time instances during cryptographic operations. It exploits the variations in power consumption that occur as the device processes the same data but at different points in time. By comparing multiple measurements, it can reveal information about sensitive data.

2.2 Messaging Protocols and Examples of Usage

Nowadays, there are several messaging protocols available to use. Messaging protocols serve as the backbone for data transfer from independent data-generating devices to the Internet, enabling further processing and analysis. Each one of them has its features and benefits. In truth, any protocol cannot deal with all possible IoT use cases, and the choice between one or the other may rest on the type of IoT system the developer intends to build, the required level of reliability needed, or performance. Therefore, developers must make an educated choice on which messaging protocol they intend to use on the project, as poor choices may result in problems later on.

Messaging protocols are present on the application layer of the TCP/IP model, as Table 2.1 shows, which is the highest layer in the stack and is responsible for providing network services to applications.

Table 2.1: TCP/IP Model.

Layer	**Protocols**
Application	HTTP, CoAP, MQTT, AmQP
Transport	TCP, UDP
Network	IP, ARP, 6LoWPAN
Data Link	Ethernet, Token Ring
Physical	Cables, connectors, modems

Despite there being several of messaging protocols available, this sub-chapter focuses on three of the most known and most used messaging protocols today:

- **MQTT** – Message Queuing Telemetry Transport
- **HTTP** – Hypertext Transfer Protocol
- **CoAP** – Constrained Application Protocol

Each one of these protocols will be reviewed and their differences in various areas will be analyzed. Real-world applications will also be a focus of this sub-chapter.

2.2.1 Message Queuing Telemetry Transport (MQTT)

MQTT was introduced in 1999 by Andy Stanford-Clark of IBM, and Arlen Nipper of Arcom [21]. It was designed to be a lightweight protocol that could be used to connect devices with limited network bandwidth or processing power. It is a publish/subscribe messaging protocol, based on a client-server model. The server, commonly known as the "broker", receives several messages from the clients. These messages can be one of two types: publish or subscribe. Figure 2.1 shows the relation

between the clients and the broker. As shown, both publish and subscribe messages need to be linked to a topic which is a hierarchical path that is used to organize messages and to make it easy for subscribers to find the messages they are interested in.

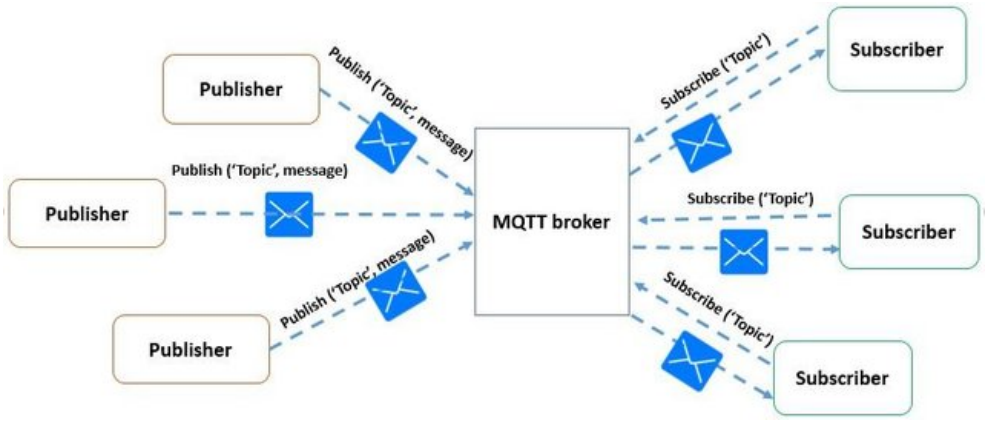


Figure 2.1: MQTT Scheme [22].

MQTT allows 3 levels of Quality of Service (QoS), them being: Message is delivered at most once (QoS 0), at least once (QoS 1), and exactly once (QoS 2). QoS levels must be properly evaluated to meet the requirement of data acquisition. To bridge the theoretical understanding of MQTT's QoS levels with practical implications, Y. Shinji et K. Shiomoto [23] propose an MQTT-based approach to try to prevent river floods while analyzing which QoS level would work the best. After several tests of both packet sizes and latency, it was determined that QoS 1 was the go-to QoS level, as QoS 2 made the packet a lot bigger (almost 2 times as big as the one in QoS 1) and had a considerably higher latency.

RK Kodali *et al* [24] developed a system incorporating MQTT and Amazon Web Services (AWS) that can detect and report in real-time air quality and temperature levels. An ESP32-based development board (named M5Stack) was used, acting as an MQTT Client, as well as different gas and temperature sensors. The M5Stack was equipped with Mongoose RTOS. JavaScript code allows the M5Stack to read the sensor values and send them to an MQTT Broker provided by AWS IoT Core. The data is then interpreted and acted upon, sending texts to users' phones reporting the air temperature and quality.

S. Jaloudi [25] studied the MQTT messaging protocol applied to smart cities. For this, a TCP/IP-based network model was proposed, where the application layer relies on JavaScript Object Notation (JSON) formatted information being sent through MQTT. MQTT uses UTF-8 encoding format, as its payload must consist of binary data [26]. However, while UTF-8 ensures consistent data representation (binary),

it lacks a standardized format for defining object structures, which leads to interoperability[2] concerns. JSON is used to address this problem in this study. It is a lightweight format for storing and transporting data and uses text-based ASCII representation. It has a compact file size, easy to use, and can be read by humans or machines. The idea behind the network proposed is to have several sensors (clients) connected to an access point via Wi-Fi by using an ESP8266 (Wi-Fi Module). The acess point is connected to an Ethernet switch that is also linked to a Raspberry Pi3, which acts as an MQTT server. A few MQTT clients, equipped with user interfaces, are used to monitor the sensor information. This study acts as a base example of how to implement MQTT in smart cities on a larger scale.

MQTT and MQTT brokers often provide additional layers of protection, which are encryption and authentication. Encryption is related to the use of protocols such as TLS or Secure Socket Layer (SSL) to convert data into a secure format so that it remains confidential and unreadable to unauthorized parties. Authentication can be referred to as the process of ensuring that only authorized clients can access the broker. This can be achieved with usernames and passwords, as well as Client Certificates (CC) and Authority Certificates (CA). A CC authenticates a device to a broker, ensuring secure online interactions, while a CA is issued to hostnames, validating the authenticity of the server, and ensuring that there are no eavesdrops in the communication [27].

Mosquitto [28], for example, is a free MQTT broker typically used for testing. It allows for different ways of connecting with different levels of security, such as:

- Unencrypted, unauthenticated
- Unencrypted, authenticated
- Encrypted, unauthenticated
- Encrypted, client certificate required
- Others

The first two, vary in the way that one needs a username and a password to be provided (authenticated), while the other doesn't (unauthenticated). The third doesn't require a username and a password, but it does however require a CA. The last one is the most secure of them all, requiring a CA, and a CC to function.

[2]Interoperability can be viewed as the ability of different devices and applications to communicate with each other and exchange data seamlessly.

It is worth noting that in S. Jaloudi's study [25], mentioned above, a secure SSL/TLS[3] connection by using port 8883 (the default MQTT TLS port) is mentioned. It does, however, come at a cost. TLS makes MQTT perform worse, especially on the TLS Handshake[4]. After all clients have been connected, Central Processing Unit (CPU) usage is still greater when using TLS, but the difference is small [30]. Figure 2.2 shows the CPU usage comparison between MQTT with and without TLS.

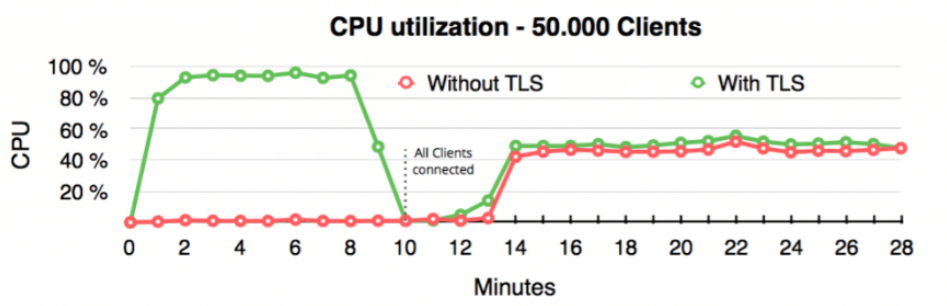


Figure 2.2: Comparison Between MQTT With and Without TLS [30].

Despite the negatively impacted performance, TLS is very useful for the prevention of MitM attacks, as it creates a secure connection between a client and a server.

Xueqin Amy Liu *et al* [31] proposes the use of Flatbuffers to format data, instead of JSON, and proceeds to test and compare the performance of various data formats, such as JSON, Extensible Markup Language (XML), and Comma-Separated Values (CSV), in an MQTT communication context. It is claimed that, despite being the most commonly used data format in IoT, JSON can have lengthy processing times. Three parameters were analyzed: payload, transfer rate (throughput), and latency. The conclusions showed that JSON performed among the worst in terms of processing latency and processing throughput, as well as having the second-highest payload, but this is compensated by performing among the best in delivery latency and throughput. Flatbuffers didn't prove to be a suitable replacement.

2.2.2 Hypertext Transfer Protocol (HTTP)

HTTP [32] is a widely used communication protocol for data transfer. It was first proposed by Tim Berners-Lee in 1989. It is based upon a Request-Response model, where the client sends a request to the server, and the server sends a response back

[3]The terms SSL and TLS are often used interchangeably. TLS is the successor to SSL, and is more secure, having been updated to address known vulnerabilities in SSL [29].

[4]Used to authenticate the server to the client, agree on a shared encryption key, and establish a secure communication channel.

to the client. The request contains various information such as the HTTP method (GET, POST, DELETE, among others), HTTP request headers and the HTTP body. Figure 2.3 shows the HTTP request-response scheme.

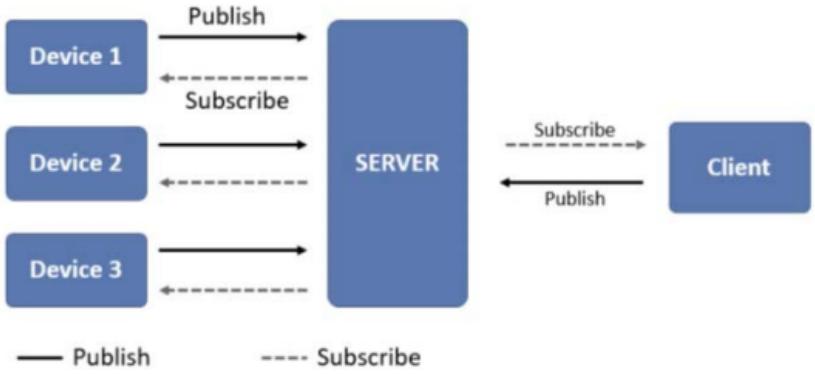


Figure 2.3: HTTP Request-Response Scheme [33].

While it is widely accepted as a good candidate for Web usage, it must be noted that HTTP was designed without consideration of resource-constrained devices [34]. Its higher overhead, its difficulty in dealing with non-continuous connection environments, and the fact that it is half-duplex[5], make it a poor choice for IoT. As a result, alternative protocols such as CoAP and MQTT have been developed to address the specific challenges posed by resource limitations and network constraints. An example of the superiority of these alternative protocols, in these situations, is the system developed by Nicușor-Mirel Drogeanu *et al* [35], that collects operating parameters from a vehicle and sends them to a web server. This was done to develop a vehicle monitoring system. An Arduino development board and two Arduino shields were used to gather vehicle data, and parameters like Global Positioning System (GPS) location and engine temperature were sent to the web server by MQTT. It had been previously tried to achieve this communication using HTTP but the delay was far too great. The recorded MQTT delay was approximately 98.4% lower than the delay experienced with the HTTP protocol for data transmission.

Constrained Application Protocol (CoAP)

CoAP [36] is a messaging protocol that was standardized by the IETF (Internet Engineering Task Force) in 2014. It stands out when compared to the other protocols mentioned, as it works mainly over User Datagram Protocol (UDP). Designed specifically for constrained devices and networks within the context of the IoT and Machine to Machine (M2M) communications, CoAP is, fundamentally, a Request-Response model that makes use of GET, PUT, POST, and DELETE methods much like how they are employed in HTTP. The client sends CoAP requests to a server

[5]Half-duplex means that communication can occur in both directions, but not simultaneously.

and waits for a response from the latter. Unlike HTTP, which works over TCP, CoAP messages are used for asynchronous exchange of requests and responses instead of relying on a pre-established connection for their transmission. Figure 2.4 shows the structure of a CoAP communication.

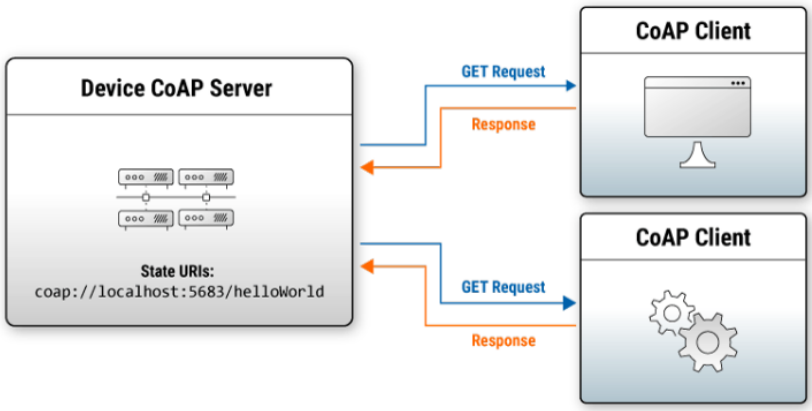


Figure 2.4: CoAP Request-Response Structure [37].

One of the main differences between CoAP and MQTT is the environment in which it should be used. While in MQTT devices act as nodes that independently exchange info with other devices over the internet, CoAP is mainly used for communication between devices in the same low-power and constrained network [38].

Di Sciascio *et al* [39] studied a way to use a CoAP-based Wireless Sensor Network (WSN) for connecting and monitoring medical sensors. In this implementation, each sensor works as a server node to which a client can establish a connection and perform actions such as POST, GET or Observe. The test bed for the WSN encompasses mainly healthcare-related systems, such as an EKG gauge and a pulse oximeter. After the sensor's measurements are completed, the data is transferred to a Telos ultra-low power wireless module over a serial line. This module will be the one to act as a server. Californium [40] is used as a client framework in Java to interact with the CoAP-based server. The monitoring system on the web browser side, which will receive JSON formatted data, is done using the Copper CoAP user agent for Firefox. It was concluded that a big advantage of this system is that it can make use of the TCP/IP model, making it easy to use the conventional internet to gather the data.

Another study made by Michele Rossi *et al* [41] intended to build a Web Services system for IoT applications, but this time the data would be formatted in XML. The CoAP module includes both client and server functionalities that efficiently handle session data, optimizing memory usage. It can, concurrently, handle up to x transactions, where x is user-defined, offering flexibility. Communication is possible

through two interfaces, CoAPClient and CoAPServer. The former facilitates sending requests to CoAP endpoints and managing responses, while the latter provides server capabilities, allowing external components to serve resources. The lightweight client/server architecture of CoAP makes it easy to deploy constrained Web services on IoT nodes, supporting M2M interactions and multiple Web servers without imposing heavy computational burdens. The experimental results are shown in Figure 2.5.

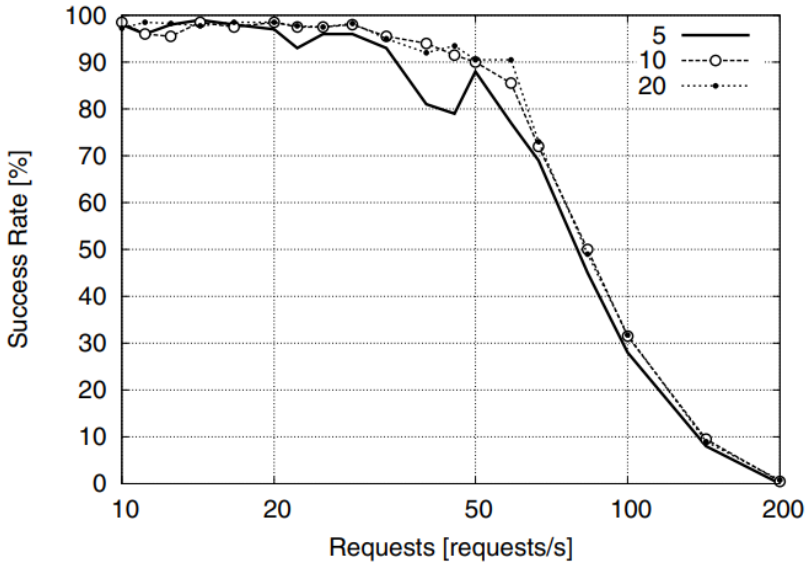


Figure 2.5: CoAP Success Rate vs. Request Rate, for different numbers of CoAP servers [41].

The percentage of successfully handled requests was measured against the client node's request rate while using different numbers of servers (5, 10 and 20). The findings showed that the likelihood of success was not significantly influenced by the quantity of servers, which showcases the scalability of the CoAP implementation. Although very high request rates (exceeding 60 requests/second) did impact access performance, it was attributed to inherent limitations in the 6LoWPAN library, used in the network layer. Overall, the experiment demonstrated that the CoAP implementation scales effectively with multiple server instances without causing a significant decline in performance.

2.2.3 Messaging Protocols Performance Comparison

As discussed before, these protocols have multiple differences between them, making them more suitable for different objectives. In this section, the three messaging protocols will be compared based on an extensive literature research. A few parameters

are approached in this comparative analysis and the objective is to have a baseline to understand what protocol should be picked in a given situation.

A study, by Dinesh Thangavel *et al* [42], on the performance of MQTT and CoAP in similar conditions was published in 2014. It sought to create a common middleware to fairly test the capabilities of both these protocols in terms of end-to-end delays and bandwidth consumption. To make this comparison as fair as possible, both protocols were set up to use a Publish-Subscribe architecture (instead of CoAP using a Request-Response architecture). The delay that each message takes to be published and received by the subscriber was recorded and Wireshark was used to analyze how many bytes were transferred. As the authors mention, one thing that must be taken into account when measuring this delay is packet loss. When a packet is lost, it should eventually be re-sent. This will consume more time and should be considered. One way to make this fair is to have both protocols implement similar levels of QoS. As reviewed previously, one of CoAP's available QoS settings is "Confirmable Messages". This setting happens to be very similar to MQTT's QoS1 in that they both provide a form of acknowledgment for message delivery. Figure 2.6 shows an analysis of the average delay for different loss rates between CoAP and MQTT. It has been generated from data from the study.

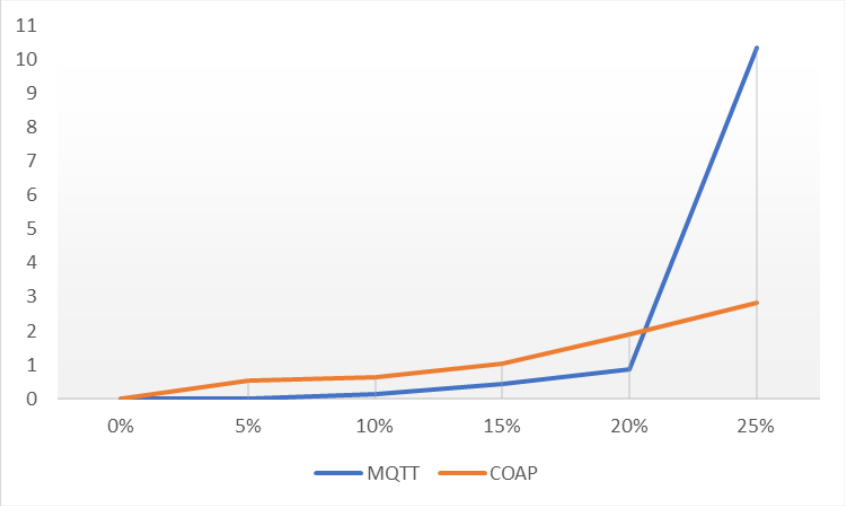


Figure 2.6: MQTT vs CoAP delay [42].

From the plot, one concludes that when the loss rate is under 20%, CoAP messages have a bigger delay than MQTT, which is faster. However, when the loss rate reaches 25%, this inverts, and MQTT registered a delay significantly higher than its counterpart. According to the authors, this has to do with the fact that MQTT's TCP re-transmission overheads are much larger than the UDP overheads. This causes a bigger influx of data to be sent and a bigger delay.

Nitin Naik's heavily referenced study [43], provides a valuable perspective on the distinctions between the three protocols under consideration. Unlike the former

study, Naik's work deliberately excludes the re-transmission scenario, simplifying the data analysis and enhancing the clarity of the findings. By narrowing the focus, the aim is to offer a more straightforward and digestible examination of the protocols' performance characteristics.

The first aspect to be reviewed is precisely the message size and overhead. As seen in the previous study, message overhead is an important factor to take into account when considering performance. MQTT, HTTP, and CoAP all have different overhead sizes which will impact the time taken to transmit messages. MQTT has an overhead of 2 bytes, the lowest overhead of the three. CoAP comes next with a 4-byte overhead. The last is HTTP which is the heavier protocol, as it was originally intended for the Web and not IoT. Despite this, in reality, the lightest protocol is CoAP, as it benefits from the fact that it uses UDP (which has a minimum overhead of 8 bytes) instead of TCP (which has a minimum overhead of 20 bytes). For the reasons mentioned before it is considered that the protocol with the smallest overhead and message size is CoAP, while the biggest belongs to HTTP.

Power consumption is also a big factor in these protocol's performance. Naik's study places CoAP as the one that consumes the lowest power and requires the least resources, followed by MQTT and HTTP. Stephen Nicholas [44] performed various tests comparing the power consumption of a long-polling HTTP connection against an MQTT connection with SSL. It was discovered that, concerning power consumption, MQTT performed much better than HTTP in every metric except the establishment of a connection. This is because, while for MQTT a raw connection is established, the certificate is exchanged and then there is a flow of additional information (like the client ID), for the HTTP all that is done is the opening of a connection and the exchange of certificates. However, MQTT only has to do this once, while HTTP needs to perform this task every time it needs to reconnect. This is why MQTT doesn't need as much power. Also, according to Walter Colitti *et al* [45], CoAP power consumption is drastically lower than HTTP.

Lastly, bandwidth and latency are also two other aspects to take into account. Niki's study describes CoAP as the one with the least latency and bandwidth while HTTP is the one with the highest. As both MQTT and HTTP use TCP, the latency gets higher. This is because TCP requires a reliable connection between server and recipient, which can slow down data transfer. UDP is a connectionless protocol and, therefore much quicker. Going back to Dinesh Thangavel's *et al* study [42], CoAP's low bandwidth was also proven. As shown in Figure 2.7 MQTT's average data transferred per message is always higher (no matter the QoS used) than CoAP's. One of the core reasons for this is, as mentioned before, the overhead size difference.

The studies presented offer valuable insights into the performance of messaging protocols in various scenarios, and CoAP has demonstrated advantages in message size, power consumption, and latency, especially in situations with higher packet

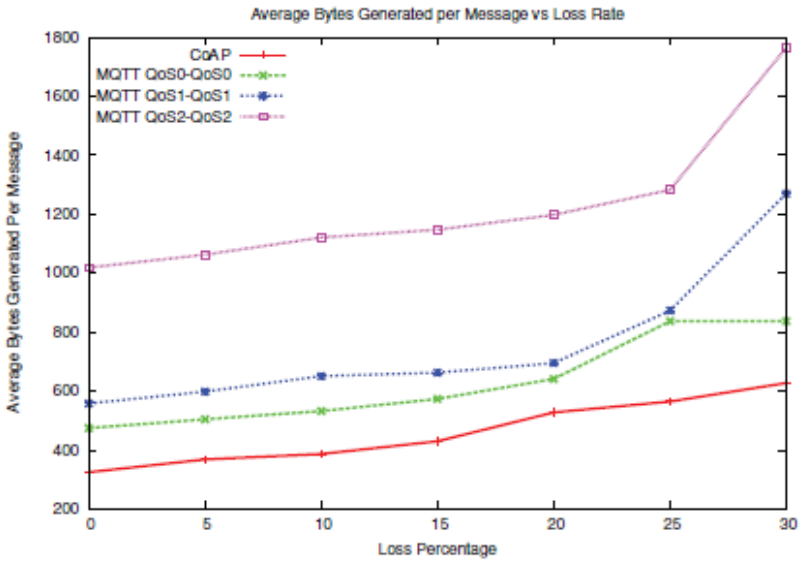


Figure 2.7: MQTT vs CoAP bandwidth [42].

loss rates.

While MQTT also performed well in many aspects, it is important to note that the choice between CoAP and MQTT depends on specific application requirements. CoAP may be more suitable for resource-constrained environments that prioritize low-latency communication, while MQTT might be more suitable for scenarios that require reliability and a publish/subscribe model.

HTTP stands as a no better choice than any of the other two, which is understandable if take into account that it wasn't primarily designed for this scenario.

CoAP and MQTT are both good choices for messaging protocols in IoT due to their efficiency in handling constrained environments. The decision between them should be based on the specific needs and constraints of the IoT application at hand.

2.3 MQTT Brokers

There are several MQTT brokers to choose from, as well as various studies analyzing their differences in performance. A study conducted by Fabio Kon [46] covers a two-part comparison of some of the available brokers. It includes a qualitative and a quantitative comparison. The former analyses factors such as documentation completeness, configuration options provided, installation experience, and so on, while the latter delves into their performance. It is worth noting that the quantitative evaluation was done just on the three highest-ranked brokers from the qualitative evaluation.

The graph in Figure 2.8 shows a summary of the results of the qualitative analysis.

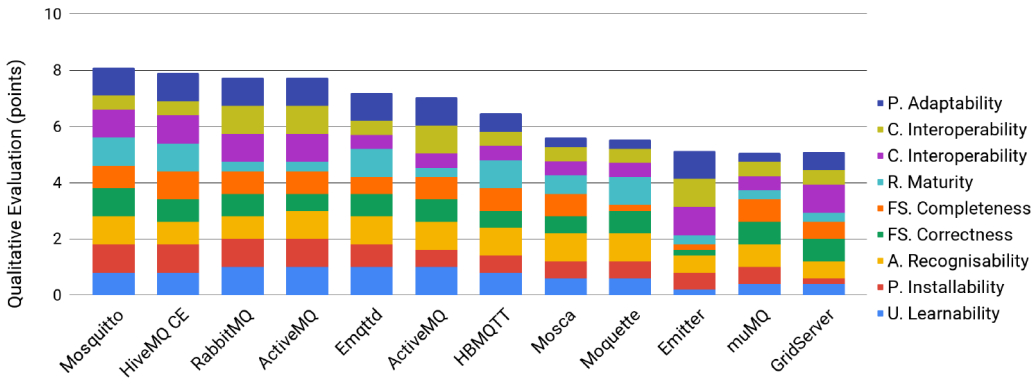


Figure 2.8: Qualitative Analysis of 12 Different Brokers [46].

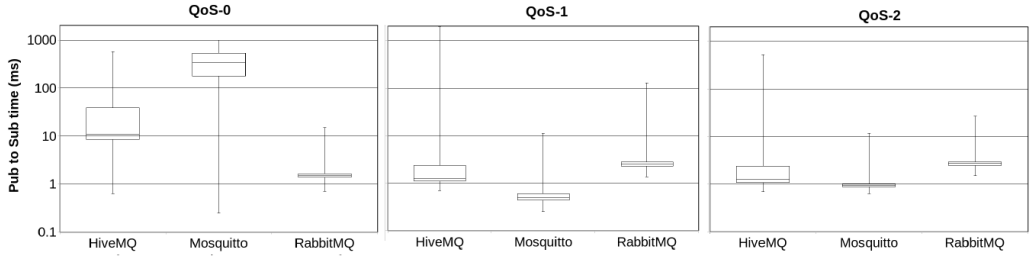


Figure 2.9: Processing Time for a message to get from the publishers to the subscribers [46].

As shown, the three brokers that stand out are Mosquitto, HiveMQ, and RabbitMQ. It is safe to say that all three fare relatively well against each other. Nevertheless, it is worth noting the low score that RabbitMQ has when it comes to the maturity aspect. This is because it is quite recent and still in heavy development. Its biggest strength against the others is also quite visible: its interoperability, as it supports several open standard protocols such as AMQP and MQTT [47].

Stepping onto the quantitative evaluation, three major aspects were studied: Responsiveness, productivity, and utilization.

Responsiveness is classified as the time a message takes to go from the publisher's side to the subscriber's side. After testing this for all three brokers, the results are presented on Figure 2.9.

RabbitMQ stands out as the most consistent one across all the QoS levels. Both Mosquito and HiveMQ are quite unreliable in terms of latency times when it comes to QoS-0. Additionally, Mosquitto registered a failure rate of almost 9% when using QoS-0.

Productivity is defined as the number of messages the broker could process taking into consideration the reception of the message, its publication, and sending it to the client. Here the winner is Mosquitto by far. As it is possible to see in Figure 2.10, Mosquito has a much bigger throughput than the others, while RabbitMQ has

the lowest. One possible explanation for this is the bigger RabbitMQ's overheads, as it has a wider range of compatibility.

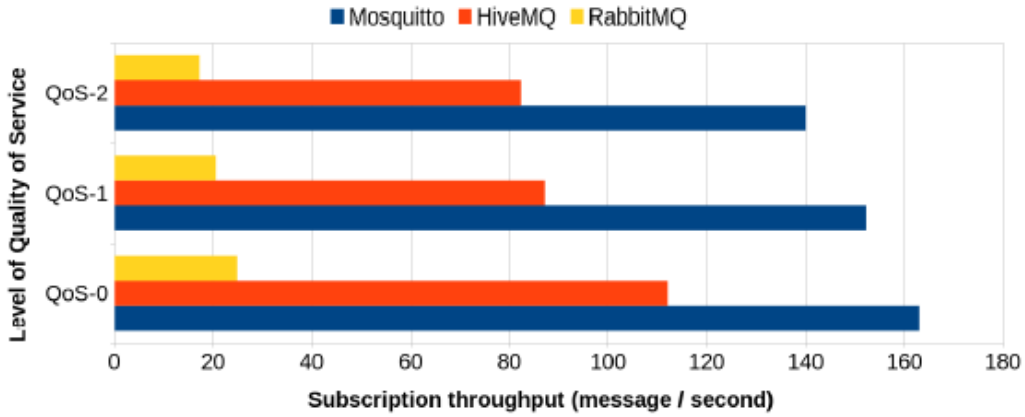


Figure 2.10: Throughput Comparison [46].

Lastly, utilization is also an important factor, as it relates to the CPU utilization and the main memory utilization that each broker has. Across all three, Mosquitto is the most stable, consistently using between 15% and 18% of RAM Memory. HiveMQ is the one that uses the most RAM across all QoS and RabbitMQ is the one that uses the least in QoS 0. Figure 2.11 shows these results.

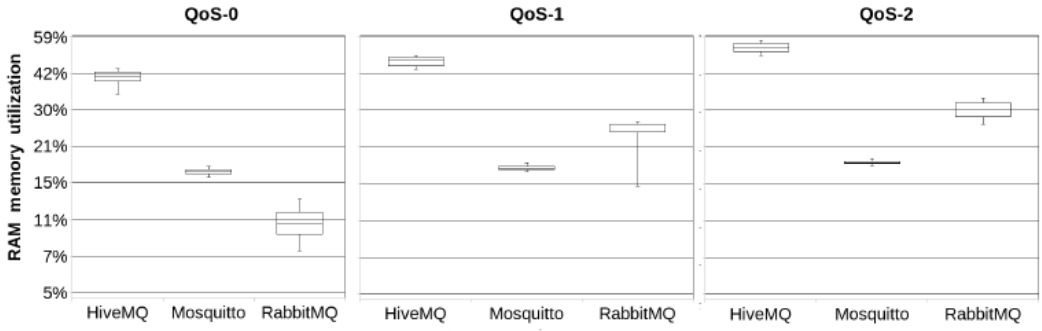


Figure 2.11: Main Memory Utilization Comparison [46].

This research supports the idea that Mosquitto is the most advantageous broker to use, in QoS > 0 contexts. It has the best processing time, the best throughput, and the most efficient memory usage. In cases where QoS-0 is required, it is probably best to consider other options such as the RabbitMQ.

2.4 Real Time Operating Systems

RTOS's have been around for a while, but their popularity has not gone down. Instead, they are being used more and more. According to Global Market Insights, the global RTOS market size was valued at USD 6 billion in 2022 [48]. In a landscape

increasingly dominated by interconnected devices, the realm of embedded systems and the IoT has thrived. There is also a lot of room and opportunities to evolve and improve this trend, as industries like automotive, manufacturing, healthcare, and even consumer electronics require real-time solutions.

Within this ecosystem, the role of RTOS in constrained devices emerges as an important enabler of efficient, responsive, and reliable operations. There are several RTOSs available on the market, with different capabilities and suiting different projects. Similar to the choice of the messaging protocol, choosing the right RTOS is an important step in the development of a project.

Some key factors typically considered are licensing models, source code modification constraints, and industry certification [49]. These are, of course, related, and even inversely proportional to each other. If an RTOS has bigger source code customization/modification, it will lack in licensing models and certifications, versus one where the user isn't allowed to change the source code. Therefore, when choosing the RTOS, the best approach would be to try to find a balance between these factors that suits the project better.

2.4.1 Some Well-Regarded Options

AspenCore does a yearly survey concerning embedded systems, and in 2023 [50] it was found that the two most used Operating System (OS) for embedded applications were Embedded Linux and FreeRTOS. FreeRTOS is commonly known as the de facto RTOS, being supported by most of the major semiconductor manufacturers [51]. Linux, on the other hand, is not an RTOS, despite being often used in this context (Embedded Linux), mainly on class 2 [6] devices (check the note below). One of the problems with Embedded Linux is the fact that it needs significant CPU resources, with factors such as memory footprint (roughly 5 MB) and boot time (roughly 2 s) being considerably higher than on a regular RTOS ((roughly 200 KB & 100 ms, respectively) [53]. As a General Purpose Operating System (GPOS), Embedded Linux is not inherently designed for hard real-time capabilities[7]. RTOSs are specifically engineered to provide deterministic behavior and timely responses

[6] IETF standardized the classification of wireless network devices with constrained system resources, based mostly on memory capacity [52]. The devices were distributed by three classes (0, 1, 2). Class 0 represents the most severely constrained devices («10 kB of RAM and «100 kB Flash). They typically have so few resource-bearing capabilities that require devices such as gateways to communicate with the Internet. Class 1 devices ($\sim$ 10 kB of RAM and $\sim$ 100 kB Flash) are unable to easily establish communication with other internet nodes utilizing a full protocol stack like TLS. However, they possess the ability to utilize a protocol stack that is deliberately designed for restricted nodes, such as CoAP. Finally, Class 2 encompasses the remaining devices. These are the less constrained ones and are typically able to handle heavier protocol stacks.

[7] Two types of real-time systems are sometimes considered. Hard and Soft. The main differences between the two revolve around the stringency of timing constraints, the consequences of missing deadlines, as well as the level of flexibility in the system's response. Hard real-time systems prioritize precise timing to prevent serious consequences, while soft real-time systems provide more flexibility, allowing for occasional delays without compromising overall system functionality [54].

to events and interruptions. They achieve this through priority-based preemptive scheduling algorithms [55], ensuring that the highest-priority task runs immediately, preempting any lower-priority task without requiring explicit CPU release.

While Linux offers various scheduling options, including a real-time scheduler [56], it falls short of providing "hard" real-time guarantees.

In this work, three different RTOSs are compared:

- FreeRTOS
- RIOT
- μC/OS-III

The choice was made based on popularity and license. The first two are open-source while the latter is close-source. This distinction is very relevant when comparing RTOSs, as it gives insight into the trade-offs of both systems. Open-source systems are transparent and allow modification and distribution. On the contrary, closed-source RTOSs offer limited visibility on the underlying code but can provide proprietary features and dedicated support, as well as additional security features.

FreeRTOS, originally developed in 2003 and currently owned by AWS, is a widely adopted open-source RTOS known for its portability and ease of use, with a wide range of supported devices [57]. It offers a preemptive, priority-based scheduler and supports a variety of architectures (ARM, x86) [58]. The preemptive nature of FreeRTOS means that it can interrupt a task that is currently running and give control to a task with a higher priority. It is also highly documented and researched and there is a lot of support for developers.

On the other hand, the micro-kernel-based RIOT presents itself as "The Real-Time Operating System for the Internet of Things", as it was designed specifically to attend to the needs of the IoT. Developed by a collaborative community in 2009, RIOT offers energy-efficient and low-resource consumption, which is good for constrained applications. One of the factors that allow it to be so power-efficient is the fact that it is tickless, which means it does not have a timer that fires periodically to emulate concurrent execution by switching threads continuously [59].

μC/OS-III is one of the versions of an RTOS provided by Micrium Inc. It is also micro-kernel-based and supports multithreading and Inter-Process Communication (IPC). It allows any number of application tasks, priority levels, and tasks per level, limited only by processor access to memory, which is an improvement from its previous version (μC/OS-II) which only allowed a maximum of 255 tasks [60].

Table 2.2 shows an organized comparison of these RTOSs.

Table 2.2: Comparison of RTOS Features [60] [61] [62].

RTOS Name	Scheduler	Programming Model	Language	Ntw Stacks
FreeRTOS	Preemptive Optional tickless	Multithreading	C	None
RIOT	Preemptive Tickless	Multithreading	C/C++	gnrc, OpenWSN ccn-lite
μC/OS-III	Preemptive Fixed tick rate	Multithreading	C/C++	None

2.4.2 A Performance Analysis

Several studies have been conducted regarding a detailed comparison between different RTOSs. One such study is from Rafael Raymundo Belleza *et al* [63] who compared different RTOSs and benchmarked their various real-time proprieties. These include task-switching time, the time for getting and releasing a semaphore, the time for passing a semaphore, and the time to pass and receive a message, among others.

Task switching time is an important test to carry out, as it serves as a crucial performance metric for evaluating the preemptive scheduling capabilities of an RTOS. This test measures the time the RTOS can transition from a higher-priority task (Task A) to a lower-priority task (Task B) and back. Both on task suspension times and on task resume times, RIOT is the faster RTOS, followed by FreeRTOS and μC/OS-III. This is also true when taking into account the Standard Deviation (STD). The STD is an important factor, as consistency and predictability are two crucial aspects in RTOSs. Tables 2.3 and 2.4 show the results of this test.

Table 2.3: Task Suspension Statistics Adapted from [63].

	FreeRTOS	RIOT	μC/OS-III
mean	6.255775	5.049999	8.441663
std	0.1203	0.000322	0.041686

Table 2.4: Task Resume Statistics Adapted from [63].

	FreeRTOS	RIOT	μC/OS-III
mean	5.720775	5.338889	8.475
std	0.099118	0.007857	0.008334

This and the remaining test led the researchers to conclude that while FreeRTOS was not as consistent as RIOT, it still achieved good results. μC/OS-III is a good

option when high-integrity and safety-critical systems are needed, as it is pre-certified in important standards.

One other important parameter is power consumption. Lobna Kriaa *et al* [64] compared RIOT and FreeRTOS's performance regarding this aspect. The experiment aims to compare power consumption between nodes implementing FreeRTOS and RIOT with the SiSP protocol on Arduino Mega 2560 boards. The study focuses on nodes exchanging data via a serial connection, utilizing an oscilloscope and a shunt resistor for current measurement. The achieved results are shown in Figure 2.12.

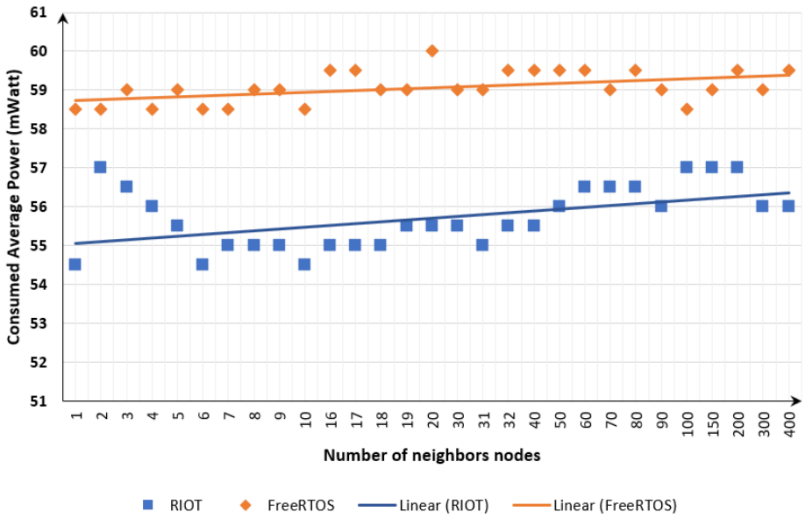


Figure 2.12: Power Consumption Comparison Between RIOT and FreeRTOS [64].

The results show that RIOT consumes less power than FreeRTOS in this situation. This can be explained by the fact that RIOT offers a tickless approach as mentioned before. Typically an RTOS schedules tasks and timers over a time base. This time base can be achieved by periodic system tick interrupts. These can be redundant as CPU time might be wasted when there are no scheduler tasks planned. Tickless systems are a good answer for this problem as these periodic interrupts are suspended, therefore saving power [65]. A possible trade-off might be lower response times.

Overall RIOT seems to be faster and more energy efficient than the others. Consistency and predictability also favor it. However, it is worth mentioning that FreeRTOS also achieved good results, and of the three, it is the most versatile and is heavily documented. μC/OS-III seems to be a good option when source code customization is not critical, and when licensing in certain areas is required. If an application requires low power consumption, RIOT seems to work the best. FreeRTOS seems to be a good fit for most other types of applications due to its

versatility and good overall performance results.

2.5 Wireless Connectivity

Wireless connectivity has become an integral part of our modern digital world, revolutionizing the way we communicate, share information, and interact with devices. As would be expected, there are a huge number of different needs and requirements that different systems have, and a one-size-fits-all approach is virtually impossible. Range, energy, data rate, and cost are just a few of these needs. Cellular Networks, LoRa, and Zigbee are good examples of these technologies.

Cellular Networks (such as LTE) provide high-speed and wide-area coverage. Facilitating seamless data transfer, these networks empower mobile devices with internet access, enabling communication on the go. It was designed for a high data rate, and it is not optimized for the IoT, where low-cost and low-power are required [66]. For short distances, other alternatives, such as Bluetooth, may be more cost-effective and power-efficient, making them more appealing.

Short-range technologies, such as Bluetooth and Zigbee are options that offer great connectivity for short-range. Zigbee, which operates in the 2.4 GHz frequency band, is tailored for low-power, home automation applications. There are ways to expand the range of these technologies, such as Mesh topologies. Here, devices are distributed in a network architecture where each node in the network is connected to every other node [67]. The problem, however, is the fact that when trying to expand over long distances the number of required nodes increases and so does the cost. Moreover, some devices might become congested which overwhelms the whole network [68].

Finally, LPWAN technologies, address the unique requirements of the IoT.

2.5.1 Low Power Wide Area Network (LPWAN)

LPWAN can be defined as a wireless network technology that can connect constrained devices (low-power, many times powered by batteries, and low-bandwidths) over long distances. Its global market size is expected to reach USD 12390 million by 2031, from USD 1689 million in 2021 [69]. This shows that the demand for these technologies will keep growing in the coming years and that they are central in the IoT world.

The market is essentially dominated by 4 LPWAN technologies [70]:

- **LTE-M** – Long Term Evolution (M2M)
- **NB-IoT** – Narrowband IoT
- **LoRa** – Long Range

- **SigFox**

They differ in several aspects such as standardization, licensing, data rate, and cost. A study conducted by Waleed Al-Sit *et al* [71], and presented at the "2019 International Conference on Wireless Communications Signal Processing and Networking" analyzed these 4 technologies and compared them. The following table (Table 2.5) is based on the results. It allows a cleaner way to visualize the differences between these technologies.

Table 2.5: LPWAN Technologies Compared [71] [72].

	LoRaWAN	Sigfox	NB-IoT	LTE-M
Licensing	Unlicensed	Unlicensed	Licenced LTE FB	Licenced LTE FB
Channel Band. (KHz)	125-500	0.1 EU 0.6 USA	180	1400-20000
Max Payload (Bytes)	243	UP: 12 DOWN: 8	1600	1000
Power Efficiency	Very high	High	Medium (Lower than LoRa)	Medium/Low (Lower than NB-IoT)
Battery Life (2000 mAH)	105 months	150 months	90 months	18 months
Max Data Rate[1] (Kbps)	∼ 50 UP ∼ 0.29 DL	∼ 0.1 UP ∼ 0.65 DL	∼ 205 UP ∼ 235 DL	Up to 1000
Coverage[2] (Km)	∼ 5 U ∼ 20 R	∼ 10 U ∼ 40 R	∼ 1 U ∼ 10 R	∼ 5

[1] UP - Upload; DL - Download.
[2] U - Urban; R - Rural.

As shown, both LoRaWAN and Sigfox operate on unlicensed bands. This has its benefits, as it allows for lower deployment costs, due to no licensing fees being needed, and also greater flexibility, as licensed bands are more standardized and regulated. Unlicensed bands also come at a cost, as they can be less reliable, and can be subject to interferences and congestion (due to shared frequency bands), while licensed ones are more reliable and predictable.

LoRaWAN, NB-IoT, and Sigfox all work in narrower bandwidths than LTE-M. This causes lower data rates but increases the battery life.

All this evidence leads to the conclusion that each one of these technologies serves a different purpose, and the choice between each one should be well thought out. LTE-M is by far the less power-efficient but provides the biggest data rate (up to 1 Mbps). In situations where good power efficiency matters, but where licensing bands are still preferable, NB-IoT is the one to go for (especially in rural areas,

where its coverage is very decent). Sigfox is suitable for applications that require small and infrequent data transmissions, such as simple sensor networks. LoRa, is also well-suited to the IoT and supports bigger data rates than Sigfox.

2.5.2 Modems

Modems are an important piece in Wireless Connectivity. Short for modulator-demodulator, they facilitate communication between devices over various networks, including the previously mentioned LPWANs, by converting digital data from connected devices into analog signals for transmission and vice versa. In the context of LPWAN, modems serve as the bridge between the low-power, long-range communication protocol used by LPWAN technologies and the data generated by connected devices. As IoT applications gain relevance, it is essential to understand the role of modems in these systems.

According to the "2022 Cellular Broadband Device and Module Market" report [73], in 2022, Quectel had the largest market share of Cellular M2M Modules (39.4%). The following is an analysis of some of the existing modems that are currently used in the industry (from Quectel and others).

Quectel BG96 - The BG96 [74] is a versatile and compact cellular modem module designed by Quectel, developed for IoT and M2M applications. It supports LTE Cat M1 (LTE-M), Cat NB1 (NB-IoT), and Enhanced General Packet Radio Services (GPRS) technologies, and facilitates reliable and low-power connectivity over a variety of cellular networks. It also supports half-duplex operation in LTE networks and has a Global Navigation Satellite Systems (GNSS) module.

Quectel BC95-G - The BC95-G [75] is a high-performance NB-IoT module that has extremely low power consumption. This ultra-compact module is also smaller than the BG96. It only supports LTE Cat NB1, and therefore has no compatibility with LTE-M.

U-Blox Sara N2 Series - The Sara N2 Series [76] are power-optimized LTE Cat NB1 (NB-IoT) modules. Similar to the BC95-G it is also very compact and only supports NB.

Some other honorable mentions would be the Quectel EG91 Series [77]. It only supports LTE Cat M1 (LTE-M) communications and Quectel M66 [78] which only supports Global System for Mobile Comunication (GSM)/General Packet Radio Services (GPRS).

Flavio Di Nuzzo *et al* [79] proposed a system to monitor the integrity of civil infrastructures using low-cost wireless sensor nodes in combination with NB-IoT to establish a long-distance connection with the LTE infrastructure network. A benchmark was done between these 3 different modules (BG96, BC96-G, and Sara N211) to assess which ones are better in which areas. The researchers focused on assessing energy consumption in power saving mode and the energy requirements

per packet during uplink[8] transmissions across diverse coverage conditions. A shunt resistor and an amplifier were used to measure the current. Identical UDP packets were transmitted, and an attenuator was used to simulate different conditions. The Received Signal Strength Indicator (RSSI) was evaluated to classify the level of coverage. The results of this test are shown in Figure 2.13.

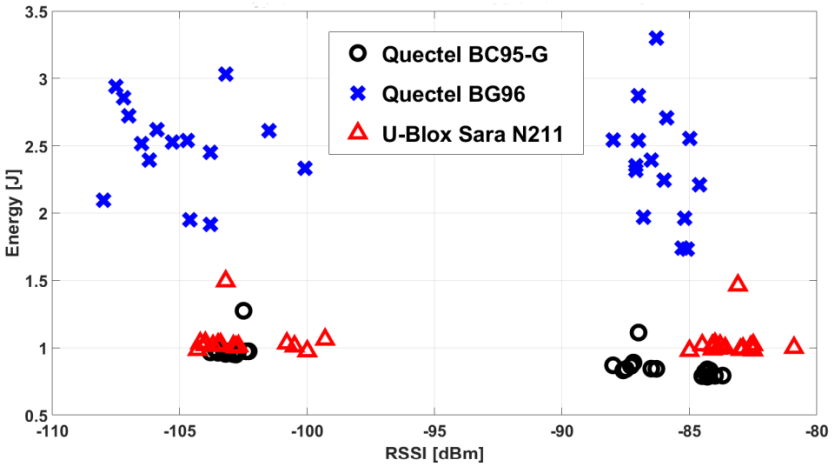


Figure 2.13: Modem Benchmark Results [79].

When it comes to power consumption, it seems obvious that the Quectel BG96 is less power efficient than the others. This comes down to the fact that, as mentioned before, it has more functionalities such as LTE CAT M1 support and GNSS, that affect its performance. As would be expected from both NB-IoT focussed modules, the results show that the Quectel BC95-G and the U-Blox Sara N2 are quite comparable. The authors reveal that their choice of using BC95-G over BG96 was due to the former's superior power efficiency, and the fact that the latter provides a set of AT-Commands that would not be needed, such as commands to configure and use MQTT.

Another project, from Nicolás Finozzi *et al* [80], introduces a collar-type device designed to monitor sheep behavior, emphasizing the integration of a Quectel BG96 for seamless communication and location tracking.

Selecting the appropriate modem for a project is important to achieving optimal and efficient results. The choice should be dependent on several factors such as the project's specific requirements, communication needs, power constraints, and geographical considerations. A careful evaluation of these aspects ensures that the chosen modem aligns with the project's objectives and technical specifications.

[8]Uplink refers to the process of sending data from a user device or sensor to a central server or network.

AT-Commands

AT commands, standing for "Attention Commands," constitute a standardized set of instructions to facilitate the communication between host devices and modems. These commands typically begin with the letter "AT" and the intended command is written afterward. There are 4 types of AT commands [81]:

- **Write:** $AT+ < command >=< value >$
- **Read:** – $AT+ < command >?$
- **Test:** – $AT+ < command >=?$
- **Execution:** – $AT+ < command >$

An example of the usage of these commands would be the establishment of an MQTT communication. The command "$AT + QMTOPEN =< tcpconnectID >$,” $< hostname >$ ”, $< port >$" is necessary to open a network for the MQTT client. A few more commands would need to be used to create a functional connection, but as mentioned, this command serves as an example.

These commands can be used in any circumstance that requires communication with modems. Qian Zhicong *et al* [82] propose a mobile forensic software system model based on AT Commands. These commands are exploited to gather information stored in mobile phones' memory such as phonebooks and call logs (AT + CPBS), short messages (AT + CMGL), calendars, and other information. The commands are sent through the serial port and can help judicial authorities solve criminal cases.

Chapter 3

Main Project

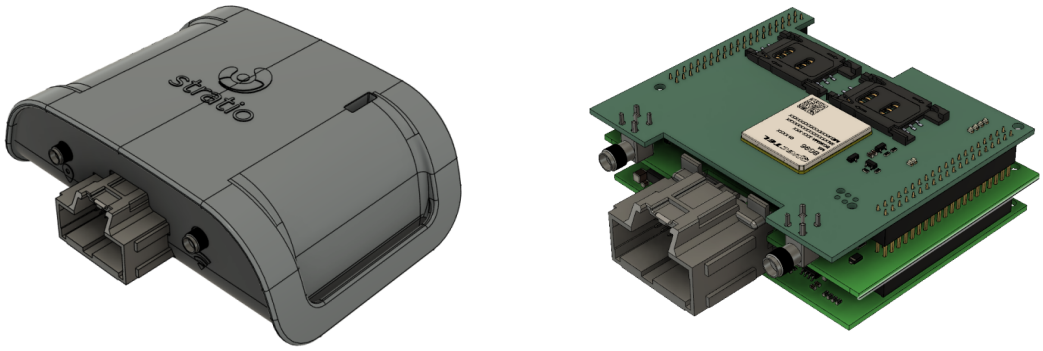
In this chapter, the focus will be on the setup and implementation of the main project. It begins with a brief description of the Stratio Databoxes, which are fundamental for generating and transmitting logs, followed by an examination of the broker infrastructure essential for managing data exchanges. Emphasis is placed on the configuration and deployment of the broker, which operates within Docker containers to facilitate efficient development and deployment processes. A Pilot Project is also addressed, developed to test the technologies being used. Several software architecture choices are explained, as well as their validation. Finally, this chapter outlines the steps taken to gather the results, which are explored in detail in the next chapter.

3.1 Hardware and Firmware

3.1.1 Stratio Databoxes

Most of the code written for this project is to be implemented in Stratio's Databoxes. Figure 3.1 shows a Databox both covered (a) and uncovered (b). This device is the one installed on the vehicles and will be publishing the logs to the broker.

Stratio uses them to gather crucial information about the vehicle's systems that help both track its location and provide data so that the predictive models determine if there is any risk of a component (such as brakes or engine) failing.



(a) Stratio Databox.

(b) Stratio Databox Uncovered.

Figure 3.1: Stratio Databox.

The databox is equipped with a BG96 modem. A render of the modem is distinctly visible in gray in Figure 3.1 (b). It is responsible for establishing the LTE-M cellular network connection. It is also equipped with a microcontroller (STM32) which runs FreeRTOS.

3.1.2 Broker

During this project, three different types of brokers were used. The first one is the "Mosquitto" public broker. This was the easiest one to implement as there is no need to build any type of infrastructure. However, being a public broker, anyone can see the data being published to the topics, and anyone can publish to a topic. This presents a problem of data protection and data integrity. The second and third brokers to be implemented share some commonalities, such as the fact that they are both based on the "Mosquitto" docker image [83]. The main difference between the two brokers is their deployment environment. One runs on the local machine (localhost), and the other runs on a dedicated server. The reason for this is that these brokers serve different goals. The former was developed so that the software could be tested without the interference of network-related issues. The other acts as the de facto server to be used in real-life implementation, acting as an intermediate between the databoxes in the vehicles and the clients in the office.

3.1.3 Other Clients

Apart from the Stratio databoxes, there are a few more clients that communicate with the broker. These can be any device that exchanges data with the said broker. They mainly subscribe to the topics the databoxes are publishing to and process the data, but they are also capable of sending information, such as commands that determine the type of log messages to be sent.

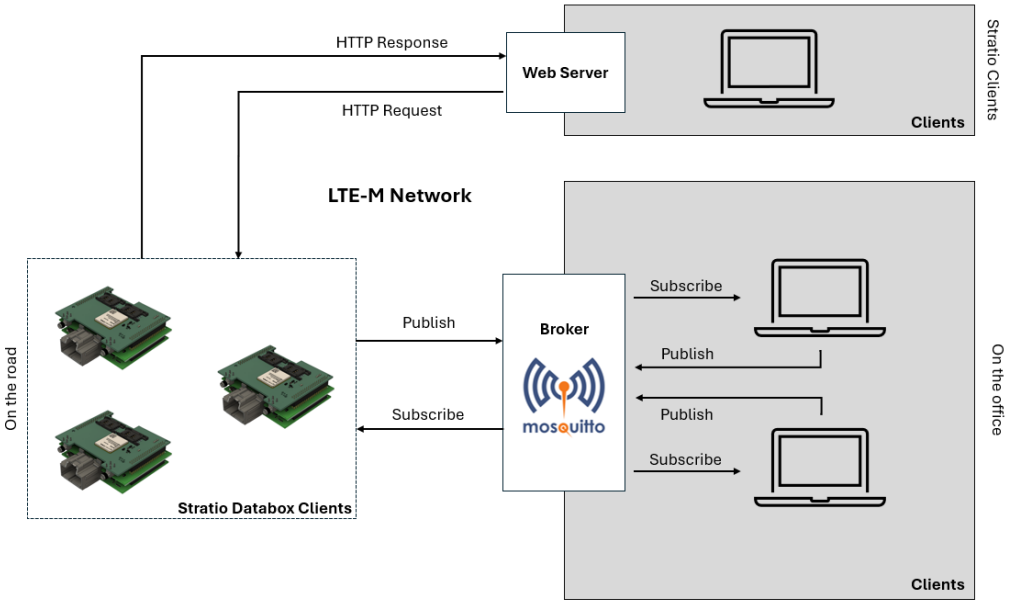


Figure 3.2: Interaction Between the Clients and the Broker.

Figure 3.3 shows an overview of the way all these intervenients interact with each other while providing insights into the project's architecture. It depicts the broker as the central entity, acting as an interlocutor between the different clients. It also shows, the previously existing network connection via HTTP, that was established to send the vehicle state information. It hopes to accurately depict the difference between both systems and help understand the focus of the project.

3.2 Pilot Project Assessment

A Pilot Project was developed as a small version of the overall project. The objective was to have a working MQTT communication system that ought to be straightforward, being able to publish and subscribe to a topic. This was done for a couple of reasons. The first was to understand if the modem could perform the intended task. A project like this one requires a high level of confidence in the hardware that is used, as it involves critical functionalities, such as communication. Without this understanding, the risk of encountering unexpected issues or failures during operation increases, and the risk is higher for problems down the line.

The second is that the development of a pilot project can act as an entry gate for developers to familiarize themselves with the project and understand the basics before hopping into the bigger project.

This chapter introduces the first part of the project. The following subsections delve into the hardware and software choices involved in the pilot, its architecture, as well as the achieved results.

3.2.1 General Overview

The idea behind this pilot project is to have a closed software loop that continuously listens to the serial port waiting to receive messages that were published to the subscribed topics. Figure 3.3 is a flowchart that represents the developed system.

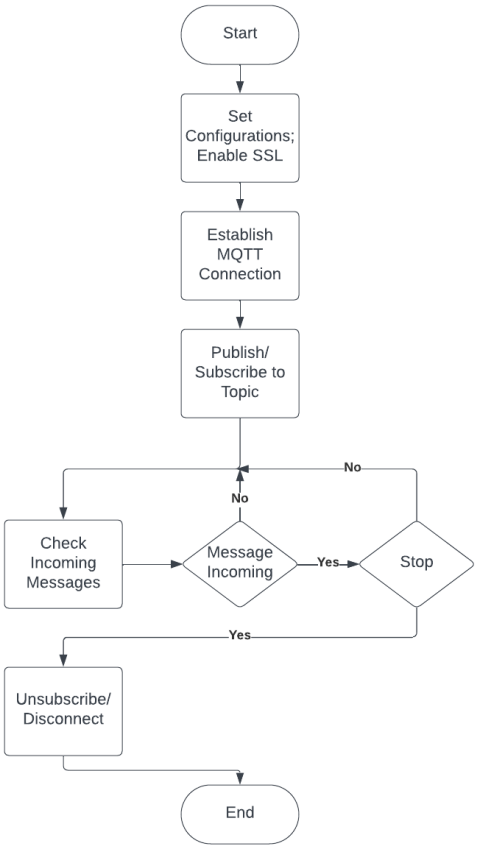


Figure 3.3: Pilot Project General Flowchart.

The first thing to happen is the modem configurations. This will prepare the modem to send and receive MQTT messages, as well as enable SSL connection for security concerns. After this, an MQTT connection is established and a publish attempt is performed. A subscription to a default topic is then made. It is worth noting that the script is organized so that, should any problem occur, errors are raised, and logs are printed to facilitate analysis and debug.

Once the subscription is achieved, the script will enter a loop that is set up to wait for any incoming publishes. If there is a message incoming, it will print it and return to waiting, unless the content of the message is the word "stop". If it is, it has instructions to unsubscribe from the topic and close the MQTT connection.

Figure 3.4 shows the process of establishing a communication, publishing, and subscribing, as well as the workflow after a subscription has been achieved.

(a) Initial Process. (b) After Subscription.

Figure 3.4: Charts Detailing the Communication.

By doing this, this pilot project can test the most important MQTT features and prepare a ramp-up to the actual project.

3.2.2 Hardware & Software

There were several technologies involved in creating this pilot. The main project itself was done in C and C++. However, for this pilot, it was determined that Python would be a better option, as it provides a more straightforward syntax, it has packages that simplify MQTT integration and Stratio already had some different PoC developed in this environment, as well as some useful infrastructure already set-up (serial parsers and so on). The development environment used was PyCharm [84].

Similarly to the main project, a modem was integrated. By default, the BG96 was chosen, as this is the one that will be used in the main project (due to its LTE-M capabilities). AT commands were implemented, since these are the standard way to command the modem.

Either to read the messages being received or to check the correct transmission of the messages being sent, the tool CuteCom [85] was used. It is a graphical serial port communications program, that allows the user to see such messages.

Lastly, it was both important to have an MQTT broker acting as the server to which this script connects and to have an external client that could publish to the topics that the script subscribed to. Mosquitto [28] was chosen as a broker, mainly due to its ease of use, as it hosts a free, publicly available server, and myMQTT [86] was used as a client (chosen for being free and practical).

3.2.3 Communication with the Modem

As mentioned previously, AT commands are used to communicate with the modem. Tools like CuteCom make this easy by allowing the user to send commands directly to the modem through a simple interface. In CuteCom, the user simply selects the serial port, types in the command, and sends it to the modem immediately.

However, automating this process with a script is more complex. In a script, there is no way to type a command and it being directly sent to the modem, as in CuteCom. Instead, a function that handles this communication is needed. "*send_at_command*" was developed to perform exactly this. It uses an adapted "*pyserial*" library [87] to write the commands to the serial line and checks for a response from the modem.

A few things are important to understand before further developing the way this function works. For every AT command that is sent, the modem generates a response back. For example, the AT command "AT" (a simple command typically used to check if there is communication with the modem) has a response of "OK", confirming that the communication is successful. More complex commands have more complex replies. Using as another example, the AT command mentioned in Chapter 2:

INPUT: $AT + QMTOPEN =< tcpconnectID >, " < hostname > ", < port >$

REPLY: $OK + QMTOPEN :< tcpconnectID >, < result >$

It is important to analyze these replies[1] every time that an AT command is sent, as they contain crucial information on whether the action was successful or not. In this case, the parameter "<result>" can contain several values such as: "-1" indicating that it failed to open the network, "0" when the operation is successful, "1" when a parameter is wrong, and so on.

The "*send_at_command*" function is provided with multiple information as a parameter: The AT-command itself (e.g. "QMTOPEN"), the payload (e.g. " $< tcpconnectID >, " < hostname > ", < port >$"), the mode (e.g. "write"), and some other less relevant information. Figure 3.5 shows how this function operates.

Figure 3.5: "send_at_command" Function Scheme.

[1] Elements like the <tcpconnectID> etc are replaced by their real values. This is just a written representation.

It starts by checking what is the mode determined for the outbound AT command. Each mode has its own different format that is applied once the checking is complete (check 2.5.2). After this, the AT command is sent via an internal Stratio function that sends the command, gets the response, and provides it in a list[2].

The last step is to check the response for errors. These errors can be of two types. Either due to there not being an "OK" in the response message, or because the "result" field has an error value.

Additionally, there is a function named "*send_at_lists*" that is simply responsible for receiving a list of AT-commands and triggering the "*send_at_command*" function for each command, using a *for loop*. This allows the sending of a bulk of AT commands at once.

Once the method that is in charge of sending these commands is functional, it is possible to start configuring the modem.

3.2.4 Configuration and Security

By default, the BG96 modem already accepts MQTT connections without any previous configuration needed. However, in some cases, it might be necessary. This section delves into how the modem configuration relates to the security of the communication.

The AT-Commands used for the configuration are:

- **AT+QMTCFG** Configure optional parameters of MQTT [88];
- **AT+QFDEL** Deletes a specified file or all the files in the storage [89];
- **AT+QFUPL** Uploads a specified file [89];
- **AT+QSSLCFG** Configure Parameters of an SSL Context [90];

"AT+QMTCFG" allows the user to set parameters such as the MQTT version (e.g. MQTT protocol v3 or v4), the keep-alive time as well as the type of session. But most importantly for this project, it allows the choice between using SSL or not.

A function named "*mqtt_configuration*" generates an AT command equivalent to the one below:

$$AT + QMTCFG = ssl, 0, 1, 2$$

Here, 0 represents the *"tcpconnectID"*, 1 represents the enabling of the SSL feature, and 2 represents the SSL context index which could be anyone from 0 to 5. This command is then sent to the modem through the "*send_at_command*" function.

[2]Essentially it writes the command into the serial line, and then runs a "read" script, waiting for an answer.

After the reply has been cleared of errors, the modem is ready to establish an SSL connection. However, there are still important steps missing. It was agreed that there was no need to have a connection both encrypted and requiring authentication. Therefore, only the former was implemented. To establish any SSL communication, a certificate authority file is required. This file identifies the server to which the certificate is issued, such as its identity, public key, and other relevant details. This adds a layer of security as it stops devices from connecting to malicious networks. Mosquitto [28] provides a certificate that the users can use for this very purpose.

A function named *"enable_ssl_connection"* is responsible for deleting the old certificate file in the case that there is one, and uploading this new one. This is done using the "AT+QFDEL" and "AT+QFUPL" AT commands respectively.

With the certificate now set in place, the last step is to configure the SSL parameters. This is done by implementing the "AT+QSSLCFG" command. It specifies parameters such as SSL version, the cipher suites (cryptographic algorithms), and the certificate file's path. After sending this command and checking for any errors in the reply, the modem is ready to establish an SSL connection.

3.2.5 MQTT Interactions

Now that all the infrastructure is set up, it is possible to finally establish an MQTT communication. This section will focus on the process of opening a network, connecting a client, and doing basic data transactions with the publish and subscribe tools. The AT-Commands used for this section were:

- **AT+QMTOPEN** Open a Network Connection for MQTT Client[88];
- **AT+QMTCONN** Connect a Client to MQTT Server [88];
- **AT+QMTDISC** Disconnect a Client from MQTT Server [88];
- **AT+QMTPUB** Publish Messages [88];
- **AT+QMTSUB** Subscribe to Topics [88];
- **AT+QMTUNS** Unsubscribe from Topics [88];

A function named "*mqtt_connection*" combines both the "AT+QMTOPEN" and the "AT+QMTCONN" and sends them to the modem via the "*send_at_command*" function. The main difference between these two is that the former creates a socket, opening a network-level connection for an MQTT client as if building a bridge between the device and the broker. As a parameter, it requires the "<tcpconnectID>", the "<hostname>", and the "<port>". The latter is responsible for actually establishing the protocol-level connection, providing authentication credentials and other parameters required for establishing the MQTT session with the broker. In this pilot,

as an input, it requires the "<tcpconnectID>" and the "<clientID>", as authentication methods were not implemented. It is worth mentioning that the command "AT+QMTDISC" serves as the antithesis of the "AT+QMTCONN" command, as it is used to terminate the connection.

Once the connection is established, the modem is ready to start publishing and subscribing. The function *"mqtt_publish"* has a particular architecture. When the "AT+QMTPUB" is sent, the modem responds with "<", letting the user know that it is ready to receive an input message.

Because of this, the *"mqtt_publish"* function needs to wait for this reply to only then send the message. Figure 3.6 shows a flowchart representative of the implementation of this function.

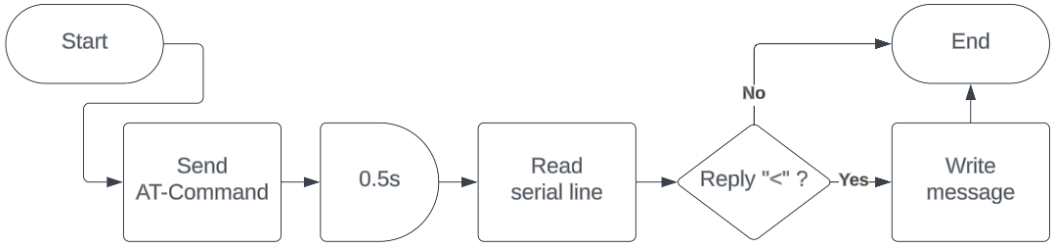


Figure 3.6: *"mqtt_publish"* Flowchart.

It is worth mentioning that as a parameter, the publish command requires, among other items, the "<messageID>", the "<topic>" to which the publish will be done, and the "<qos>" level.

A subscription was also implemented. This implementation is fairly simple, as it only requires a function ("*mqtt_subscribe*") that sends the "AT+QMTSUB" command to the modem. It requires similar parameters as the publish command. With this, any message that is published on the topic will be delivered to the subscribed client as a published message.

As mentioned in the General Overview section (3.2.1), after executing all these processes, the script enters a loop of waiting for incoming messages and analyzing them, looking for the keyword "stop". Upon receiving this message, it will unsubscribe and terminate the MQTT connection, ending the script. The function responsible for this loop is "*mqtt_check_reply*". Listing 3.1 shows a snippet of this function.

```
1 if response_bytes:
2     logging.debug(f'[IN][B] {response_bytes[len(response_bytes) -
      1]}')
3     for response_str in response_bytes:
4         if "stop" in response_str.decode('ascii'):
5             self.mqtt_unsubscribe(tcp_connect_id=tcp_connect_id,
      message_id=message_id, topic=topic)
```

```
6                 return 1
```

Listing 3.1: "*mqtt_check_reply*" Snippet.

The cycle is now complete and the script ends.

3.2.6 Final Result

This section intends to show the final result of the pilot project implementation. As Figure 3.7 shows, the output of this script is composed of several logs detailing the actions that take place. A close analysis of these logs validates, among other things, the successful publishing and subscription of the demo topics, as well as the interruption of the loop when the word "stop" is received.

```
2024-03-27 16:37:28,730 DEBUG: [OUT][STR] AT+QMTOPEN=0,"test.mosquitto.org",8883
2024-03-27 16:37:32,398 DEBUG: [IN][B] [b'AT+QMTOPEN=0,"test.mosquitto.org",8883\r', b'OK', b'+QMTOPEN: 0,0']
2024-03-27 16:37:32,899 INFO: connect MQTT client to server
2024-03-27 16:37:32,899 DEBUG: [OUT][STR] AT+QMTCONN=0,"testClient"
2024-03-27 16:37:35,864 DEBUG: [IN][B] [b'AT+QMTCONN=0,"testClient"\r', b'OK', b'+QMTCONN: 0,0,0']
2024-03-27 16:37:38,976 INFO: Successfully Published: Publish Test
2024-03-27 16:37:39,477 INFO: subscribe to topic
2024-03-27 16:37:39,478 DEBUG: [OUT][STR] AT+QMTSUB=0,1,"aeris/test",1
2024-03-27 16:37:41,890 DEBUG: [IN][B] [b'AT+QMTSUB=0,1,"aeris/test",1\r', b'OK', b'+QMTSUB: 0,1,0,1']
2024-03-27 16:37:50,735 DEBUG: [IN][B] b'+QMTRECV: 0,0,"aeris/test","Demo Message #1"'
2024-03-27 16:37:54,756 DEBUG: [IN][B] b'+QMTRECV: 0,0,"aeris/test","Demo Message #2"'
2024-03-27 16:37:57,470 DEBUG: [IN][B] b'+QMTRECV: 0,0,"aeris/test","Demo Message #3"'
2024-03-27 16:38:02,445 DEBUG: [IN][B] b'+QMTRECV: 0,0,"aeris/test","Demo Message #4"'
2024-03-27 16:38:11,844 DEBUG: [IN][B] b'+QMTRECV: 0,0,"aeris/test","stop "'
2024-03-27 16:38:11,844 INFO: unsubscribe to topic
2024-03-27 16:38:11,844 DEBUG: [OUT][STR] AT+QMTUNS=0,1,"aeris/test"
2024-03-27 16:38:14,206 DEBUG: [IN][B] [b'AT+QMTUNS=0,1,"aeris/test"\r', b'OK', b'+QMTUNS: 0,1,0']
2024-03-27 16:38:14,707 INFO: disconnect client from server
2024-03-27 16:38:14,707 DEBUG: [OUT][STR] AT+QMTDISC=0
2024-03-27 16:38:16,868 DEBUG: [IN][B] [b'AT+QMTDISC=0\r', b'OK']
```

Figure 3.7: Pilot Project Result Logs.

The development of a Proof of Concept had a crucial role in the development process of the final project. It has an impact on the validation of ideas, and testing of technologies and methodologies before the full-scale implementation. The pilot addressed several key aspects of MQTT communication, including hardware and software integration, configuration, security, and basic data transactions. Communication with the modem was achieved through AT commands, facilitating the configuration of MQTT parameters such as SSL encryption. The pilot provided insights into potential challenges and optimizations, laying a solid foundation for the subsequent development phases and mitigated risks and uncertainties.

3.3 Main Project Implementation

In this section, the main focus will be on the implementation of the project. Emphasis will be placed on how data exchanges occur at the software level and how they

are integrated with the existing codebase. The section will also delve into the build and configuration of the broker. By exploring these aspects in detail, the section aims to provide a comprehensive understanding of the project's implementation and its integration into the existing software infrastructure.

3.3.1 Broker Infrastructure

The first thing to be set up ought to be the broker. It is needed for nearly every step of this project, and therefore it should be functional before anything else.

This project is divided into different docker containers. The main firmware is present in one container and the broker is in another. This has some advantages, as Docker [91] is a platform used for not only developing applications but also for shipping and running them within portable containers. Developers can build applications and package them, together with their dependencies in an isolated and concise environment. One of the biggest benefits of this approach is the consistency of not having to worry about outside factors (such as changing the machine running the application, the operative systems, and so on).

Figure 3.8 represents a block diagram of the docker container organization in this project. It shows that, despite being two different containers they can still interact with each other. This is due to the way they are created, which will be delved into after.

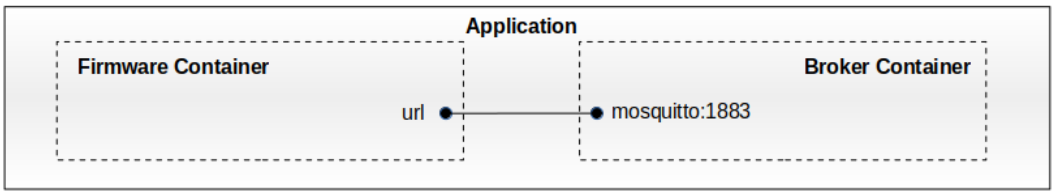


Figure 3.8: Docker Organization.

To create a docker container, a docker image is required. Much like a blueprint provides a plan or template for constructing a building, a Docker image contains all the necessary instructions and dependencies needed to create and run a Docker container. This includes the code, libraries, dependencies, and more, all in a lightweight package.

A Dockerfile is used to build an image. This file is equipped with a set of instructions that command actions such as copying files into the image, installing dependencies, and more.

Eclipse Foundation's open-source MQTT Broker (Mosquitto) [92] has its official docker image [83].

Configuration and Deployment

The main development environment should be able to communicate with the Mosquitto container. In order to configure multi-container docker applications, a "*docker-container.yml*" file is very useful. It allows the user to define multiple important configurations. Listing 3.2 shows the contents of this file that are related to the Mosquitto docker container.

```
1 mosquitto:
2    build:
3      context: services/mosquitto
4    hostname: "mosquitto"
5    restart: always
6    ports:
7      - 1883:1883
8      - 9001:9001
```

Listing 3.2: "*docker-container.yml*" Snippet.

Here, "mosquitto" in line 1 represents the service, and it is used to identify the operation. As an example, docker-compose commands use these names to target specific containers, such as: "*docker compose up mosquitto*".

The "build" parameter specifies the context for the build, which in this case leads to a different directory ("services/mosquitto"), where it will look for a Dockerfile to build an image from.

The "hostname" parameter sets the hostname of the container, and its importance is related to the communication between the different containers. If "Container A" wants to connect to the localhost broker created by "Container B", it can not use the "localhost" address. If it does, then it will be trying to connect to itself, as it assumes "local" as itself. It needs to use the hostname defined by "Container B", which in this case is "mosquitto".

"restart" is a parameter that commands the container to always restart if it stops. And finally, the "ports" parameter is used to map ports from the container to the host. For example, "1883:1883" means that the port 1883 of the host machine is mapped to the port 1883 of the container.

Inside the "services/mosquitto" directory, a few folders and a file are present:

- **Dockerfile** Contains instructions for image build;
- **Config** Contains a configurations file "mosquitto.conf";
- **Data** Stores data from MQTT interactions;
- **Log** Stores logs from MQTT interactions;

Listing 3.3 shows the contents of the Dockerfile.

```
1 FROM eclipse-mosquitto
2
3 RUN mkdir -p /mosquitto/
4 COPY ./mosquitto /mosquitto
```

Listing 3.3: Contents of Dockerfile.

This file imports the official Dockerfile provided by Eclipse [93] and mounts the "mosquitto" directory in the image.

The last step is to configure the broker. This is done in the "mosquitto.conf" file. Listing 3.4 shows its contents.

```
1 allow_anonymous true
2 listener 1883
3 listener 9001
4 protocol websockets
5 persistence true
6 persistence_location /mosquitto/data
7 log_dest file /mosquitto/log/mosquitto.log
```

Listing 3.4: Contents of "mosquitto.conf" File.

"*allow_anonymous*" being set to true means that clients can connect without providing a username. Ports 1883 and 9001 are specified and persistence is set to true so that data is stored.

With all the configuration done, the only thing left to do is to compile the container. Two docker commands are necessary for this: "*docker compose build mosquitto*" and "*docker compose up -d mosquitto*". The former triggers the build of the docker image and the latter starts the "mosquitto" service.

Interacting With Broker as "localhost"

The first, and fastest way, to test if the broker is working is by hosting it locally. This asserts that the second type of broker, mentioned at the beginning of this section, is functional and that it can be used to validate the software. For this, the only thing needed is a program to act as a client, such as MQTT Explorer [94]. Figure 3.9 shows the result of an established connection to "localhost:1883".

This broker comes with a few pre-established topics that depict some information, such as the number of clients that are connected to it and the number of subscriptions, among others. A test message was published and received without any problems, therefore the docker container that shelters the Mosquitto broker is working properly.

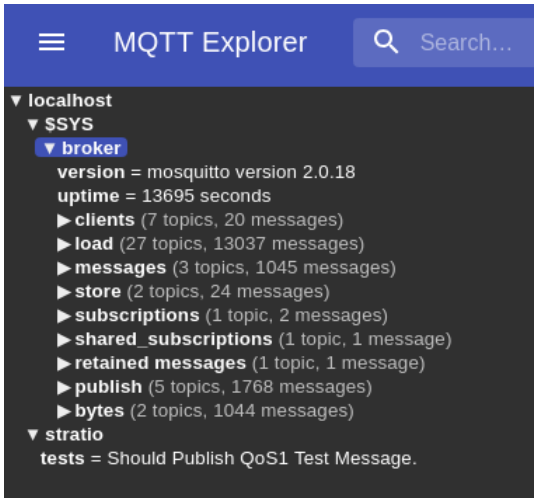


Figure 3.9: MQTT Explorer Client on Localhost.

Interacting With Broker via Dedicated Server

For the main project, it is important that a dedicated server is built to accommodate the broker. After some discussion, it was decided that the server should be running on a PC. This solution requires the creation of a new port forwarding rule in the router, as well as the definition of a static IP address for the machine hosting the server. By default, a machine like a PC has a dynamically attributed IP that is chosen between a certain range of valid options. This method has several advantages as it simplifies network configuration, saves time, optimizes resource use, and supports network scalability[3]. However, when using a machine to host a server, a static IP address ensures that the server can always be reached at the same address.

The way this was set up, makes it so that any incoming traffic directed to port 1883 on the router's external/public IP address will be forwarded to the internal IP address 192.168.1.253 on the same port. As the Mosquitto docker container also is programmed to use port 1883, the dedicated server is running as a broker.

[3]Using dynamically attributed IP in large networks is useful as developers can add new devices to the network without worrying about defining IP addresses manually

3.3.2 Software Architecture

To begin developing the core functions of any MQTT communication (MQTTPublish, MQTTSubscribe, etc), some base software infrastructure, such as libraries is useful.

FreeRTOS provides several libraries concerning multiple technologies, that intend to facilitate their integration. One of the biggest FreeRTOS praises is the fact that it is so well documented and offers a wide range of support. MQTT is one of such technologies, as FreeRTOS provides the "coreMQTT" library [95]. By their definition, it is a "client implementation of the MQTT standard", providing a lightweight, high-level, and simple API.

To integrate coreMQTT resources into the development environment, a "core-mqtt.cmake" file was created. This file is tailored specifically for CMake (which then orchestrates the build process), facilitating the process of incorporating coreMQTT libraries and dependencies into CMake-based projects. This file fetches the coreMQTT's git repository and allocates it locally.

The library uses default macros to configure the MQTT communication. However, to customize some of these macros a configuration file is necessary. Listing 3.5 shows the contents of the file created for this project.

```
1 #define MQTT_PINGRESP_TIMEOUT_MS 5000
2 #define MQTT_RECV_POLLING_TIMEOUT_MS 50
3 #define MQTT_SEND_TIMEOUT_MS 5000
4 #define MQTT_MAX_CONNACK_RECEIVE_RETRY_COUNT 5
```

Listing 3.5: coreMQTT Configuration File.

These "*#define*" statements set various timeout values and retry counts for MQTT communication, including the duration for receiving a ping response, polling for incoming messages, sending messages, and the maximum number of retries for receiving a connection acknowledgment.

With this infrastructure set up, the MQTT communication features are easier to build, but nonetheless challenging.

To make this communication work, several functions were developed within the FreeRTOS environment. Figure 3.10 shows each of these functions with a small description of their objective and the functions on which they are dependent.

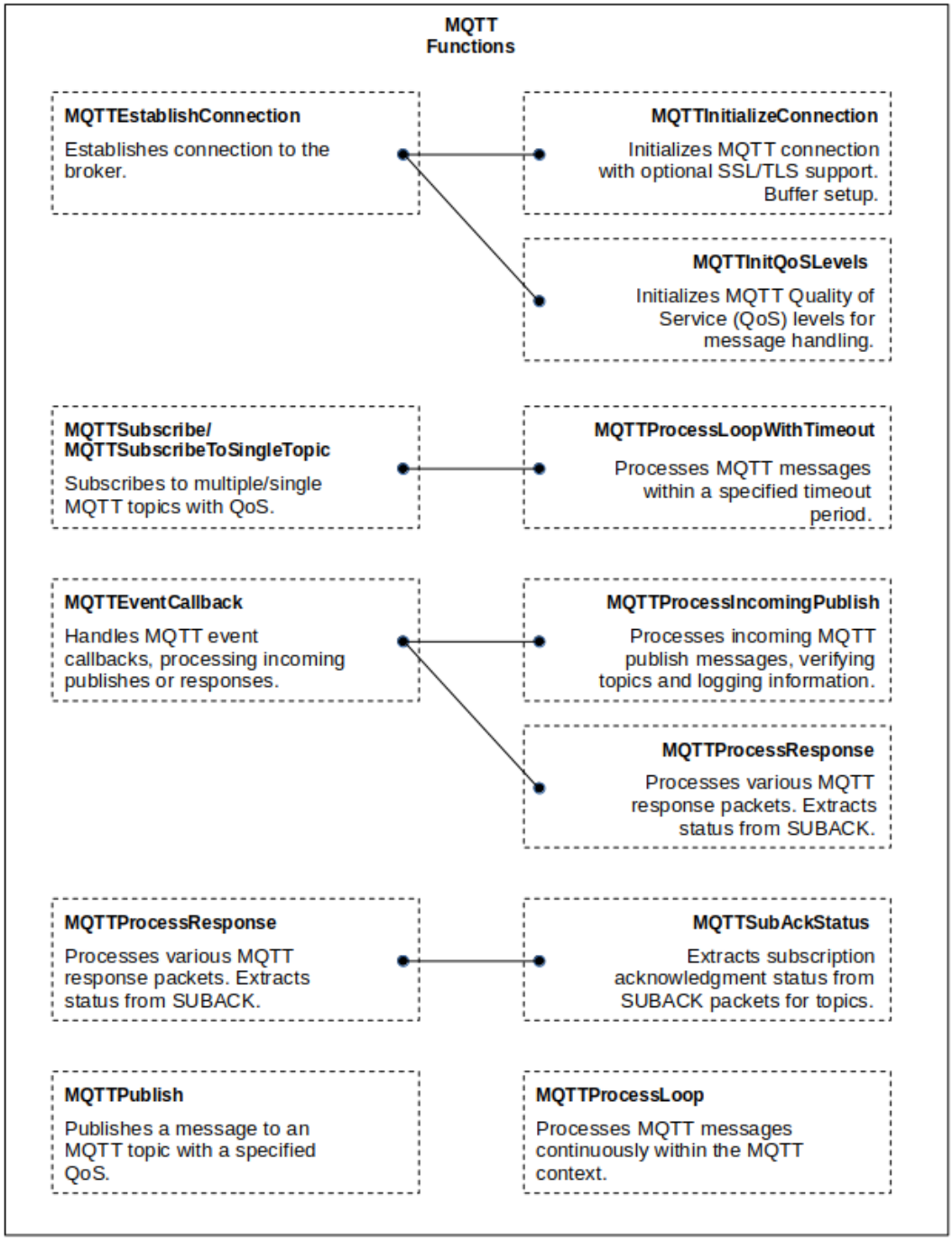


Figure 3.10: MQTT Function Dependency Scheme.

MQTTInitializeConnection

The "*MQTTInitializeConnection*" function is responsible for setting up the necessary data structures and configurations to enable MQTT communication. There, memory is allocated in the buffer for the MQTT context object, parameters are configured such as the transport interface, buffer size, URL, and port. Initialization prepares

the MQTT client to perform MQTT operations. Here, a TCP socket connection is also established at the network level.

Figure 3.11 shows a flowchart containing a simplified version of this function's logic.

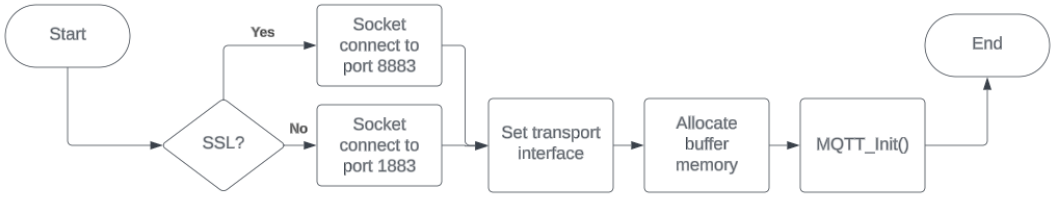


Figure 3.11: "*MQTTInitializeConnection*" Flowchart.

The first thing to happen is the decision on whether SSL is being used for this session or not. If it is, then a function named "*TCP_Sockets_Connect*" creates a socket and connects it to port 8883, which is the default port for SSL MQTT on Mosquitto. After this, the transport interface is defined. It allows higher-level protocols, such as MQTT, to communicate with the underlying network transport layer. Listing 3.6 shows a snippet of this definition.

```
1 .recv = reinterpret_cast<TransportRecv_t>(TCP_Sockets_Recv),
2 .send = reinterpret_cast<TransportSend_t>(TCP_Sockets_Send),
3 .pNetworkContext = reinterpret_cast<NetworkContext_t *>(this->
      socket),
```

Listing 3.6: Transport Interface Definition.

Here, the "*.recv*" field points to a function that receives data from the network, while "*.send*" points to one that sends data over the network. The last field, "*.pNetworkContent*", contains a pointer to the socket handle. This decoupling between the MQTT protocol and the underlying network allows for flexibility and portability, as different network implementations can be used interchangeably with the same MQTT client code.

Next, it is important to allocate memory for the network buffer, which is used by an MQTT client to store outgoing and incoming MQTT packets. In this case, 2 kilobytes were allocated.

Lastly, "*coreMQTT*"'s function "*MQTT_Init*" can be executed, as shown in Listing 3.7. This function initializes the MQTT context. Other than the context, the transport interface, and the buffer, there are two more parameters required: the time utility function (to provide the current time in milliseconds to the MQTT client), and the MQTTEventCallback that will be delved into in this section.

This function is validated in the end with log messages detailing its success or failure.

```
MQTT_Init(&MQTTContext, &transportInterface, MQTTGetTimeMs,
    MQTTEventCallback, &fixedBuffer);
```

Listing 3.7: "*MQTT_Init*".

MQTTInitQoSLevels

This function is only responsible for calling the "*coreMQTT*"'s "*MQTTInitQoSLevels*" function and validating it. It initializes QoS levels for an MQTT session. It sets up the MQTT client's internal state to handle QoS levels for both outgoing and incoming publish messages.

MQTTEstablishConnection

The establish connection function is responsible for using the established socket connection to create a session with an MQTT broker. It sends the CONNECT packet, which contains crucial information, such as the client ID, and session parameters. It also expects to receive the CONNACK from the broker, confirming that the connection has been successful. This packet exchange is done by virtue of the transport interface defined in the "*MQTTInitializeConnection*" function. Figure 3.12 shows a flowchart that covers this function's logic in a simplified way.

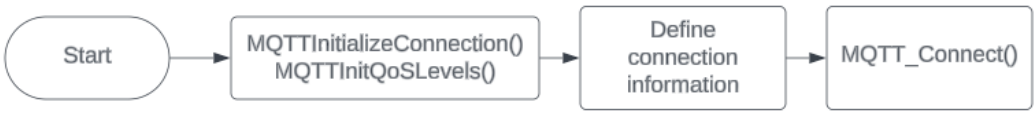


Figure 3.12: "*MQTTEstablishConnection*" Flowchart.

As the flowchart specifies, the "*MQTTEstablishConnection*" begins by running the two previous functions ("*MQTTInitializeConnection*" and "*MQTTInitQoSLevels*"). After this, connection information was defined as shown in Listing 3.8.

```
connectInfo.cleanSession = true;
connectInfo.pClientIdentifier = clientID;
connectInfo.clientIdentifierLength = strlen(clientID);
connectInfo.keepAliveSeconds = 60;
```

Listing 3.8: Connection Information.

This includes the "*cleanSession*" bool[4], the clientID (which must be unique), and the keep-alive period. This period acts as a mechanism that ensures the connection remains active. A PINGREQ packet is sent to the broker (at intervals defined by the keep-alive period), and if there is no response then the connection is considered as lost.

Lastly, the only thing remaining is executing "*coreMQTT*"'s "*MQTT_Connect*", as shown in Listing 3.9. Other than the context, and the connection information, there are a few more parameters involved in this function. The "NULL" is relative to the last will info, as it will not be used. The "100" refers to the time that the program will wait for the CONNACK packet, and the session [5].

```
1 MQTT_Connect(&MQTTContext, &connectInfo, NULL, 100, &
      sessionPresent);
```

Listing 3.9: "*MQTT_Init*"

The return of this function is validated and logs are printed regarding its success.

MQTTSubscribe & MQTTSubscribeToSingleTopic

These functions handle MQTT subscription attempts, retry failed subscriptions with the backoff algorithm [96], and process any received SUBACK messages. They are designed to handle potential failures and provide informative logs about the subscription process. The only difference between these two is the fact that one subscribes to multiple topics at once and the other only subscribes to a single topic. Figure 3.13 shows a flowchart that simplifies the work done by the *"MQTTSubscribeToSingleTopic"* function.

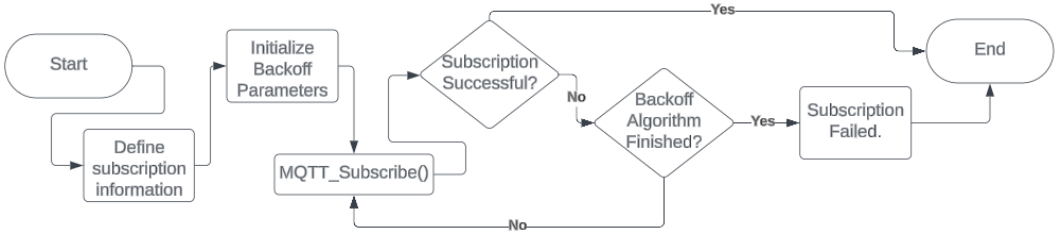


Figure 3.13: *"MQTTSubscribeToSingleTopic"* Flowchart.

As represented in the flowchart, first the information for the subscription request is formulated. This contains information about the QoS level to be applied, the name of the topic, and its length. After this, the backoff parameters are initialized.

[4]This indicates whether the broker should store information about the client's session when the client disconnects. If set to true, it does not remember

[5] *"sessionPresent* is a bool to determine if there was a previous session. It isn't used as a clean session was defined.

The backoff algorithm is used in network communication to handle retries in case of failures and congestion. The idea behind it is to gradually increase the time between successive retry attempts. This helps increase the chance of success as it reduces the load on the network. Then, the "*coreMQTT*"'s "*MQTT_Subscribe*" is executed, and the code expects to receive a SUBACK package. If this exchange works, and the subscription request is accepted, the function reaches its end. However, if it fails, the backoff algorithm will check if it has reached its limit of retries. If so, then a log detailing a subscription failure is printed. If not, then it will try to subscribe once again.

MQTTPublish

The implementation of an MQTT publishing function is relatively simple. All that is required is to fill in the publish information and to execute "*coreMQTT*"'s "*MQTT_Publish*", as Figure 3.14 indicates. Listing 3.10 shows the publish information code snippet.

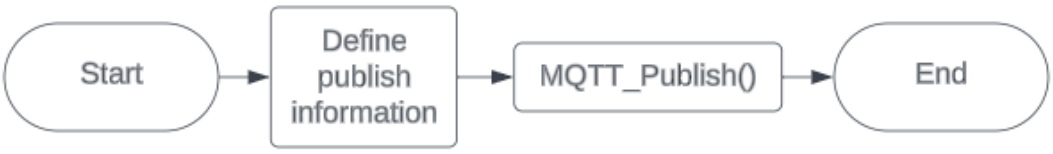


Figure 3.14: *"MQTTPublish"* Flowchart.

```
1 PublishInfoMQTT.qos = qos;
2 PublishInfoMQTT.retain = false;
3 PublishInfoMQTT.pTopicName = pTopic;
4 PublishInfoMQTT.topicNameLength = (uint16_t)strlen(pTopic);
5 PublishInfoMQTT.pPayload = pMessage;
6 PublishInfoMQTT.payloadLength = strlen(pMessage);
```

Listing 3.10: Publish Information.

As it shows, several parameters are required in order to perform an MQTT publish. Here, both the QoS, the topic, and the message are all passed as parameters. The "*retain*" flag is also needed, as it is a property of a published message that indicates whether the broker should retain the last message published on the topic for future subscribers. If the publish is not successful, a log alluding to that failure is printed.

MQTTProcessLoop & MQTTEventCallback

"MQTTProcessLoop" is a simple function that verifies the success of "*coreMQTT*"'s "*MQTT_ProcessLoop*". It is responsible for receiving packets from the transport

layer. The client continually listens for incoming messages from the broker. When a message is received, the function parses and processes it according to the MQTT protocol specifications, by signaling callback functions to execute. In this project, the logic in which this function is applied is showcased by Figure 3.15.

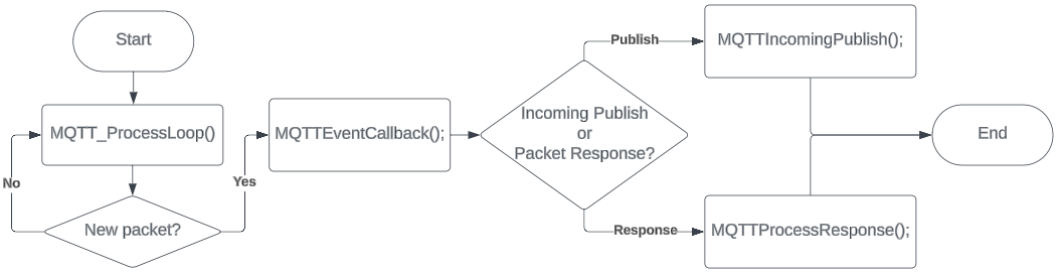


Figure 3.15: "*MQTT_ProcessLoop*" Logic.

As this diagram shows, it waits until there is a new packet in the transport layer, and once it is there, it activates the *"MQTTEventCallback"* function. This one will check if the packet is an incoming publish message (a message published to a topic the client is subscribed to) or a packet response (an acknowledgment packet for example).

A slightly different version of the *"MQTTProcessLoop"* was developed, in that it is given a timeout time: "*MQTTProcessLoppWithTimeout*".

MQTTProcessIncomingPublish & MQTTProcessResponse

This function is responsible for processing incoming publishes from topics that the client has subscribed to. Its work is represented in a simplified way in Figure 3.16. This is where developers can get the received message and perform actions with it (such as parsing it and tying its content with other functions).

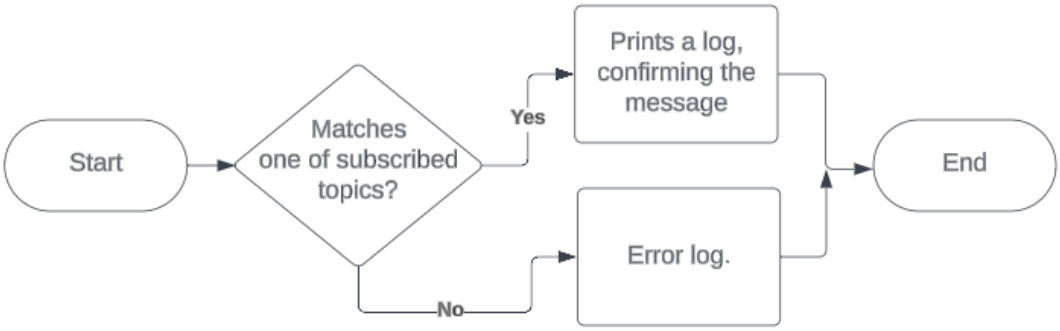


Figure 3.16: "*MQTTProcessIncomingPublish*" Flowchart.

The "*MQTTProcessResponse*" function is triggered every time an MQTT response packet is received, such as a PUBACK, SUBACK, etc. It handles these packets, accepting them and printing log messages for each of them. As an example, when it comes to SUBACK packets, it will run the "*MQTTSubAckStatus*" and

check if the acknowledgment was successful. The "*MQTTSubAckStatus*" parses the SUBACK and updates an array that saves the current state of the subscriptions.

Implementation Strategies

In FreeRTOS, code is structured into tasks that can execute concurrently, enabling efficient multitasking and real-time responsiveness. This environment facilitates the development of complex systems, by allowing different parts of the program to run independently, yet seamlessly interact with one another. In this section, the focus will be on the organization, functionality of the code created for this project, and what strategies were implemented to overcome certain challenges.

When reduced to its most basic form, the idea behind this project is to send the logs, generated by the Stratio Databox, to an MQTT Broker. With all the basic functions set in place, as explained in the previous chapter, this task becomes doable, but not without its challenges.

The first one to be noticed was one of recursion. Let's begin by understanding how log messages are printed. A log message is created, via a line of code similar to the one below:

$$LogInfo((”HelloWorld!”));$$

Here, "*LogInfo*" activates a macro that launches a function named *"vLogging-Printf"*. This function is responsible for sending this log to the UART so that it will appear on the serial line. The idea, in this project, is to also have it publish said message to the broker.

Now, lets imagine a system similar to Listing 3.11. Here once a log goes through *"vLoggingPrintf"*, it will immediately be published using "*MQTTPublish*".

```
1 void vLoggingPrintf (message){
2
3     char buffer[128] = message;
4     HAL_UART_Transmit (buffer);
5     MQTTPublish(message);
6 }
```

Listing 3.11: Publish Example (Pseudo-Code).

There is an intrinsic problem with publishing the message this way, which is the possibility of creating an infinite loop, as Figure 3.17 exemplifies.

A certain task has a log message to be sent. It will go through *"vLoggingPrintf"*, and then be published via "*MQTTPublish*". However, if "*MQTTPublish*" also wants to write a log, then when it reaches that log, the *"vLoggingPrintf"* will be called again. And this cycle goes on, as it will never be able to finish the publication.

Figure 3.17: Example of Recursion Problem.

The solution found for this problem, was the implementation of FreeRTOS Events [97] in addition to a buffer. Events allow tasks to wait for specific conditions or signals to occur before proceeding with their execution. These conditions can be set or cleared by other elements in the FreeRTOS system. As Figure 3.18 shows, when a log triggers *"vLoggingPrintf"*, it will be added to a buffer and will trigger an event notification. Another function waits for this notification, and when it is received, it will then publish the messages. The buffer is used to store all the logs that appear in the time between sending the notification and receiving it. This way, if "*MQTTPublish*" also wants to write a log, this log will be stored in the buffer until it is ready to be published.

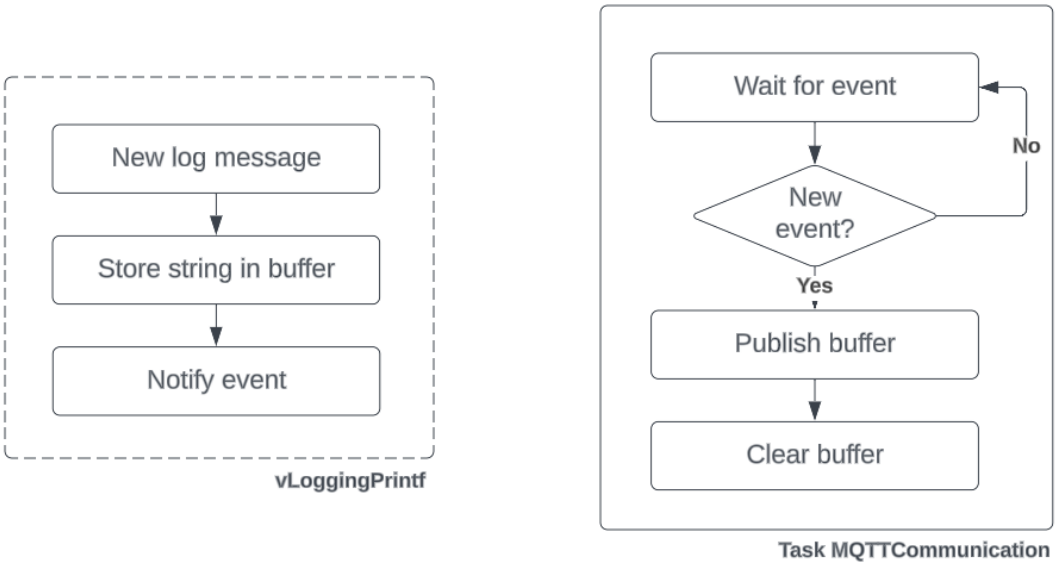


Figure 3.18: FreeRTOS Events Logic.

By implementing this method, the mentioned challenge is surpassed. However, not all problems are solved. Whilst the contents of the buffer are being published, which takes time, the buffer is essentially useless as it will not be able to store any more messages. Only when it is released by the *"MQTTPublish"* function, will it return to storing logs. This is a problem, as messages will be lost.

To solve this, a method named "double buffering" was implemented. It is commonly used in computer graphics, multimedia, and other real-time systems to improve performance. In this method, two buffers are created. Let's call them "buffer

1" and "buffer 2". The log messages are stored in "buffer 1", and once the event notification is received, and the publish is about to happen, the buffers switch. Now the one storing the logs is "buffer 2", and "buffer 1" can be published without fear of losing messages. This process, which repeats itself every time there is a publish, is represented in Figure 3.19.

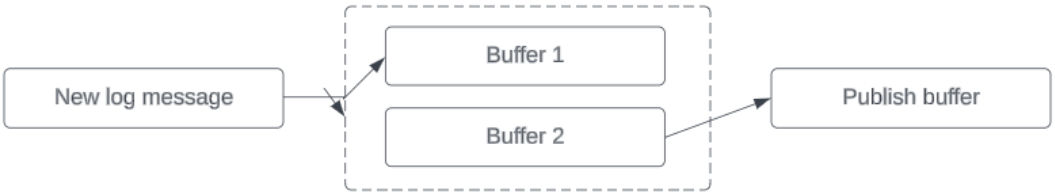


Figure 3.19: Double Buffering Approach.

With these approaches, the code is ready to withstand the listed complications. The following Listings show a more in-depth look at the actual code implemented. Listing 3.12 shows the snippet of the function responsible for notifying the event every time a message arrives (*"vLoggingPrintf"*).

```
1 if (connectDone)
2 {
3     logBuffer.writeToActiveBuffer(buffer);
4     notifyMqttEvent(MQTT_PUBLISH_BUFFER_FULL);
5 }
```

Listing 3.12: Notify an Event (*"vLoggingPrintf"*).

It first checks if there is an MQTT connection established, as it wouldn't be able to publish otherwise. Then it stores the message in the one buffer that is active (from the double buffering technique) and sends an event notification. Listing 3.13 shows the Event notification function.

```
1 void notifyMqttEvent(EventBits_t event)
2 {
3     xEventGroupSetBits(publishMqttEventGroup, event);
4 }
```

Listing 3.13: Inside the *"notifyMqttEvent"* Function.

Essentially, it sets the bit specified by *"MQTT_PUBLISH_BUFFER_FULL"*, which was passed as a parameter, of the created event group (*"publishMqttEvent-Group"*).

Lastly, a simplified version of the *"waitForMqttEvent"* function is shown on Listing 3.14.

```
EventBits_t waitForMqttEvent()
{
event = xEventGroupWaitBits(publishMqttEventGroup,
    MQTT_PUBLISH_BUFFER_FULL, pdFALSE, pdFALSE, pdMS_TO_TICKS
    (50000));

if ((event & (MQTT_PUBLISH_BUFFER_FULL)) == (
    MQTT_PUBLISH_BUFFER_FULL))
{
    logBuffer.switchActiveBuffer();

    result = mqttSession.MQTTPublish(logBuffer.getNonCurrentBuffer
    (), MQTT_TOPIC_DEMO1, MQTTQoS1);
    }

    logBuffer.clearBuffer(logBuffer.getNonCurrentBuffer());
    xEventGroupClearBits(publishMqttEventGroup,
    MQTT_PUBLISH_BUFFER_FULL);
}
```

Listing 3.14: Inside the *"waitForMqttEvent"* Function.

It starts by waiting for an update on the state of the wanted bit. After checking that it is set, the active buffer will switch (new logs will now be stored in the other buffer). The data is published, and after that, both the buffer and the event bits are cleared.

With this code infrastructure established, all that remains is to have a task that sets the overall workflow.

Tasks

For this project, there are two important tasks to consider. One that was already built, as part of Stratio's firmware, and another one created from scratch. The former (Message Task) is responsible for the printing of logs (it is the one that implements the *"vLoggingPrintf"* function). The latter (*"MQTTCommunication"* task) is the one responsible for establishing a connection, and placing *"waitForMqttEvent"* on a loop (as mentioned in Figure 3.18). Figure 3.20 shows a flowchart of the workflow of the *"MQTTCommunication"* task.

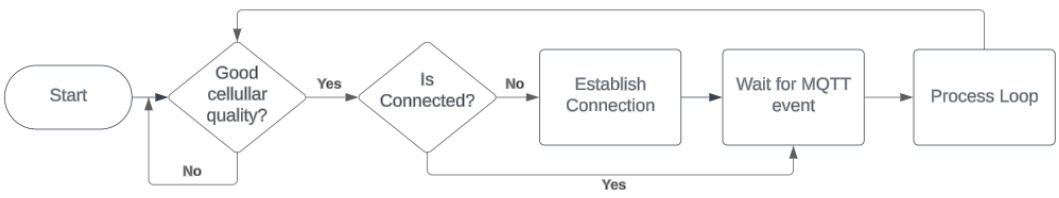


Figure 3.20: *"MQTTCommunication"* Task Flowchart.

It starts by checking whether there is enough signal quality to establish communication over the network. Once there is, a new MQTT connection is established (considering it wasn't established before). Lastly, both *"waitForMqttEvent"* and "*MQTTProcessLoop*" are triggered.

This way, the program connects to the network and keeps waiting for any event update (which means a new message or set of messages are ready to be published).

Publishing Messages

Once the program is working properly, the broker should be receiving messages such as the ones shown in Figure 3.21. Each published message contains multiple logs.

```
Message 45
[1374373] [DEBUG] [MAIN_TASKS] [MqttCommunication:567] Reusing network connection.
[1383821] [INFO] [CELLULAR_STATE] [updateSignalQuality:457] Setting signal quality [bars=2, dbms=-72]: 2
[1393878] [INFO] [CELLULAR_STATE] [updateSignalQuality:457] Setting signal quality [bars=2, dbms=-72]: 2
[1403933] [INFO] [CELLULAR_STATE] [updateSignalQuality:457] Setting signal quality [bars=2, dbms=-72]: 2
[1413988] [INFO] [CELLULAR_STATE] [updateSignalQuality:457] Setting signal quality [bars=2, dbms=-72]: 2
[1414866] [DEBUG] [TCP_Socket_Utils] [tcpSocketConnect:59] (Re)-using socket to communicate. Net Type: 8
[1414869] [INFO] [MQTT_SESSION] [waitForMqttEvent:37] Handling MQTT event: 1

Message 46
[1416470] [INFO] [APICOMM_STRATIO] [SendMeasurementsToAPI:1701] Sending a compressed bin measurements request to the API
[1417006] [DEBUG] [TCP_Socket_Utils] [tcpSocketConnect:59] (Re)-using socket to communicate. Net Type: 8
[1424044] [INFO] [CELLULAR_STATE] [updateSignalQuality:457] Setting signal quality [bars=2, dbms=-72]: 2
[1434099] [INFO] [CELLULAR_STATE] [updateSignalQuality:457] Setting signal quality [bars=2, dbms=-72]: 2
[1437009] [INFO] [TCP_WRAP] [prvNetworkRecvCellular:293] prvNetworkRecv timeout
[1437017] [DEBUG] [MAIN_TASKS] [MqttCommunication:567] Reusing network connection.
[1437020] [INFO] [MQTT_SESSION] [waitForMqttEvent:37] Handling MQTT event: 1

Message 47
[1437614] [DEBUG] [MAIN_TASKS] [MqttCommunication:567] Reusing network connection.
[1444155] [INFO] [CELLULAR_STATE] [updateSignalQuality:457] Setting signal quality [bars=2, dbms=-72]: 2
[1454211] [INFO] [CELLULAR_STATE] [updateSignalQuality:457] Setting signal quality [bars=2, dbms=-72]: 2
[1464269] [INFO] [CELLULAR_STATE] [updateSignalQuality:457] Setting signal quality [bars=2, dbms=-72]: 2
[1474327] [INFO] [CELLULAR_STATE] [updateSignalQuality:457] Setting signal quality [bars=2, dbms=-72]: 2
[1478388] [DEBUG] [TCP_Socket_Utils] [tcpSocketConnect:59] (Re)-using socket to communicate. Net Type: 8
[1478392] [INFO] [MQTT_SESSION] [waitForMqttEvent:37] Handling MQTT event: 1
```

Figure 3.21: Published Messages Example.

3.4 Software Validation

Testing is an important part of the CI/CD Pipeline - a series of steps that ought to be performed before the release of a new version of the software [98]. It helps streamline the process of delivering code changes into production environments.

It is a good policy to perform a good variety of tests, trying to cover every possible scenario so that errors and bugs can be minimized. In this project, two types of tests were performed to make the firmware as robust and reliable as possible:

- Component Tests;
- Stress Tests;

3.4.1 Component Testing

Component tests are essential for software validation. They assert that individual components of a given system are working as intended. This section explores the way these tests were implemented in this project.

One important thing to consider is that these tests are running in a Software In the Loop (SIL) environment. This means that they are happening in a simulated environment, instead of being run in the device itself. This is very helpful as it eliminates any problems involved with the device, such as malfunctioning, as well as outside factors, like an unreliable internet connection. The simulated environment simulates the hardware's behavior and its interaction with the surrounding environment. Moreover, SIL can help reduce production costs and improve efficiency, as developers can test features without even having the hardware.

Multiple tests were developed for this project, and Table 3.1 lists them all.

Table 3.1: Performed Component Tests.

Test Name	Description
ShouldInitialize	Asserts if the MQTT initialization works correctly.
ShouldInitializeQoSLevels	Asserts if multiple QoS Levels acceptance is working.
ShouldConnect	Asserts if the connection is established (assuming correct parameters).
ShouldNotConnectBadURL	When given an incorrect URL, ensure the connection fails and reports the problem.
ShouldPublish	Asserts if the publication is performed successfully.
ShouldSubscribe	Asserts if the subscription to multiple topics is performed successfully.
ShouldSubscribeToSingleTopic	Asserts if the subscription to a single topic is performed successfully.
ShouldSubscribeAndRead	Asserts if the system can receive incoming publications.

Before the tests are performed, multiple objects and parameters must be initialized, such as the emulated modem and the network. These tests use the very same functions the main program does, but do it so, expecting a confirmation that such a function worked well. Below, two examples are shown of two of the different tests: "ShouldPublish" and "ShouldNotConnectBadURL". Listing 3.15 showcases the former, which is responsible for ensuring that the publishing of a message is successful.

```
1 TEST_F(MqttSessionTest, ShouldPublish)
2 {
```

```
3       MQTTStatus_t result = MQTTSuccess;
4
5       mqttSession.MQTTEstablishConnection(testUrl, false);
6
7       result = mqttSession.MQTTPublish("Should Publish QoS1 Test
        Message.", MQTT_TOPIC_TESTS, MQTTQoS1);
8       ASSERT_EQ(result, MQTTSuccess) << "Mqtt publish QoS1 should
        succeed";
9   }
```

Listing 3.15: Publish Test.

As the Listing shows, at the end of the test, there is an assertion checking whether the "*result*" variable is equivalent to "*MQTTSuccess*". If it is, then it means that the test has succeeded. It if doesn't, something is inherently wrong. Figure 3.22 shows the result of this test, including the log messages and the assertion result. It also shows the time that the test took, which is useful in more complex tests, as it helps the developer make their code more efficient, reducing the runtime.

```
[18939] [DEBUG] [MQTT SESSION] [MQTTInitializeConnection:78] Setting up transport interface.
[18939] [DEBUG] [MQTT SESSION] [MQTTInitializeConnection:85] Allocating Buffer.
[18939] [DEBUG] [MQTT SESSION] [MQTTInitializeConnection:99] Preforming MQTT Init
[18940] [INFO] [MQTT SESSION] [MQTTInitializeConnection:109] Mqtt Init Success!
[18940] [DEBUG] [MQTT SESSION] [MQTTEstablishConnection:141] Start Init Qos Levels
[18940] [DEBUG] [MQTT SESSION] [MQTTInitQoSLevels:118] Preforming MQTT InitStatefulQos.
[18940] [INFO] [MQTT SESSION] [MQTTInitQoSLevels:128] Mqtt InitStatefulQoS Success!
[18940] [DEBUG] [MQTT SESSION] [MQTTEstablishConnection:148] Setting up Connect Info.
[18940] [DEBUG] [MQTT SESSION] [MQTTEstablishConnection:158] Preforming MQTT Connect
[19004] [INFO] [MQTT SESSION] [MQTTEstablishConnection:168] MQTT Connection Successeful.
      OK ] MqttSessionTest.ShouldPublish (1256 ms)
```

Figure 3.22: Publish Test Result.

It is also a good practice to do negative tests. In other words, perform tests that are intended to report a failure. It can help reveal edge cases and understand unexpected behavior. Listing 3.16 showcases the second previously mentioned case, where the URL was purposefully wrong, hoping that the test returns a failure.

```
1   TEST_F(MqttSessionTest, ShouldNotConnectBadURL)
2   {
3       MQTTStatus_t result = MQTTSuccess;
4
5       result = mqttSession.MQTTEstablishConnection("mqtt://mosquitto
        ");
6       ASSERT_NE(result, MQTTSuccess) << "Mqtt connection should
        report a failure as expected";
7   }
```

Listing 3.16: Bad URL Test.

Figure 3.23 depicts the result of this test. As shown, the MQTT Connection has failed, however, the test was successful, as intended.

```
[19327] [DEBUG] [MQTT SESSION] [MQTTInitializeConnection:78] Setting up transport interface.
[19327] [DEBUG] [MQTT SESSION] [MQTTInitializeConnection:85] Allocating Buffer.
[19327] [DEBUG] [MQTT SESSION] [MQTTInitializeConnection:99] Preforming MQTT Init
[19327] [INFO] [MQTT SESSION] [MQTTInitializeConnection:109] Mqtt Init Success!
[19328] [DEBUG] [MQTT SESSION] [MQTTEstablishConnection:141] Start Init Qos Levels
[19328] [DEBUG] [MQTT SESSION] [MQTTInitQoSLevels:118] Preforming MQTT InitStatefulQos.
[19328] [INFO] [MQTT SESSION] [MQTTInitQoSLevels:128] Mqtt InitStatefulQoS Success!
[19328] [DEBUG] [MQTT SESSION] [MQTTEstablishConnection:148] Setting up Connect Info.
[19328] [DEBUG] [MQTT SESSION] [MQTTEstablishConnection:158] Preforming MQTT Connect
[39370] [INFO] [TCP WRAP] [prvNetworkRecvCellular:293] prvNetworkRecv timeout
[39370] [ERROR] [MQTT SESSION] [MQTTEstablishConnection:163] MQTT Connection Failed.
      OK ] MqttSessionTest.ShouldNotConnectBadURL (20747 ms)
```

Figure 3.23: Bad URL Test Result.

Component testing successfully helped improve code quality and assured a higher confidence in the end product, being a key step in the development of this project.

3.4.2 Stress Testing

Stress testing this system is intended to test the robustness and resilience of the system. These tests are relevant, as they allow the verification of how the system operates over long periods of time or under less-than-ideal conditions. These stress tests also help to predict how the system will fare in real-world conditions.

The main conducted test was one of prolonged execution. Here, the program was continuously executed during a selected time interval, or until a specified number of publications were done. During said period, data about the transmitted information was gathered to help understand the system's stability, reliability, and performance. Tests were conducted in different scenarios: varied signal strength and message size. These variations are essential to understand how different factors can influence the general system.

More details of these tests are given in Chapter 4, as the findings are discussed in the "Results" chapter of this report.

3.5 Python Analysis Script

After the entire system is working, it is important to gather practical information on its performance. This information will be analyzed in Section 4, but a script must be developed in order to obtain said data.

Stratio's device is now publishing every log to a topic in the "Mosquitto" Broker. A Python script was developed to subscribe to that topic and receive every message that is being published. Data from every received publish is then stored in a ".txt" file in JSON format. This data is processed afterward to create charts and give a more in-depth look at the actual metrics of the communication. Figure 3.24 shows a flowchart of the script workflow.

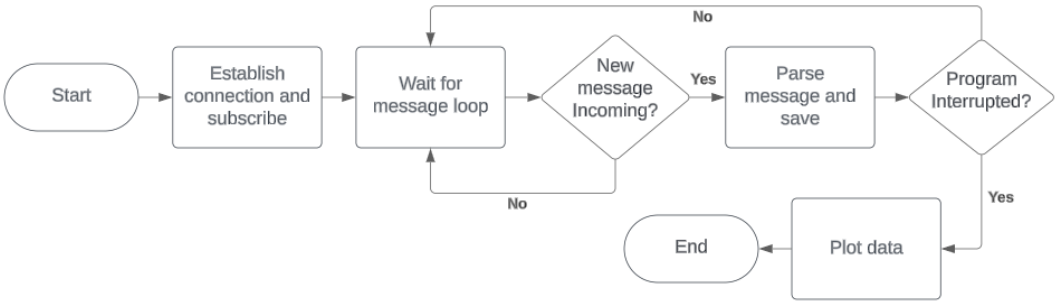


Figure 3.24: Python Analysis Script Flowchart.

In the beginning, the script starts by establishing a connection to the MQTT broker and subscribing to a topic. This was done using Python's "*paho*" library [99], and Listing 3.17 shows a snippet of this.

```
def on_message(client, userdata, msg):
    message = msg.payload.decode()
    parse_message(message)

if client.connect("192.168.1.253", 1883, 60) != 0:
    print("Could not connect to MQTT Broker!")
    sys.exit(-1)

try:
    client.loop_forever()
except KeyboardInterrupt:
    plot_data()
    client.disconnect()
```

Listing 3.17: Establishing MQTT Connection in Python.

As shown, every new message that arrives has its contents decoded and handed, as a parameter, to a function named "*parse_message(message)*". This one is responsible for gathering the information that the message holds and storing it. Figure 3.25 shows an example of two messages received by the Python program. The separation via a blank space means they are from 2 different publishes. Here is possible to check that each publication contains several individual logs.

```
[43106] [ERROR] [APICOMM_STRATIO] [PostJson:2201] PostJson failed: 1. Won't retry
[43106] [ERROR] [APICOMM_STRATIO] [SendMeasurementsToAPI:1722] Unable to establish a connection to the API
[43108] [INFO] [LOGGING] [SetLastPostError:1037] Reverting circular buffer read
[43109] [INFO] [MQTT_SESSION] [waitForMqttEvent:37] Handling MQTT event: 1

[43837] [DEBUG] [MAIN_TASKS] [MqttCommunication:567] Reusing network connection.
[48101] [INFO] [CELLULAR_STATE] [updateSignalQuality:457] Setting signal quality [bars=1, dbms=-80]: 1
[48104] [INFO] [MQTT_SESSION] [waitForMqttEvent:37] Handling MQTT event: 1
```

Figure 3.25: Example of Published Message.

Five different parameters are being stored, some of which can be directly extracted from the message payload, while others can't. These are:

- **Message Size**: This is directly extracted from the message, in bytes.
- **Message ID**: Whenever the device generates a new log, it comes with a number. For example, if the number is "[43105]" (as the first log in Figure 3.25 is), it means that 43.105 seconds have passed since Stratio's device program started. In this script, this value is used as a message identifier, as each message will have a different value.
- **Log Initial Timestamp**: This value is stored every time the device generates each new log. As an example, each singular message present in Figure 3.25 has its own timestamp stored (for a total of 7 timestamps for those 2 published groups - 1 per log). The reason this value is saved is to know at what time the log left Stratio's device.
- **Message Arrival Timestamp**: This value is stored every time there is a new incoming publish to the Python client. For example, in Figure 3.25 there are two timestamps that are stored, one for each message. The reason this value is saved is to know at what time the message reached the Python client.
- **Message Delta Time**: This is simply the difference between the message arrival timestamp and the message initial timestamp. This is calculated to figure out the time each publish took to leave Stratio's device and reach the Python client.

Figure 3.26 presents a scheme detailing the location/state of the program where each one of these values is gathered.

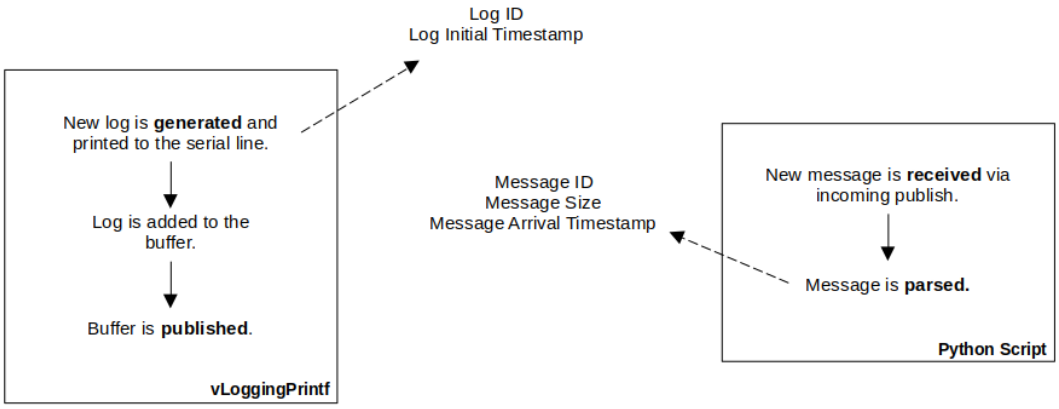


Figure 3.26: Where Data is Gathered.

Chapter 4

Results

In this chapter, the focus will be the display of results gathered by the tests that were performed. These results are highly important to understand the strengths and weaknesses of the system, to confirm good behavior, and to stress test the system. The chapter aims to showcase the main findings, and analyze and interpret the achieved results, thereby completing the project. Furthermore, the limitations found for this project are also covered.

4.1 Tests

Each set of data is accompanied by a set of graphs/tables that give a visual interpretation of the results, and facilitate comprehension.

When it comes to testing, it is crucial to change certain variables in order to have diverse and representative results. The main variables that were actively changed from test to test were:

- **Message Size**: The message size is an important variable, as it can directly affect the system's performance. For these tests, messages with 4 different sizes were deployed. Messages could contain a minimum of 3, 5, 7, and 9 logs. This is useful to understand the impact of the size when transmitting the message through the network. This impact can be translated into areas such as latency and throughput.

- **Signal Strength**: Signal strength is another important metric when it comes to transmitting data in IoT. Stronger signals typically result in faster message transmission and smaller number of lost packets, while weaker signals can be responsible for delays and faults in communication. Testing was done using a signal strength of 2 bars and a signal strength of 3 bars.

Different tests were performed to analyze different metrics, with the goal of understanding what can be done to improve performance. These tests aim to evaluate the efficiency of the selected variables and how they behave in relation to each other.

4.1.1 Transmission Rate per Time Interval

In this test, the evaluation focus was the number of messages which were sent in 30-minute intervals, for different message sizes. It aims to determine if a higher number of messages sent in a given time period is advantageous or not. The analysis was done by assessing the number of publishes made to the broker in a 30-minute period.

The results are shown in Figure 4.1. They show that there is a clear tendency for the number of messages sent to be lower the bigger the packets are. This is completely expected, as bigger messages accumulate more logs, and therefore fewer are needed to transmit all the information. These results alone aren't very valuable, as one could already guess the outcome. However, when coupled with the next test, results get increasingly interesting.

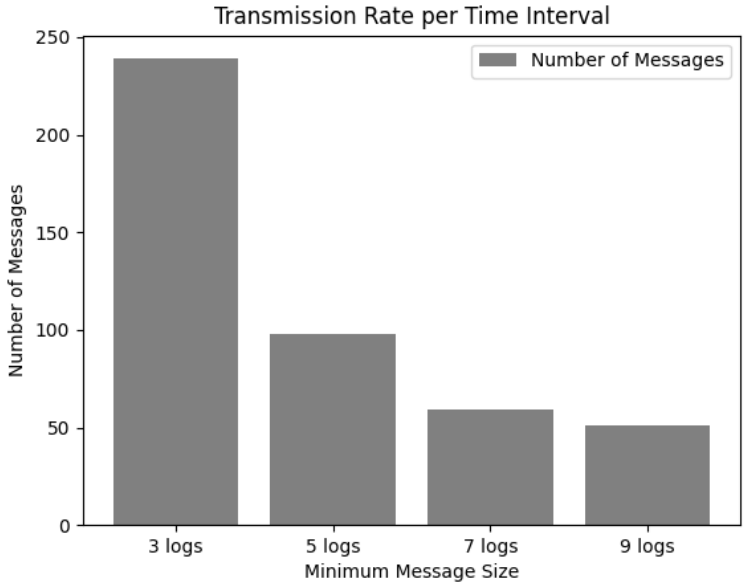


Figure 4.1: Transmission Rate per Time Interval.

4.1.2 Message Size Impact on Delay

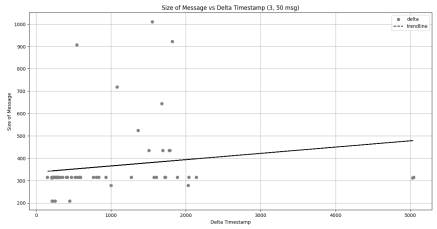
This test aims to evaluate the time it takes for a message to leave the device and to reach a client. This is achieved by calculating the time difference between the message being sent to the serial line (Log Initial Timestamp) and it reaching a client (Message Arrival Timestamp).

Bigger messages, containing more logs, may take more time to be transmitted and can present bigger fluctuations (meaning more inconsistent), especially in limited resources environments. On the other hand, typically smaller messages are faster to reach their destination. They demand, however, an increased number of transmissions to send the same amount of data, which can put pressure on the network.

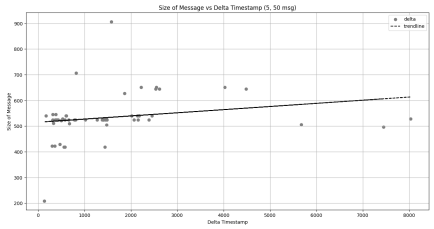
It is worth noting that this test was done for all message sizes, by using a 50-message sample for each size, and with a signal quality of 2 bars.

Overall, this test allowed to access different performance patterns related to the message size.

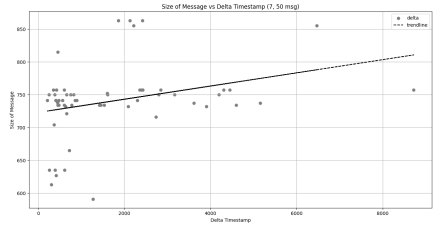
Figure 4.2 shows four graphics detailing the size/delta time relation for different message sizes.



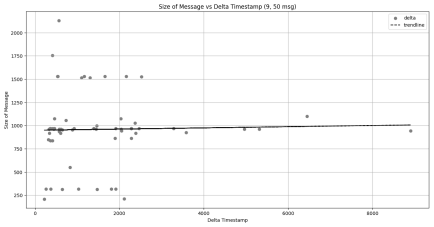
(a) Message with Minimum 3 Logs.



(b) Message with Minimum 5 Logs.



(c) Message with Minimum 7 Logs.



(d) Message with Minimum 9 Logs.

Figure 4.2: Message Size Impact on Delay (2-bar Signal Strength).

A thorough analysis of the graphics shows that the time it takes to send a message directly correlates to its size, in the context of a 2-bar quality signal. Additionally, the results appear to be consistent among the different message sizes excluding the one with a minimum of 9 logs, which appears to be an outlier (despite also having a slight positive slope, it isn't as noticeable as the others).

This kind of graphic is really useful for understanding the correlation between two variables. And in this case, that objective was succeeded. However, being the relation between size and delay as important as it is, there is another metric that is quite relevant. The difference in the average transmission delay of the different test cases.

With the results of the first test, one can figure out what message size might be the most efficient, being in terms of message flow, in terms of delay, or, most importantly, both. It is crucial to find the right balance between these two aspects to further improve the whole project.

Figure 4.3 is a box plot graph detailing this exact relation. These graphics show the range of values that a data set contains, making it a really good tool to analyze the consistency of a system. A bigger "box" means that the system is not as consistent in that circumstance and vice-versa. In this case, consistency refers to the system's ability to minimize variation in transmission delays.

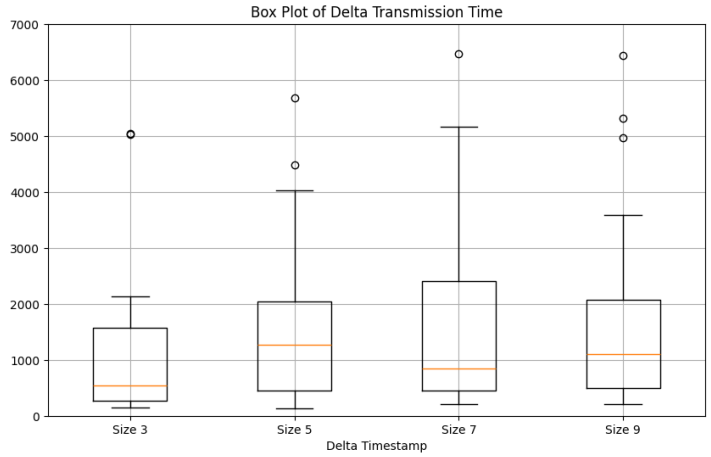


Figure 4.3: Box Plot of Delta Transmission Time.

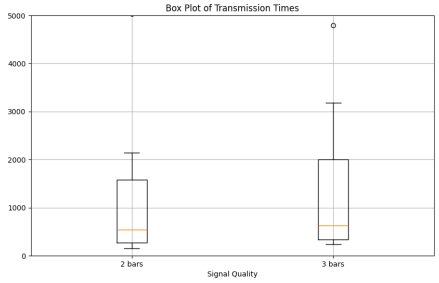
The results shown in the box plot seem to indicate that apart from the last test (which seems to be an outlier), there is a tendency that relates a lack of consistency with the size of the message. Even taking into account the possible outlier ("Size 9"), it still is considerably less consistent than the "Size 3" test.

Both this and the previous test (Section 4.1.1) help to understand that to have a system working in the most efficient way, a balance is required. The very word "efficient" can be subjective in this case, as it will depend on the developer's favored characteristics. For a system working with a signal strength of 2 bars, a bigger message might mean more inconsistent results, as well as more delayed messages, but has the upside of not having as much traffic on the network (as Figure 4.1 depicts). A good balance would probably be the usage of "Size 5" messages, as it encompasses somewhat of a middle ground between these characteristics.

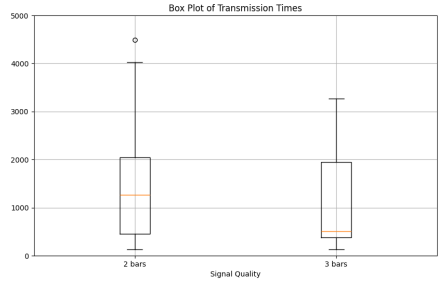
4.1.3 Performance in Different Signal Strengths

Signal quality is an important aspect of a wireless network's performance. It directly affects the stability of the connection and the transmission rate. In this test, the focus was the possible difference between the time a message takes to be transmitted in different signal strength conditions. Up until now, all the tests were performed in an average 2-bar signal quality environment. This test intends to evaluate the differences between the results when the environment is of a signal quality of 3 bars, with the objective of finding out if adjustments should be made to accommodate these differences.

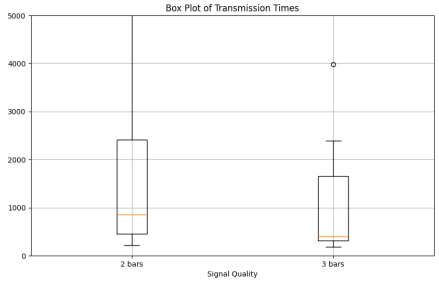
Figure 4.4 showcases four different graphs detailing the relation between the time a message takes to reach the client, and the signal quality. Each of the plots for the 2-bar signal strength is the same as the ones presented in Figure 4.3. When one compares these to the 3-bar signal strength, an interesting result is evident. The progression of the consistency is somewhat inverted. While in the 2-bar signal strength, as mentioned in the previous test, the consistency decreases the bigger the packets get, in the 3-bar signal quality it increases. Not only that but also the median times decrease with the bigger packets. This means, that on average, bigger packets are transmitted faster than smaller ones.



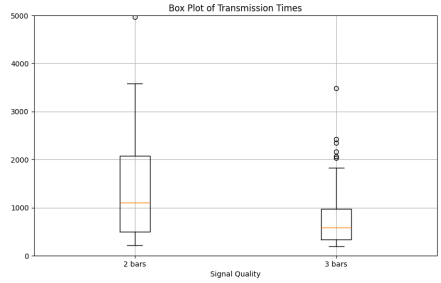
(a) Message with Minimum 3 Logs.



(b) Message with Minimum 5 Logs.



(c) Message with Minimum 7 Logs.



(d) Message with Minimum 9 Logs.

Figure 4.4: Time Variation for Different Signal Qualities.

This dichotomy is further evident in Table 4.1. The data suggests that higher signal strength (3-bar) tends to improve both the speed and consistency of message

transmission. For smaller message sizes (3 logs), there might be an initial increase in delta with higher signal strength, but as the message size increases, the transmission time decreases significantly in a 3-bar environment compared to a 2-bar environment. The standard deviation also consistently decreases with higher signal strength, indicating more predictable and stable performance.

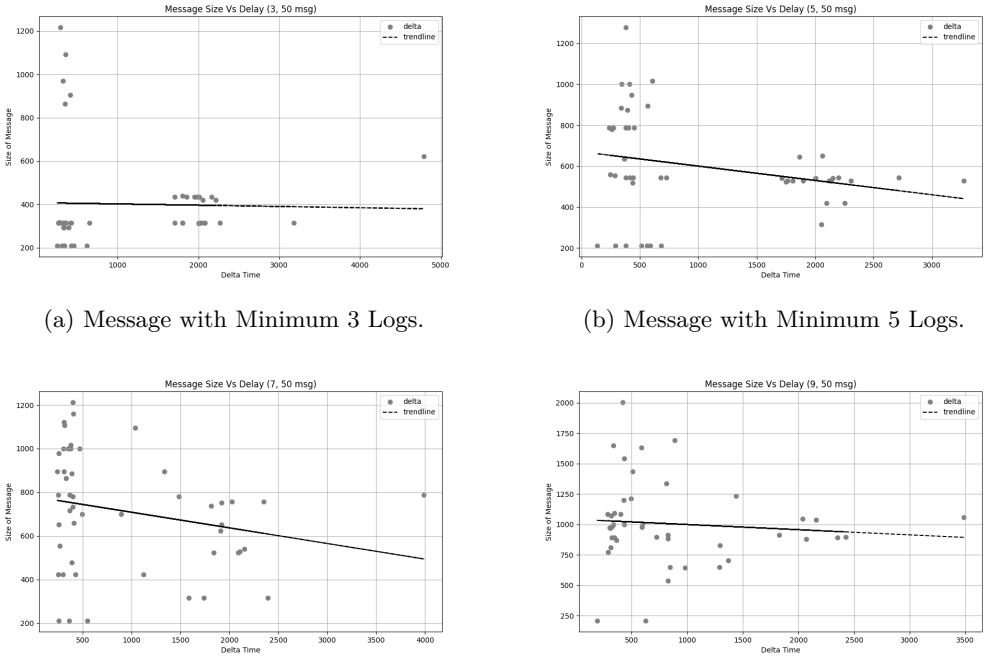
Table 4.1: Table Detailing the Measured Time Values.

Size	Value (ms)	2-bar	3-bar
3 logs	delta	997.041	1187.083
	std	1058.595	1006.841
5 logs	delta	1434.02	1043.469
	std	1394.404	866.243
7 logs	delta	1820.612	942.51
	std	1839.318	857.302
9 logs	delta	1658.72	858.356
	std	1695.874	739.938

These results can be explained by a few reasons. However, it is hard to assess exactly which is the differentiating factor. It can have to do with the way the network manages packets. Different networks might have different policies for traffic prioritization and scheduling. In some cases, networks may give priority to certain types of traffic or packets based on their size or importance. A 3-bar environment might benefit from more efficient scheduling and prioritization mechanisms, leading to better performance for larger packets. It is worth noting that tests were taken in two different locations, and therefore, the network rules/handling of the packets might be different. Different latency and throughput conditions coupled with a different way to prioritize packages (for example) can explain these interesting results. Environmental factors may also contribute to this difference. A weaker signal is also more prone to suffering interference than a stronger one. Other wireless devices, physical obstructions, and distance to the access point also can affect these values.

One other way to understand the relation between message size and the time it takes to reach the client is a similar evaluation to the one in Section 4.1.2 (Figure 4.2)). Figure 4.5 unveils the same comparison but for a 3-bar signal quality environment.

These graphics confirm the assessment that bigger packages are prioritized in the 3-bar signal strength environment, as the slope is negative.



(a) Message with Minimum 3 Logs.

(b) Message with Minimum 5 Logs.

(c) Message with Minimum 7 Logs.

(d) Message with Minimum 9 Logs.

Figure 4.5: Message Size Impact on Delay (3-bar Signal Strength).

4.2 Final Considerations

The results gathered from the several tests done, give an overarching view of message transmission behavior in different conditions and help conclude possible ways of improving the system. The comparative analysis of signal strengths (2 bars and 3 bars) highlights its significant impact on the system performance (speed and consistency). In stronger signal environments, larger messages seem to be transmitted faster and with a bigger system stability/consistency, which is positive, as the system is more predictable. This suggests that the optimization of the networks can be an important step in improving communication.

These results show that the system could be made more adaptable to different network situations. Dynamic sizing would be a positive change in that direction. Assuring that, for example, when the signal strength is lower, the messages sent are of a smaller size (each one carrying fewer logs), suits well with the fact that these smaller messages seem to be faster in this environment. The opposite could be done for better signal networks, where the packets sent should be bigger, as these are transmitted faster in said conditions.

Nevertheless, this approach (Dynamic Sizing) should take into account the fact that the size of the packets is a burden on the network, and a compromise should be made. While it is true that in conditions of lower signal strength 3-log messages are the faster ones to be transmitted, this can have an effect of overcrowding the network.

A decision would have to be made between what trait should be prioritized: speed and consistency, or assuring a less crowded network. The same rationale should be had relating to larger packet sizes in 3-bar networks.

With the goal of ensuring maximized efficiency and reliability, robust performance in varied environments is required. These test results not only confirm some of the initial assumptions but also lay the ground for future improvements and optimizations for this wireless communication.

Chapter 5

Conclusions

The main goal of this project was to create an efficient system that allows the transmission of messages using the MQTT protocol. This is required to improve the workflow and supervision capabilities of Stratio's device, as with such a system, developers can more easily troubleshoot, and understand the device's work cycle. The theoretical basis of the work was established through a solid literary review, hoarding topics such as the IoT and its security measures, different communication protocols, different RTOS's, software validation techniques, and more.

To build this project, several steps had to be made, starting with the implementation and validation of the necessary hardware and firmware. To assure the robustness of the system, a CI/CD pipeline was implemented. This allowed a continuous validation of the software and the fast identification of any existing problems. On top of this, several Python scripts were implemented to gather the results, facilitating the analysis of the project as a whole.

The pilot project was crucial to determine the viability of using the chosen hardware in a real-life scenario. It also served as a learning/familiarization project, that helped immensely in the development of the main project.

Upon a close evaluation of the results, the dynamic sizing approach was considered a potential way to improve the performance of the system. Adjusting message size to attenuate a difference in signal strength seems to be a solid method of reducing its impact. However, this approach should consider loading a large number of small packets on the network, as well as the loading big packets might have, and it

is important to find an equilibrium between speed, consistency, and the load on the network.

Lastly, it is important to consider that a system like this, which is connected to the internet, might be vulnerable to attacks such as the ones stated in Chapter 2.1. A DDoS attack can have a big impact on the system, as it can lead to a temporary loss of access of the gathered data. Furthermore, the existence of open ports can facilitate MitM attacks, where an attacker intercepts and potentially tampers the communication between the databox and the server. The consequences of these kinds of attacks might be more costly than just loss of information. They can affect the integrity and reliability of the system, and thus it is important to implement preemptive security measures (such as SSL), to mitigate the risks, and assure data protection.

Overall the project was completed and the objectives were achieved.

5.1 Future Work

Despite the gathered results from the tests being promising, there are several areas for future improvements. Firstly, as aforementioned, the implementation of a dynamic sizing technique could help balance the downsides of different signal strengths, improving efficiency and adaptability. Furthermore, data compression techniques could also have a big impact on the transmission times and would be a welcomed improvement.

Another good feature to add to this project would be a system where the user can subscribe to a certain topic in order to receive commands. For example, if developers only wanted one type of log to be sent, or wanted to manually command the message sizing, a publication could be done to a topic to which the device subscribes.

Efforts can be made to further improve the firmware, such as making it more modular and adaptable to different circumstances. Also, as some of the information being transmitted might be sensitive, it is important to take security measures to overcome some of the liabilities explored in this project. More tests could be written to help in future problem-solving as well.

References

[1] I. Analytics, "Iot connections market update." Available at `https://iot-analytics.com/number-connected-iot-devices/`, May 2023. (Last accessed in 25/10/2023). [Cited on pages 1 and 7]

[2] "Stratio oficial webpage." Available at `https://stratioautomotive.com/pt-pt/recolha-de-dados/`. (Last accessed in 05/19/2024). [Cited on pages 1 and 3]

[3] M. N. Rajkumar, "A survey on latest dos attacks: classification and defense mechanisms," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 1, no. 8, pp. 1847–1860, 2013. [Cited on page 8]

[4] A. Jurcut, T. Niculcea, P. Ranaweera, and N.-A. Le-Khac, "Security considerations for internet of things: A survey," *SN Computer Science*, vol. 1, pp. 1–19, 2020. [Cited on pages 8 and 10]

[5] A. Munshi, N. A. Alqarni, and N. A. Almalki, "Ddos attack on iot devices," in *2020 3rd International Conference on Computer Applications & Information Security (ICCAIS)*, pp. 1–5, IEEE, 2020. [Cited on page 8]

[6] J. Mirkovic and P. Reiher, "A taxonomy of ddos attack and ddos defense mechanisms," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 39–53, 2004. [Cited on page 8]

[7] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, *et al.*, "Understanding the mirai botnet," in *26th USENIX security symposium (USENIX Security 17)*, pp. 1093–1110, 2017. [Cited on pages 8 and 9]

[8] Z. Ahmed, S. M. Danish, H. K. Qureshi, and M. Lestas, "Protecting iots from mirai botnet attacks using blockchains," in *2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pp. 1–6, IEEE, 2019. [Cited on page 9]

[9] R. Vishwakarma and A. K. Jain, "A honeypot with machine learning based detection framework for defending iot based botnet ddos attacks," in *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pp. 1019–1024, IEEE, 2019. [Cited on page 9]

[10] NIST, "Computer security resource center." Available at `https://csrc.nist.gov/glossary/term/advanced_persistent_threat`. (Last accessed in 30/10/2023). [Cited on page 9]

[11] L.-X. Yang, K. Huang, X. Yang, Y. Zhang, Y. Xiang, and Y. Y. Tang, "Defense against advanced persistent threat through data backup and recovery," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 3, pp. 2001–2013, 2020. [Cited on page 9]

[12] P. Paganini, "European and russian interests." Available at `https://www.cyberdefensemagazine.com/nettraveler-apt-still-targets-european-and-russian-interests/`. (Last accessed in 11/06/2023). [Cited on page 10]

[13] G. Research and A. Team, "The nettraveler (aka 'travnet')." Available at `https://d2538mqrb7brka.cloudfront.net/wp-content/uploads/sites/43/2018/03/20134120/kaspersky-the-net-traveler-part1-final.pdf`. (Last accessed in 11/06/2023). [Cited on page 10]

[14] J. O. Agyemang, J. J. Kponyo, and I. Acquah, "Lightweight man-in-the-middle (mitm) detection and defense algorithm for wifi-enabled internet of things (iot) gateways," *Information Security and Computer Fraud*, vol. 7, no. 1, pp. 1–6, 2019. [Cited on page 10]

[15] S. Y. Nam, D. Kim, and J. Kim, "Enhanced arp: preventing arp poisoning-based man-in-the-middle attacks," *IEEE communications letters*, vol. 14, no. 2, pp. 187–189, 2010. [Cited on page 10]

[16] A. A. R. Said, "Securing the iot: Defending against man-in-the-middle attacks." Available at `https://www.linkedin.com/pulse/securing-iot-defending-against-man-in-the-middle-refaat-said-dduwf/`. (Last accessed in 02/18/2024). [Cited on page 11]

[17] "Man in the middle (mitm) attacks." Available at `https://www.rapid7.com/fundamentals/man-in-the-middle-attacks/`. (Last accessed in 02/18/2024). [Cited on page 11]

[18] I. Tudosa, F. Picariello, E. Balestrieri, L. De Vito, and F. Lamonaca, "Hardware security in iot era: The role of measurements and instrumentation," in *2019 II Workshop on Metrology for Industry 4.0 and IoT (MetroInd4. 0&IoT)*, pp. 285–290, IEEE, 2019. [Cited on page 11]

[19] B. Sim and D. Han, "A study on the side-channel analysis trends for application to iot devices," *J. Internet Serv. Inf. Secur*, vol. 10, pp. 2–21, 2020. [Cited on page 11]

[20] J. Fan, X. Guo, E. De Mulder, P. Schaumont, B. Preneel, and I. Verbauwhede, "State-of-the-art of secure ecc implementations: a survey on known side-channel attacks and countermeasures," in *2010 IEEE international symposium on hardware-oriented security and trust (HOST)*, pp. 76–87, IEEE, 2010. [Cited on page 11]

[21] "Mqtt version 3.1.1." Available at `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`. (Last accessed in 04/02/2024). [Cited on page 12]

[22] G. Cherradi, A. El Bouziri, and A. Boulmakoul, "Smart data collection based on iot protocols," *JDSI*, vol. 16, pp. 2509–2103, 2016. [Cited on pages ix and 13]

[23] Y. Shinji and K. Shiomoto, "Qos in iot-based river flood monitoring system using mqtt brokers in tandem connection," in *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)*, pp. 1–2, IEEE, 2022. [Cited on page 13]

[24] R. K. Kodali, S. C. Rajanarayanan, and L. Boppana, "Iot based vehicular air quality monitoring system," in *2019 IEEE R10 Humanitarian Technology Conference (R10-HTC)(47129)*, pp. 258–262, IEEE, 2019. [Cited on page 13]

[25] S. Jaloudi, "Mqtt for iot-based applications in smart cities," *Palestinian Journal of Technology and Applied Sciences*, no. 2, 2019. [Cited on pages 13 and 15]

[26] steves-internet guide, "Understanding the mqtt protocol packet structure." Available at `http://www.steves-internet-guide.com/mqtt-protocol-messages-overview/`. (Last accessed in 11/15/2023). [Cited on page 13]

[27] digicert, "What's the difference between client certificates vs. server certificates?." Available at `https://www.digicert.com/faq/public-trust-and-certificates/whats-the-difference-between-client-certificates-vs-server-certificates`. (Last accessed in 02/16/2024). [Cited on page 14]

[28] "Mosquitto test broker." Available at `https://test.mosquitto.org/`. (Last accessed in 02/16/2024). [Cited on pages 14, 37, and 40]

[29] "The difference between ssl and tls." Available at `https://aws.amazon.com/pt/compare/the-difference-between-ssl-and-tls/`. (Last accessed in 11/15/2023). [Cited on page 15]

[30] H. Team, "Hivemq - tls and mqtt: How is the performance affected?." Available at `https://www.hivemq.com/article/how-does-tls-affect-mqtt-performance/`. (Last accessed in 11/15/2023). [Cited on pages ix and 15]

[31] M. A. Pradana, A. Rakhmatsyah, and A. A. Wardana, "Flatbuffers implementation on mqtt publish/subscribe communication as data delivery format," in *2019 6th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, pp. 142–146, IEEE, 2019. [Cited on page 15]

[32] IEFT, "Hypertext transfer protocol – http/1.1." Available at `https://datatracker.ietf.org/doc/html/rfc2616`. (Last accessed in 04/02/2024). [Cited on page 15]

[33] B. Wukkadada, K. Wankhede, R. Nambiar, and A. Nair, "Comparison with http and mqtt in internet of things (iot)," in *2018 International Conference on Inventive Research in Computing Applications (ICIRCA)*, pp. 249–253, IEEE, 2018. [Cited on pages ix and 16]

[34] W. Bziuk, C. V. Phung, J. Dizdarević, and A. Jukan, "On http performance in iot applications: An analysis of latency and throughput," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 0350–0355, IEEE, 2018. [Cited on page 16]

[35] N.-M. Drogeanu, L.-A. Perişoară, and J.-A. Văduva, "Web interface for iot vehicle monitoring system," in *2022 IEEE 28th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, pp. 185–190, IEEE, 2022. [Cited on page 16]

[36] I. E. T. Force, "The constrained application protocol (coap)." Available at `https://www.rfc-editor.org/rfc/rfc7252.html`. (Last accessed in 11/21/2023). [Cited on page 16]

[37] I. Craggs, "Mqtt vs coap for iot." Available at `https://www.hivemq.com/article/mqtt-vs-coap-for-iot/`. (Last accessed in 11/21/2023). [Cited on pages ix and 17]

[38] I. Craggs, "Mqtt vs coap for iot." Available at `https://www.hivemq.com/article/mqtt-vs-coap-for-iot/?utm_source=YouTube&utm_medium=YouTube+video+mqtt+vs+coap&utm_campaign=YouTube+video+mqtt+vs+coap`. (Last accessed in 01/07/2024). [Cited on page 17]

[39] H. A. Khattak, M. Ruta, and E. E. Di Sciascio, "Coap-based healthcare sensor networks: A survey," in *Proceedings of 2014 11th International Bhurban Conference on Applied Sciences & Technology (IBCAST) Islamabad, Pakistan, 14th-18th January, 2014*, pp. 499–503, IEEE, 2014. [Cited on page 17]

[40] "Eclipse californium (cf) coap framework." Available at `https://projects.eclipse.org/projects/iot.californium`. (Last accessed in 02/16/2024). [Cited on page 17]

[41] A. P. Castellani, M. Gheda, N. Bui, M. Rossi, and M. Zorzi, "Web services for the internet of things through coap and exi," in *2011 IEEE International Conference on Communications Workshops (ICC)*, pp. 1–6, IEEE, 2011. [Cited on pages ix, 17, and 18]

[42] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan, "Performance evaluation of mqtt and coap via a common middleware," in *2014 IEEE ninth international conference on intelligent sensors, sensor networks and information processing (ISSNIP)*, pp. 1–6, IEEE, 2014. [Cited on pages ix, 19, 20, and 21]

[43] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE international systems engineering symposium (ISSE)*, pp. 1–7, IEEE, 2017. [Cited on page 19]

[44] S. Nicholas, "Power profiling: Https long polling vs. mqtt with ssl, on android." Available at `http://stephendnicholas.com/posts/power-profiling-mqtt-vs-https`. (Last accessed in 01/06/2024). [Cited on page 20]

[45] W. Colitti, K. Steenhaut, and N. De Caro, "Integrating wireless sensor networks with the web," *Extending the Internet to Low power and Lossy Networks (IP+ SN 2011)*, 2011. [Cited on page 20]

[46] E. Bertrand-Martinez, P. Dias Feio, V. d. Brito Nascimento, F. Kon, and A. Abelém, "Classification and evaluation of iot brokers: A methodology," *International Journal of Network Management*, vol. 31, no. 3, p. e2115, 2021. [Cited on pages ix, 21, 22, and 23]

[47] "Rabbitmq." Available at `https://www.rabbitmq.com/`. (Last accessed in 03/19/2024). [Cited on page 22]

[48] GMI, "Real-time operating system market." Available at `https://www.gminsights.com/industry-analysis/real-time-operating-system-market`. (Last accessed in 02/03/2024). [Cited on page 23]

[49] R. Elberger, "Three key factors in choosing a real-time operating system (rtos)." Available at `https://www.embedded.com/three-key-factors-in-choosing-a-real-time-operating-system-rtos/`. (Last accessed in 02/03/2024). [Cited on page 24]

[50] N. Dahad, "Embedded survey 2023: more ip reuse as workloads surge." Available at `https://www.embedded.com/embedded-survey-2023-more-ip-reuse-as-workloads-surge/`. (Last accessed in 02/03/2024). [Cited on page 24]

[51] BYTESNAP, "Freertos vs linux for embedded systems." Available at https://www.bytesnap.com/news-blog/freertos-vs-linux-embedded-systems/. (Last accessed in 02/05/2024). [Cited on page 24]

[52] IETF, "Terminology for constrained-node networks." Available at https://www.ietf.org/rfc/rfc7228.txt. (Last accessed in 02/03/2024). [Cited on page 24]

[53] T. Loveless, "What are the problems with embedded linux?." Available at https://www.lynx.com/embedded-systems-learning-center/what-are-the-problems-with-embedded-linux. (Last accessed in 02/05/2024). [Cited on page 24]

[54] "Difference between hard real time and soft real time system." Available at https://www.geeksforgeeks.org/difference-between-hard-real-time-and-soft-real-time-system/. (Last accessed in 02/05/2024). [Cited on page 24]

[55] IntervalZero, "Rtos: The key to deterministic behavior." Available at https://www.intervalzero.com/rtos-key-to-deterministic-behavior/. (Last accessed in 02/05/2024). [Cited on page 25]

[56] V. Bridgers, "Real time linux scheduling comparison." Available at https://elinux.org/images/d/de/Real_Time_Linux_Scheduling_Performance_Comparison.pdf. (Last accessed in 02/06/2024). [Cited on page 25]

[57] "Freertos kernel ports." Available at https://www.freertos.org/RTOS_ports.html. (Last accessed in 02/07/2024). [Cited on page 25]

[58] "Freertos supported architectures." Available at https://freertos.org/partners/architectures.html. (Last accessed in 02/07/2024). [Cited on page 25]

[59] "Riot os documentation." Available at https://doc.riot-os.org/group__core__sched.html. (Last accessed in 02/08/2024). [Cited on page 25]

[60] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating systems for low-end devices in the internet of things: a survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, 2015. [Cited on pages xi, 25, and 26]

[61] "What is µc/os-iii?." Available at https://www.phaedsys.com/principals/micrium/mucos3.html#top. (Last accessed in 02/07/2024). [Cited on pages xi and 26]

[62] "µc/os-iii users manual." Available at `https://www.analog.com/media/en/dsp-documentation/software-manuals/Micrium-uCOS-III-UsersManual.pdf`. (Last accessed in 02/08/2024). [Cited on pages xi and 26]

[63] R. Raymundo Belleza and E. de Freitas Pignaton, "Performance study of real-time operating systems for internet of things devices," *IET Software*, vol. 12, no. 3, pp. 176–182, 2018. [Cited on pages xi and 26]

[64] S. Challouf, L. Kriaa, and L. A. Saidane, "Power consumption comparison of synchronized iot devices running freertos and riot," in *2019 8th International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN)*, pp. 1–5, IEEE, 2019. [Cited on pages ix and 27]

[65] Segger, "Tickless mode." Available at `https://wiki.segger.com/Tickless_Mode`. (Last accessed in 02/08/2024). [Cited on page 27]

[66] R. Ratasuk, N. Mangalvedhe, and A. Ghosh, "Overview of lte enhancements for cellular iot," in *2015 IEEE 26th annual international symposium on personal, indoor, and mobile radio communications (PIMRC)*, pp. 2293–2297, IEEE, 2015. [Cited on page 28]

[67] "Mesh topology." Available at `https://www.computerhope.com/jargon/m/mesh.htm`. (Last accessed in 02/09/2024). [Cited on page 28]

[68] K. Mekki, E. Bajic, F. Chaxel, and F. Meyer, "Overview of cellular lpwan technologies for iot deployment: Sigfox, lorawan, and nb-iot," in *2018 ieee international conference on pervasive computing and communications workshops (percom workshops)*, pp. 197–202, IEEE, 2018. [Cited on page 28]

[69] P. I. Reports, "Low power wide area network (lpwan) market detailed analysis." Available at `https://www.linkedin.com/pulse/low-power-wide-area-network-lpwan-market-7tqyc/?trk=article-ssr-frontend-pulse_more-articles_related-content-card`. (Last accessed in 02/09/2024). [Cited on page 28]

[70] E. Pasqua, "Lpwan technologies: How cellular mnos are placing their bets." Available at `https://iot-analytics.com/lpwan-technologies-cellular-mnos/`. (Last accessed in 02/09/2024). [Cited on page 28]

[71] K. A. Aldahdouh, K. A. Darabkh, W. Al-Sit, *et al.*, "A survey of 5g emerging wireless technologies featuring lorawan, sigfox, nb-iot and lte-m," in *2019 International conference on wireless communications signal processing and networking (WiSPNET)*, pp. 561–566, IEEE, 2019. [Cited on pages xi and 29]

[72] L. Alliance, "A tecnical overview of lora and lorawan." Available at https://cdn.everythingrf.com/live/LoRaWAN101-technical-overview.pdf. (Last accessed in 02/12/2024). [Cited on pages xi and 29]

[73] T. Niwa, "Cellular iot module market update." Available at https://iotbusinessnews.com/2023/01/25/06155-cellular-iot-module-market-update/. (Last accessed in 02/13/2024). [Cited on page 30]

[74] "Lpwa bg96 cat m1/nb1/egprs." Available at https://www.quectel.com/product/lpwa-bg96-cat-m1-nb1-egprs. (Last accessed in 02/13/2024). [Cited on page 30]

[75] "Quectel bc95-g." Available at https://www.quectel.com/wp-content/uploads/pdfupload/Quectel_BC95-G_NB-IoT_Specification_V1.9.pdf. (Last accessed in 02/13/2024). [Cited on page 30]

[76] u blox, "World's first nb-iot module certified for use in hazardous environments enters initial production." Available at https://www.designworldonline.com/worlds-first-nb-iot-module-certified-for-use-in-hazardous-environments-enters-init (Last accessed in 02/13/2024). [Cited on page 30]

[77] "Lte eg91 series." Available at https://www.quectel.com/product/lte-eg91-series. (Last accessed in 02/13/2024). [Cited on page 30]

[78] "Gsm/gprs m66." Available at https://www.quectel.com/product/gsm-gprs-m66. (Last accessed in 02/13/2024). [Cited on page 30]

[79] F. Di Nuzzo, D. Brunelli, T. Polonelli, and L. Benini, "Structural health monitoring system with narrowband iot and mems sensors," *IEEE Sensors Journal*, vol. 21, no. 14, pp. 16371–16380, 2021. [Cited on pages ix, 30, and 31]

[80] V. Campiotti, N. Finozzi, J. Irazoqui, V. Cabrera, R. Ungerfeld, and J. Oreggioni, "Wearable device to monitor sheep behavior," *IEEE Embedded Systems Letters*, 2022. [Cited on page 31]

[81] "At commands." Available at https://docs.monogoto.io/tips-and-tutorials/at-commands. (Last accessed in 02/14/2024). [Cited on page 32]

[82] Q. Zhicong, L. Delin, and W. Shunxiang, "Analysis and design of a mobile forensic software system based on at commands," in *2008 IEEE International Symposium on Knowledge Acquisition and Modeling Workshop*, pp. 597–600, IEEE, 2008. [Cited on page 32]

[83] "eclipse-mosquitto." Available at `https://hub.docker.com/_/eclipse-mosquitto`. (Last accessed in 04/02/2024). [Cited on pages 34 and 43]

[84] "Pycharm." Available at `https://www.jetbrains.com/pycharm/`. (Last accessed in 03/17/2024). [Cited on page 37]

[85] "Cutecom." Available at `https://help.ubuntu.com/community/Cutecom`. (Last accessed in 03/17/2024). [Cited on page 37]

[86] "mymqtt." Available at `https://mymqtt.app/`. (Last accessed in 03/17/2024). [Cited on page 37]

[87] "pyserial 3.5." Available at `https://pypi.org/project/pyserial/`. (Last accessed in 05/21/2024). [Cited on page 38]

[88] "Bg96 mqtt application note." Available at `https://www.quectel.com/wp-content/uploads/2021/03/Quectel_BG96_MQTT_Application_Note_V1.2.pdf`. (Last accessed in 03/12/2024). [Cited on pages 39 and 40]

[89] "Bg96 file manual." Available at `https://www.quectel.com/wp-content/uploads/2021/03/Quectel_BG96_FILE_AT_Commands_Manual_V1.1.pdf`. (Last accessed in 03/20/2024). [Cited on page 39]

[90] "Bg96 ssl manual." Available at `https://www.quectel.com/wp-content/uploads/2021/03/Quectel_BG96_SSL_Application_Note_V1.1.pdf`. (Last accessed in 03/20/2024). [Cited on page 39]

[91] "Docker main page." Available at `https://www.docker.com/`. (Last accessed in 04/02/2024). [Cited on page 43]

[92] "Eclipse mosquitto™." Available at `https://mosquitto.org/`. (Last accessed in 04/03/2024). [Cited on page 43]

[93] "Dockerfile github." Available at `https://github.com/eclipse/mosquitto/blob/master/docker/1.5/Dockerfile`. (Last accessed in 04/02/2024). [Cited on page 45]

[94] "Mqtt explorer." Available at `http://mqtt-explorer.com/`. (Last accessed in 04/02/2024). [Cited on page 45]

[95] "core-mqtt." Available at `https://www.freertos.org/mqtt/index.html`. (Last accessed in 04/12/2024). [Cited on page 47]

[96] "Backoff algorithm." Available at `https://www.freertos.org/backoff-algorithm.html`. (Last accessed in 04/23/2024). [Cited on page 51]

[97] "Event bits (or flags) and event groups." Available at `https://www.freertos.org/FreeRTOS-Event-Groups.html`. (Last accessed in 04/28/2024). [Cited on page 55]

[98] "What is a ci/cd pipeline?." Available at `https://www.redhat.com/en/topics/devops/what-cicd-pipeline`. (Last accessed in 05/8/2024). [Cited on page 58]

[99] "paho-mqtt 2.1.0." Available at `https://pypi.org/project/paho-mqtt/`. (Last accessed in 05/24/2024). [Cited on page 62]