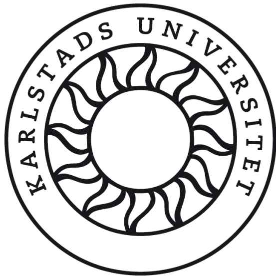


Remote robot control from Docker

Irene Campo Prieto

Faculty of Health, Science and Technology

Computer Science

30HP (ECTS)

Supervisor: Andreas Kassler

Examiner: Sebastian Herold

Date: 281020

Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Computer Science

Irene Campo Prieto

Remote robot control from Docker

Abstract

Currently, we witness a new phase of digitization which is fueled by the development of Internet-connected smart sensors (Internet of Things - IoT). Also, about the processing of large data volumes that they create using Big Data analytics, leveraging the compute resources from the Cloud and Edge-based systems. For data exchange in the IoT world, typically lightweight communication protocols such as Message Queuing Telemetry Transport (MQTT) are used which are based on publisher/subscriber communication pattern where a broker mediates data among interested parties.

In order to provide reliable communication, MQTT provides different Quality of Service (QoS) mechanisms. MQTT publishers, subscribers and brokers can be deployed inside containers on virtualized infrastructure to facilitate large-scale virtualized compute frameworks from the cloud for scalable data analytics. However, each docker containers requires a specific amount of resources to provide the required response time.

In this thesis, we evaluate the impact of resource sharing due to the virtualized deployment of MQTT components on latency and response time of IoT applications. We deploy a testbed of Arduino and Raspberry Pi devices that host MQTT clients to pull sensor data towards MQTT clients inside the cloud. We also evaluate the impact of different QoS levels at the MQTT protocol on latency. Our results indicate that proper resource allocation and QoS parametrization is important for maintaining low and stable latency.

Resumen

Actualmente, somos testigos de una nueva fase de digitalización que es alimentada por el desarrollo de sensores inteligentes conectados a Internet (Internet de las Cosas - IoT). También, el procesamiento de grandes volúmenes de datos que crean usando análisis de Big Data, aprovechando los recursos de computación de los sistemas basados en la Cloud y Edge-based. Para el intercambio de datos en el mundo del IoT, se utilizan protocolos de comunicación típicamente ligeros como el Message Queuing Telemetry Transport (MQTT) que se basan en el patrón de comunicación editor/suscriptor en el que un intermediario media los datos entre las partes interesadas.

Con el fin de proporcionar una comunicación fiable, MQTT proporciona diferentes mecanismos de Quality of Service (QoS). Los editores, suscriptores y brokers de MQTT pueden desplegarse dentro de contenedores en una infraestructura virtualizada para facilitar marcos de computación virtualizados a gran escala desde la nube para el análisis de datos escalables. Sin embargo, cada contenedor Docker requiere una cantidad específica de recursos para proporcionar el tiempo de respuesta necesario.

En esta tesis, evaluamos el impacto de compartir recursos debido al despliegue virtualizado de los componentes MQTT en la latencia y el tiempo de respuesta de las aplicaciones de IoT. Implementamos un banco de pruebas con los dispositivos Arduino y Raspberry Pi, que albergan clientes MQTT para atraer los datos de los sensores hacia los clientes MQTT dentro de la nube. También evaluamos el impacto de los diferentes niveles de QoS en el protocolo MQTT sobre la latencia. Nuestros resultados indican que una adecuada asignación de recursos y parametrización de la QoS es importante para mantener una latencia baja y estable.

Acknowledgements

I would like to express my gratitude to my advisor Enrica Zola who has given me the opportunity to do this thesis.

I would also like to thank my co-advisor Andreas Kassler for his assistance during the project. A special thanks goes also to the University of Karlstad, for all the support, even though the special situation of the Covid-19.

And last but not least, I would like to thank my mother for all the confidence and support that she has given me at all moments, not only during this project but throughout my studies.

Contents

1 Introduction **1**

1.1 Objectives and goals . . . 3

1.2 Evaluation . . . 3

1.3 Ethics and sustainability . . . 3

1.4 Contribution . . . 4

1.5 Structure . . . 4

2 Background and Related Work **5**

2.1 Docker . . . 5

2.2 IoT . . . 6

2.3 MQTT . . . 7

2.3.1 What is it? . . . 7

2.3.2 How does it works? . . . 7

2.3.3 Quality of Service . . . 8

2.4 Related works . . . 10

3 System Design **12**

3.1 Communication Protocols . . . 12

3.2 Hardware . . . 12

3.2.1 mBot Ranger - Arduino . . . 12

3.2.2 Raspberry Pi 4 14

3.2.3 Communication Raspberry Pi + Arduino 15

3.3 Software 15

3.3.1 MQTT Broker 16

3.3.2 MQTT Clients 16

3.4 Summary 17

4 Implementation **18**

4.1 MQTT Broker container installation 18

4.1.1 MQTT Broker container with prioritisation 18

4.2 MQTT Client container installation 19

4.2.1 MQTT Client container with prioritisation 20

4.3 MQTT Client installation in RPi 20

4.4 Summary 22

5 Evaluation **23**

5.1 Evaluation Scenario 0 – No Arduino 25

5.2 Evaluation Scenario 1 – Arduino 57600bps 26

5.3 Evaluation Scenario 2 – Docker Stress 28

5.4 Evaluation Scenario 3 – Docker Stress without Arduino 30

5.5 Evaluation Scenario 4 – MQTT Broker Prioritisation 32

5.6 Evaluation Scenario 5 – MQTT Broker and MQTT Docker Client Prioritisation . . . 35
5.7 Evaluation Scenario 6 – MQTT Broker Prioritisation without Arduino . . . 37
5.8 Evaluation Scenario 7 - MQTT Stresser . . . 39
5.9 Evaluation Scenario 8 - MQTT Stresser with MQTT Docker Client Prioritsation . . . 40
5.10 Summary . . . 43

6 Results Analysis **44**

6.1 Arduino 57600 vs No Arduino . . . 44
6.2 Arduino 57600 + Docker Stress vs No Arduino + Docker Stress . . . 45
6.3 Arduino 57600 vs Arduino 57600 + Docker Stress . . . 47
6.4 Arduino 57600 + Docker Stress vs MQTT Broker Prioritisation vs MQTT Broker and MQTT Docker Client Prioritisation . . . 48
6.5 No Arduino + Docker Stress vs No Arduino + MQTT Broker Prioritisation 49
6.6 Summary . . . 51

7 Conclusion and Future Work **52**

References **53**

A Appendix - Scripts **56**

List of Figures

2.1 Scheme of the functioning of an MQTT scenario. 8

3.1 Image of mBot Ranger 13

3.2 Image of RPi 4 14

3.3 Code for Serial communication at RPi 4 15

3.4 Code for Serial communication at Arduino 15

3.5 System design overview 17

4.1 Command to download the last image of Docker 18

4.2 Execution to create the Broker's container 18

4.3 Execution to create the Broker's container with prioritisation 19

4.4 Command to download the last Docker image and execution to create the Client's container 19

4.5 Execution to install python package 19

4.6 Execution to install *paho.mqtt.python* 20

4.7 Execution to run the Client's container with prioritisation 20

4.8 Execution to install MQTT Client in RPi 21

4.9 Execution to install *paho.mqtt.python* 21

4.10 Configuration of the *file mosquito.conf* 21

5.1 Diagram of the full setup with the evaluation metrics 23

5.2 Diagram of the scenario without Arduino 25

5.3 Diagram of scenario 1 27

5.4 Diagram of scenario 2 29

5.5 Diagram of scenario 3 31

5.6 Diagram of scenario 4 33

5.7 Diagram of scenario 5 35

5.8 Diagram of scenario 6 38

5.9 Diagram of scenario 7 41

5.10 Diagram of scenario 8 42

6.1 CDF graph comparing latencies of Scenarios 0 and 1 44

6.2 CDF graph comparing comparing latencies of Scenarios 2 and 3 46

6.3 CDF graph comparing latencies of Scenarios 1 and 2 47

6.4 CDF graph comparing latencies of Scenarios 2, 4 and 5 49

6.5 CDF graph comparing latencies of Scenarios 3 and 6 50

A.1 Docker MQTT Client Publishing Script 56

A.2 Docker MQTT Client Subscribing Script with QoS 57

A.3 Docker MQTT Client Subscribing Script 58

A.4 RPi MQTT Client Subscribing-Publishing Script 59

List of Tables

5.1 Description of the evaluation metrics and timestamps 24

5.2 Table of the latencies gathered in Scenario 0 26

5.3 Table of the latencies gathered in Scenario 1 27

5.4 Table of the latencies gathered in Scenario 2 29

5.5 Table of the latencies gathered in Scenario 3 31

5.6 Table of the latencies gathered in Scenario 4 34

5.7 Table of the latencies gathered in Scenario 5 36

5.8 Table of the latencies gathered in Scenario 6 38

5.9 Table of the latencies gathered in Scenario 8 42

1 Introduction

The term "Internet of things" concerns the connection of devices to each other and to humans throughout the Internet. This interaction makes it possible to create more intelligent objects and connect them to the intelligent network, which makes it possible to detect and control the physical world remotely. This remote control has made the Internet of Things (IoT) so important, as it permits different sensors/devices to be connected through the network, sending millions of data in real-time to interpretation and analysis centres for analysis and decision making.

Above all, IoT is now becoming very popular for the control and automation of household appliances and devices at home. However, it is also starting to be used in other cases such as communication between autonomous vehicles, monitoring of machinery in factories (Industry 4.0) and drones with search and rescue operations.

In order for the devices in an IoT system to be able to communicate through the Internet, different communication protocols are used, depending on the application and the environment in which it is located. Message Queuing Telemetry Transport (MQTT) is a lightweight IoT communication protocol that can be implemented in devices connected to networks with limited bandwidth and also provides flexibility that can support several application scenarios for IoT devices and services. Besides, it uses a communication pattern based on publisher/subscriber, which requires a central server where MQTT Clients connect. This server is the MQTT Broker and is in charge of managing the communications between the clients. MQTT has become the standard for IoT communications.

Quality of Service (QoS) is a mechanism used to ensure traffic prioritisation and guarantee minimum bandwidth. In MQTT, the QoS level is an agreement between the sender and the receiver (MQTT Client and MQTT Broker) that defines what guarantee of delivery

there is for a specific message. Two parts must be considered to understand how this service works. In one hand, the message delivered from Publishing Client to Broker, and on the other hand, the message delivered from the Broker to the Subscribing Client.

Many IoT devices have limited computing and communication resources. In order to be able to control it, a resource management approach is required. Virtualisation is a technology that enables the creation of useful IT services through resources that are traditionally tied to hardware. It is one of the approaches that lead to better optimisation and efficient resource management. In addition, the excessive amount of data generated by IoT devices also causes them to be more resource-constrained. Therefore, virtualisation plays a crucial role in IoT applications with resource limitations. Docker is an Open Source tool that allows lightweight virtualisation. It is designed to make the creation, implementation and execution of applications easier through lightweight and portable containers.

A container is a standard unit of software that packages the code and all its dependencies so that the application can run quickly and reliably from one environment to another. By default, a Docker container has no resource restrictions but allows different resource control policies to be applied to the containers. Through the establishment of flags, it can control how much memory or CPU the container is entitled to use. Moreover, some flags allow CPU scheduler of the Operating System, enabling real-time use case for Docker containers.

Going deeper into the performance of an IoT system, the devices communicate over the Internet with the cloud, just like any other device does. The cloud acts as an application service provider to exchange data and control message traffic. Once the data reaches the cloud, the software processes it and then can decide to implement an action, such as sending an alert or automatically setting the sensors.

1.1 Objectives and goals

The main idea in this thesis is to evaluate how network performance and resource sharing due to virtualisation impact the control cycle in IoT applications when control actions are outsourced to Docker containers. So, it has been designed and implemented an IoT system that allows remote control through Docker, using MQTT as a communication protocol. The MQTT Broker, hosted in Docker, will perform the function of the cloud and will be in charge of controlling the message traffic and processing them.

1.2 Evaluation

In order to carry out an evaluation of the system, an MQTT Client connected to an Arduino and another one hosted in Docker have been created, and, also, a loop control has been implemented. From here, a testbed has been created, and the latency time of the loop control has been evaluated. This latency time will be referred to the time it takes for a message to be published by an MQTT Client hosted in Docker, reaches the sensor (at the Arduino) by passing through Broker and until the response message makes the return path. The interval time between messages within the loop has been varied in order to make the evaluation more complete. All these comparisons will be shown through CDF (Cumulative distribution function) graphs.

Besides, the scenarios with different communication protocol QoS values have been compared in order to complete the evaluation.

1.3 Ethics and sustainability

As mentioned above, Docker allows the creation, implementation and execution of applications through portable containers. This facility, if considered from the point of view of

sustainability, is outstanding as it allows applications to be implemented consistently with little concern for the configuration of the underlying hardware or software. In other words, it gives the opportunity to "create" a new robot by updating the container without the need to manufacture a new robot. This reuse of hardware has privacy issues, as the new container can have access to confidential host information.

1.4 Contribution

The main contribution of this project is the demonstration that there is a significant impact on latency when other containers are deployed on the same hosts that exert stress on the MQTT Broker's container. A further contribution has been the creation of a loop control application, with which it has been possible to configure a test bench and analyse the latency time. Finally, there is also a contribution to the design of different scenarios of the IoT system that have allowed, through the comparison of their latency times, to make a complete system evaluation.

1.5 Structure

This document is structured as follows. Section 2 presents a description of all the main concepts used in the system and some related work. In Section 3, all the elements of the system are described and how they are connected. Moreover, how these elements have been configured as described in Section 4.

The design of all analysis scenarios is explained in Section 5, and in Section 6, are the analyses with their CDF plots.

Finally, in Section 7, are the overall conclusions and possible extensions of the work.

2 Background and Related Work

2.1 Docker

Docker is an open-source tool designed to make the creation, implementation and execution of applications easier through lightweight and portable containers [2]. It allows packing an application, with all its necessary parts (such as libraries, system tools, execution time and other dependencies) in standardized units called containers. Besides, Docker allows to deploy and scale the application to any environment, knowing that the code is going to run.

In a certain way, Docker is a bit like a virtual machine, but with the difference that instead of creating a whole virtual operating system, Docker allows applications to use the same kernel as the system they are running on. This results in a significant increase in performance and a reduction in the size of the application.

The essential elements of Docker are the images and the containers [1]:

- **Images**

  Docker images are the static representation of the application or service with its configuration and dependencies. In other words, it is a kind of template that captures the state of a container. The images are used to create new containers, and they never change.

  In Dockerhub, there are many public images with primary elements that can be

downloaded and used. If the image wanted is not found here, it can be created through the DockerFile configuration file. In this file, it is specified what needs to be contained in the image and the installation tool commands.

- **Containers**

  Containers are instances that execute an image, meaning the ones in charge of running the commands and the application. As opposed to images, containers can change. Docker tracks these changes as a version control tool.

2.2 IoT

The Internet of Things, or better know as IoT, has the basic premise and goal to "connect the unconnected"[18]. This means being able to communicate objects that are not connected to the Internet and be able to interact them with people and other objects. This interaction makes it possible to create smarter objects and connect them to the intellectual network [15], allowing us to detect and control the physical world remotely. This point enables closer integration between the physical world and equipment by improving in the areas of efficiency, accuracy, automation, and enablement of advanced applications.

The IoT world is vast and encompasses a lot of protocols and components, which is why it must be considered as the combination of various concepts, protocols and technologies.

In 1999, pioneering British technologist Kevin Ashton [14] introduced the term Internet of Things to describe the ability to connect sensors to the Internet for new features.

In an IoT ecosystem, smart devices share the different data that sensors collect and can be analysed in the cloud or on-premises. Sometimes these devices communicate with other related devices and act on the information they get from [16] each other. As can be

understood, these devices are the ones that do most of the work, without the need for human intervention.

2.3 MQTT

2.3.1 What is it?

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol that uses a publish/subscribe communication pattern [19]. Typical it is used for machine-to-machine (M2M) communication and in the types of Internet of Things connections. This protocol is a good option when the connections of the devices are with unreliable networks or networks with limited bandwidth resources.

2.3.2 How does it works?

As explained above, MQTT has a push messaging service operation with a publisher/subscriber pattern. This type of infrastructure is form by a central server where the clients connect. The server is called Broker and is in charge of receiving communications from the clients and sending these communications to the other clients. As we can see, clients can only communicate with each other through the Broker [3]. The Client - Broker connection, which is TCP/IP, is made through port 1883 (or port 8883 when it is TLS) and remains open until the client closes it [12].

As shown in Figure 2.1, MQTT Clients can be a publisher, a subscriber or both. The publisher is the Client that wants to send data to the Broker. On the other hand, the subscriber is the Client that wants to receive the data from another Client.

Another element reflected in the scheme, and that is important in an MQTT scenario, are

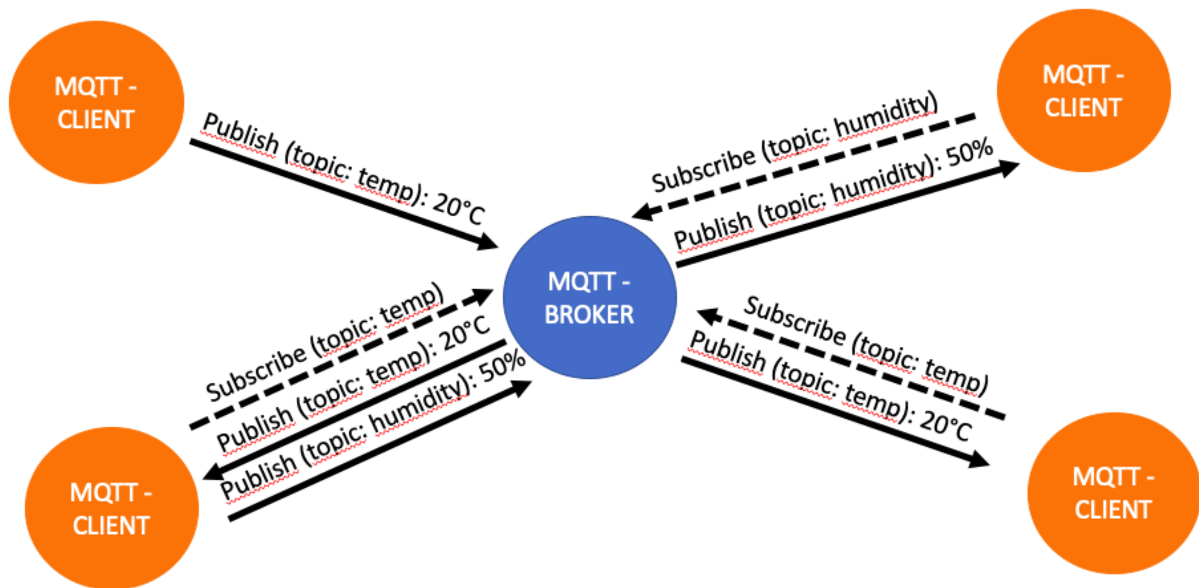


Figure 2.1: Scheme of the functioning of an MQTT scenario.

the Topics. A Topic [10] is a kind of filter that MQTT - Broker uses with the received messages in order to discriminate to which subscribing customers it should be delivered. It is essential to have clear the following points: Broker accepts all Topics, Clients can subscribe to one or more Topics (by establishing several subscriptions)and Clients publish messages indicating a single Topic.

Therefore, the Broker receives the message and, if he has a subscription that matches the Topic filter, the message is transmitted to the subscribed customers.

2.3.3 Quality of Service

Quality of Service (QoS) is a mechanism used to ensure traffic prioritisation and guarantee minimum bandwidth. QoS measures bandwidth and prioritises packets based on priority queues [20].

In MQTT, the QoS level is an agreement between the sender and the receiver (MQTT Client and MQTT Broker) that defines what guarantee of delivery there is for a specific message. Two delivery parts of the package must be considered to understand how this service works.In one hand, the message delivered from Publishing Client to Broker, and on the other hand, the message delivered from the Broker to the Subscribing Client.

When the Client who publishes the Broker sends him a message, he defines the level of QoS. The Broker transmits this message to the subscribed Clients, but with the QoS level they have subscribed to [9]. That is to say, if the Client subscribes with a lower QoS level than that of the publication, the Broker will transmit the message with the same level.

There are three levels of QoS defined (0, 1 and 2). Each one of them and its function, are detailed below:

- **QoS Level 0 - at most once**

  This is the minimum level of service, the simplest and the least overloaded by sending a message. There is no guarantee of delivery. The Customer posts the message, and the Broker does not reply with an acknowledgement of receipt[11].

- **QoS Level 1 - at least once**

  This level ensures that the message will be transferred correctly to the Broker. The Client keeps the message until it receives an acknowledgement from the Broker (PUB-ACK). If, after a reasonable time, he does not receive the PUBACK, he publishes the message again. Repeat this so many times until it receives confirmation of receipt. The Broker does not send the PUBACK until it has transferred it to all the

subscribers. Therefore, at this level, the sending is guaranteed, although the Broker may receive the message more than once[11].

- **QoS Level 2 - exactly once**

  Is the highest level of service, the safest and the slowest. It guarantees that the recipient receives the message only once through a package identifier from the original Publish message. At this level, a sequence of 4 messages is produced.

  First, the Client makes a Publish and waits for the Broker to reply with an ack (PUBREC). If after a reasonable time it does not respond, resends the message with the duplicate Packet Flag. Second, once the Client receives PUBREC it sends a message saying that the Publish message can be removed from the queue (PUBREL). Third, the Doker receives PUBREL, which means that they can now send it to all subscribers. Fourth and last, the Publish Client receives a message from Broker (PUBCOMP) with confirmation of the entire process[11].

2.4 Related works

In recent years, there have been many research and development efforts to incorporate the Internet of Things into everyday life. Since the focus of this work is the evaluation of remote control of a robot through Docker, two related research works are detailed in this section.

***"Benchmarking and Profiling 5G Verticals' Applications: An Industrial IoT Use Case"*[13]**

This article was introduced at the IEEE Conference on Network Softwarization, NetSoft

2020.

The document details an integrated solution for 5G-oriented benchmarking and vertical applications at profiling, accompanying with a use case study in a smart manufacturing industry. It combines the design and execution of evaluation experiments, as well as the analysis of the extracted data.

As future work, based on the results of efficiency analyses, they plan to work on the development and evaluation of models that can help the management of VNF escalation actions.

***A scalable and low-cost MQTT broker clustering system*[17]**

This article was introduced at the 2nd International Conference on Information Technology (INCIT), in 2017

In this document, they intend to build a scalable MQTT Broker by combining the Raspberry Pi and an open-source broker software. They propose a scalable and low cost, but high-performance MQTT Broker scenario.

3 System Design

3.1 Communication Protocols

As the system communication protocol, it has been configured the MQTT protocol (defined in the section 2.3).

The central highlight of this protocol is that the publisher and subscriber do not need to know each other, as the protocol has a star topology. For this reason, an MQTT Broker has been created and hosted in Docker. This peculiarity will allow to easily add or replace any component of the system without the need to modify the other parts.

3.2 Hardware

The hardware involved in the system consists of an Raspberry Pi 4 connected in serial with the Arduino inside the mBot Ranger robot. All the elements are detailed in the following sections.

3.2.1 mBot Ranger - Arduino

MBot Ranger is a kit that permits the construction of a robot. It consists of:

- Arduino Mega 2560 as a motherboard with 10 expansion ports.
- Motors with encoder for precise motion control
- Wireless communication via bluetooth/2.4G
- Module with 12 RGB LEDs

- 6 sensors:

  Sensor line following

  Ultrasonic obstacle sensor

  Light sensor

  Sound sensor

  Temperature sensor

  Gyroscopic

- Powered by USB cable or 6 AA batteries

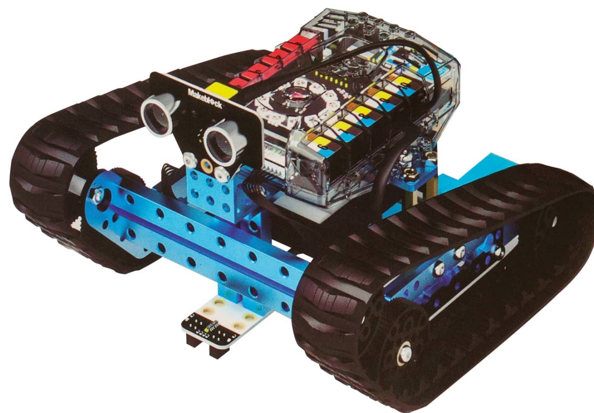


Figure 3.1: Image of mBot Ranger

This robot is in charge of consulting the data to the sensors. Although six different types of sensors have been detailed, only the temperature sensor has been used in this project since one sensor is enough to calculate the latency of the whole system.

A Raspberry Pi 4 has been connected in series to allow the mBot Ranger to communicate through the MQTT Protocol.

3.2.2 Raspberry Pi 4

As commented in the previous section, the Raspberry Pi 4 was used to allow the mBot Ranger to communicate through the MQTT protocol with the MQTT Broker. In that way, Raspberry Pi 4 receives the messages from the Broker by MQTT and transmits it to the Arduino through the serial connection.

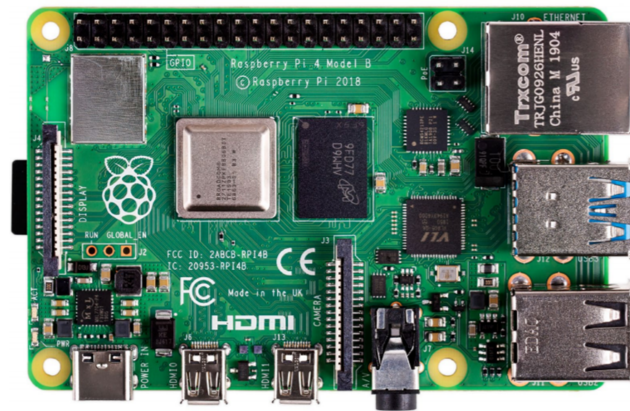


Figure 3.2: Image of RPi 4

Raspberry Pi 4 (Figure 3.2) is a Single Board Computer with 64-bit ARM architecture with the following characteristics [7]:

Processor:	Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
Memory:	4GB LPDDR4
Connectivity:	2.4 GHz and 5.0 GHz IEEE 802.11b/g/n/ac wireless LAN, Bluetooth 5.0, BLE Gigabit Ethernet 2 USB 3.0 ports 2 USB 2.0 ports

A 16GB Micro-SD card has been used to install the operating system and as primary storage.

3.2.3 Communication Raspberry Pi + Arduino

As explained above, MBot and Raspberry Pi 4 are connected via a serial connection. For this purpose, a USB cable has been connected between both.

For the Raspberry Pi, has been used a USB connector connected to one of the 3.0 ports. Also, as shown in the image, the Serial library and Arduino variable have been added to the script.

```
import serial
arduino = serial.Serial('/dev/ttyUSB0', baudrate=57600)
```

Figure 3.3: Code for Serial communication at RPi 4

For the Arduino, has been used the USB for uploading the code (from the computer to the board). First, the code has been upload from the computer, including the part of the code shown in Figure 3.4, and then connected to the Raspberry Pi.

```
include <SoftwareSerial.h>
void setup(){
    Serial.begin(57600);
```

Figure 3.4: Code for Serial communication at Arduino

3.3 Software

The following section details the different Docker containers and services that will be executed to create our system.

3.3.1 MQTT Broker

As commented in Section 3.1, a Broker is needed for the implementation of the MQTT protocol.

Eclipse Mosquitto [6] is an open-source (EPL/EDL licensed) message broker that implements the MQTT protocol versions 5.0, 3.1.1 and 3.1. Mosquitto is designed for use on low-resource equipment, because it is lightweight and can be used on all devices, from low-power single-board computers to complete servers.

Also in DockerHub, there is available an official image of eclipse-mosquitto [4] which has been used to create the MQTT Broker.

3.3.2 MQTT Clients

The scenario has two MQTT Clients. A publisher/subscriber Client in Docker and another publisher/subscriber Client in RPi.

- **MQTT Client - Docker**
  This client is hosted in Docker and is in command of Publish that it wants to know the temperature and to Subscribe to the topic to receive the temperature. As for the MQTT Broker, there is an image available in DockerHub [5] from mosquitto Client that has been used.

- **MQTT Client - RPi 4**
  The MQTT Client hosted in the RPi 4 is in command of Subscribe to the topic that the MQTT Client hosted in Docker publishes and thus request the Arduino. Also, in Publish the value that the Arduino transfers to it. The RPi 4 has been configured with Raspberry Pi OS, and the mosquitto-customer package has been installed.

3.4 Summary

To summarise, the system is formed by an MQTT Broker, two MQTT Clients and the Arduino incorporated in the mBot Ranger. The different steps that proceed are as follows.

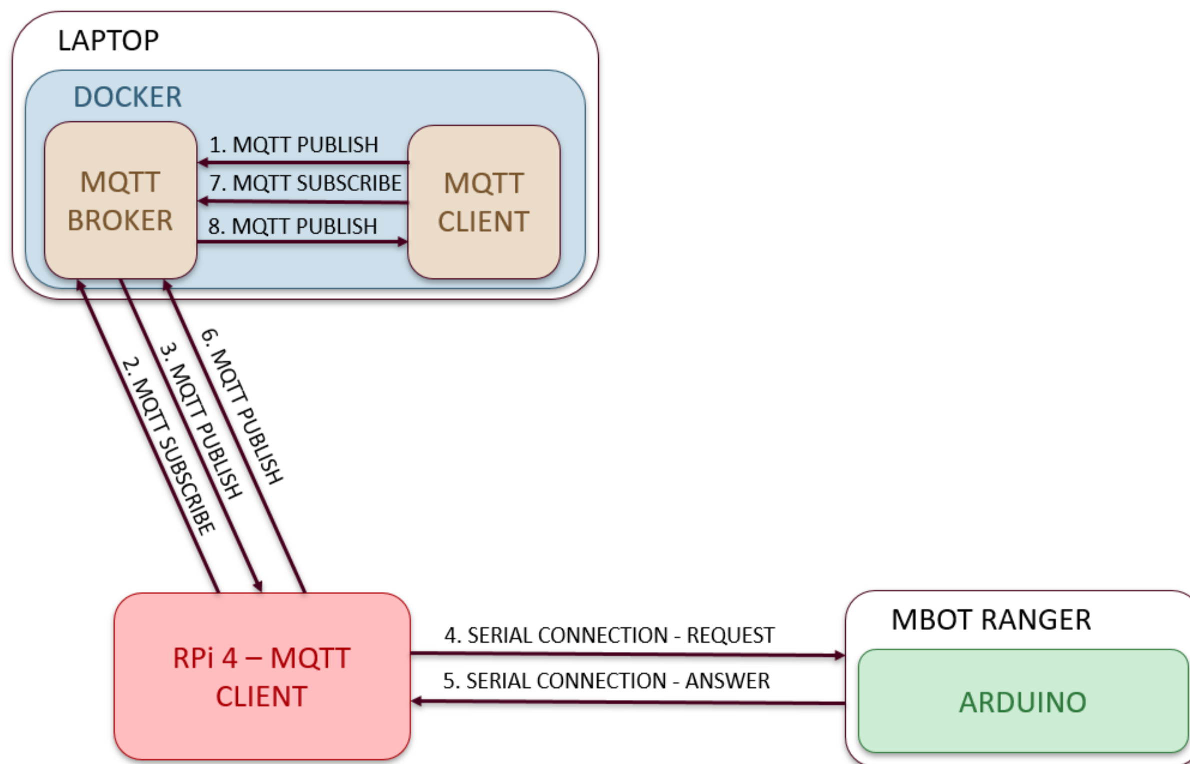


Figure 3.5: System design overview

First, MQTT Client in Docker does a publish that wants to know the temperature. Second, MQTT Client in RPi 4 subscribes to the MQTT Client topic in Docker, and then MQTT Broker transmits the MQTT Client consultation in Docker to MQTT Client in RPi 4. Below, the RPi 4 sends the request by serial connection to the Arduino, and it queries the sensor and returns the value to the RPi 4 by serial connection. Then, MQTT Client in RPi 4 Publishes temperature value. Finally, MQTT Client in Docker subscribes to the MQTT Client topic in RPi 4, and MQTT Broker transmits the MQTT Client in RPi 4 response. In Figure 3.5, there is an overview of the system.

4 Implementation

In this section are detailed how the MQTT elements of the system are installed and configured.

4.1 MQTT Broker container installation

First, the last version of the Dockerhub image is pulled. Then, the image has been run.

```
$ docker pull eclipse-mosquitto
```

Figure 4.1: Command to download the last image of Docker

```
$ docker run -it --name MQTTB -p 1883:1883 eclipse-mosquitto
1597849971: mosquitto version 1.6.9 starting
1597849971: Config loaded from /mosquitto/config/mosquitto.conf.
1597849971: Opening ipv4 listen socket on port 1883.
1597849971: Opening ipv6 listen socket on port 1883.
```

Figure 4.2: Execution to create the Broker's container

The argument -p 1883:1883 assigns MQTT's default port to the localhost (127.0.0.1) port 1883. No further configuration has been made, the Broker is now ready.

4.1.1 MQTT Broker container with prioritisation

For some scenarios, as is described in the next section, the Broker container with prioritisation has been used. For this reason, another Docker container has been created, following the same steps.

```
$ docker run -it --ulimit rtprio=90 --cap-add=sys_nice --name priMQTTB eclipse-mosquitto
1597851269: mosquitto version 1.6.9 starting
1597851269: Config loaded from /mosquitto/config/mosquitto.conf.
1597851269: Opening ipv4 listen socket on port 1883.
1597851269: Opening ipv6 listen socket on port 1883.
```

Figure 4.3: Execution to create the Broker's container with prioritisation

In this case, two flags have been assigned [8]:

–ulimit rtprio=90 : Maximum real-time priority allowed for the container.

–cap-add=sys_nice : Allows the container to raise process *nice* values, set real-time scheduling policies, set CPU affinity and other operations.

4.2 MQTT Client container installation

First, pull the last version of the Dockerhub image and run the image.

```
$ docker pull efrecon/mqtt-client

$ docker run -it --name MQTTC efrecon/mqtt-client
```

Figure 4.4: Command to download the last Docker image and execution to create the Client's container

In this image, the package *paho.mqtt.python* has been installed. To install it, you have first logged into the container as root and installed the python package. Also, has been

```
$ docker exec -it --user root MQTTCTEST /bin/sh
$ apk add python
```

Figure 4.5: Execution to install python package

added the folder *paho.mqtt* (cloned of Github) to the container and executed the following command:

```
$ cd paho.mqtt.python
$ python setup.py install
```

Figure 4.6: Execution to install *paho.mqtt.python*

Paho.mqtt.python is a python package that allows the MQTT Client to publish and subscribe through a python script.

4.2.1 MQTT Client container with prioritisation

As it happens with the Broker container, some scenario has also been used with the Client container with priority. Therefore, the *priMQTTC* container has been created:

```
docker run -it --ulimit rtprio=99 --cap-add=sys_nice --name priMQTTC efrecon/mqtt-client
```

Figure 4.7: Execution to run the Client's container with prioritisation

To this new container, has also been installed the package *paho.mqtt.python* in the same way that has been commented before.

4.3 MQTT Client installation in RPi

Through the terminal, the mosquitto-clients package has been installed in the RPi. This package is built into the Raspbian operating system.

As in the Docker-hosted Client, the package paho.mqtt.python is installed.

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get dist-upgrade
$ sudo apt-get install mosquitto-clients
```

Figure 4.8: Execution to install MQTT Client in RPi

```
$ cd paho.mqtt.python
$ python setup.py install
```

Figure 4.9: Execution to install *paho.mqtt.python*

In order to finish the setup, the configuration file *etc/mosquitto/mosquitto.conf* has been customised as follows in Figure 4.10. Then, the RPi has been configured for local use, and it will listen through port 1883.

```
pid_file /var/run/mosquitto.pid

user mosquitto

# Port to use for the default listener.
port 1883
listener 1883 192.168.0.175

# retained_persistence is a synonym for this option.
persistence true

# The filename to use for the persistent database, not including
# the path.
persistence_file mosquitto.db

# Location for persistent database. Must include trailing /
# Default is an empty string (current directory).
# Set to e.g. /var/lib/mosquitto/ if running as a proper service on Linux or
# similar.
persistence_location /mosquitto/data/
```

Figure 4.10: Configuration of the *file mosquito.conf*

4.4 Summary

To sum up, in this section, we have seen how easy it has been to configure the system thanks to the Dockerhub images for the Docker containers. Two containers have been created for the Broker, and two for the Client hosted in the Docker, configuring one of each with container priority.

On the other hand, setting up the MQTT Client on the RPi has also been easy since it has only meant installing a package of the Raspbian OS.

5 Evaluation

The purpose of this section is to carry out an evaluation of the system. This section aims to be able to identify whether the prioritisation of Docker containers or MQTT message QoS have an impact on latency. For this reason, a distributed application that implements a control loop between a sensor and an actuator. The system has been used to evaluate the impact of different resource sharing strategies of the docker system and MQTT message prioritisation on the cycle time of the control loop application.

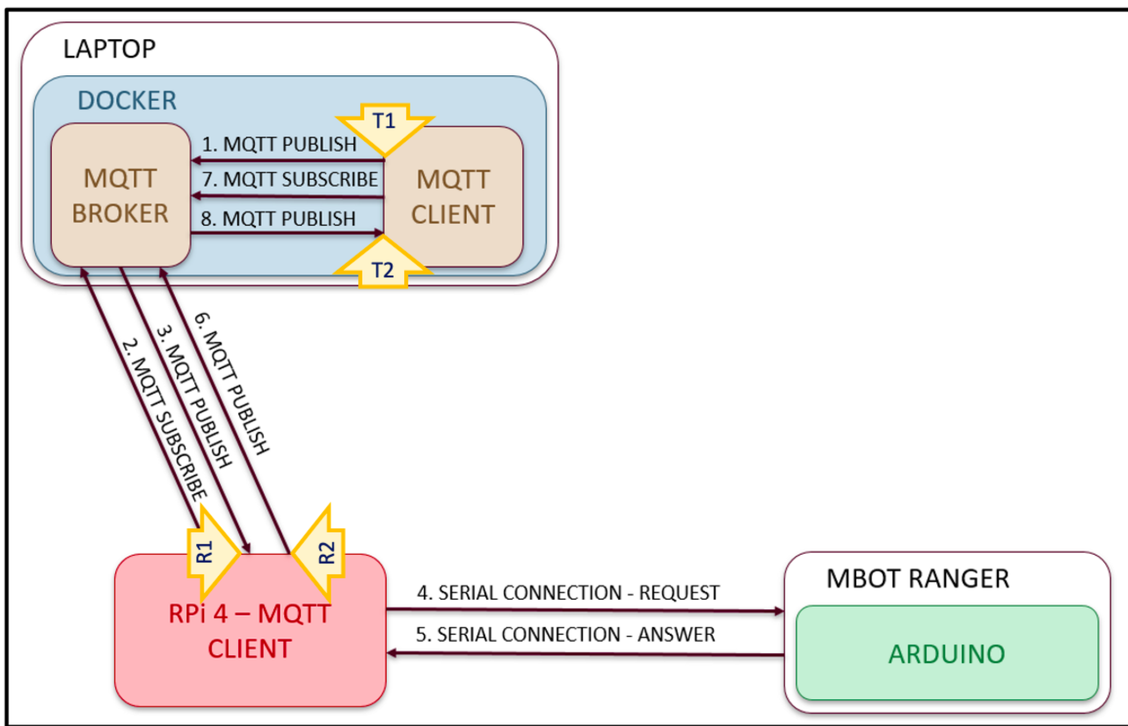


Figure 5.1: Diagram of the full setup with the evaluation metrics

The application performs a loop that ensures that the MQTT Client hosted in Docker publishes 100 messages to a given topic. Between each message, different interval times have been used (100ms, 50ms, 20ms, 10ms) where the publisher sleeps to issue the next publication message. The MQTT Client in the RPi is subscribed to the topic. When

the MQTT Client in the RPi receives the publish message from the Broker in a different docker container, it retrieves the corresponding sensor values from the Arduino through a serial connection. Then the MQTT Client in the RPi publishes the values in the topic Answer. The MQTT Client in Docker subscribes to the topic Answer and receives the publish response message from the Broker. A testbed has been created, which is configured differently for each test scenario in order to study the impact of different system configurations on the control loop cycle time. The MQTT client repeats 10 times the execution of the 100 different publishing cycles, thus obtaining 1000 latency samples.

Table 5.1: Description of the evaluation metrics and timestamps

Timestamp	Metric	Description
T1:		Initial time when the MQTT Client in Docker send the publishing message at the application layer.
T2:		End time of when the MQTT Client in Docker receives the response message at the application layer.
	T=T2-T1	Total response time.
R1:		Initial time when the MQTT Client hosted in RPi receives the publish message at the application layer.
R2:		End time of when the MQTT Client hosted in RPi send the publishing response message at the application layer.
	R=R2-R1	Time to get the sensor measurement from Arduino.
	O=T-R	MQTT processing and transmission time plus Docker overhead.

To correctly evaluate the latency time, the evaluation metrics and timestamps shown in the Table 5.1 have been defined. In the following section, the previously defined metrics of the entire system have been evaluated in the 9 scenes shown below. In each scenario different performance conditions are represented and studied.

5.1 Evaluation Scenario 0 – No Arduino

The aim of Scenario 0 is to abstract the processing of the Arduino emulating its ideal behaviour so that when the RPi receives the messages, it directly returns a random message. In such case, the processing time needed to gather the measurement from the sensor is not considered, thus the minimum time needed by the MQTT client to gather a response can be estimated. Therefore, this scenario has been created to evaluate the latency of the control loop which is referred to the time it takes for a message to be published by the MQTT Client hosted in Docker, to reach the MQTT Client hosted in the RPi passing through Broker and until the response message makes its way back to the MQTT Client.

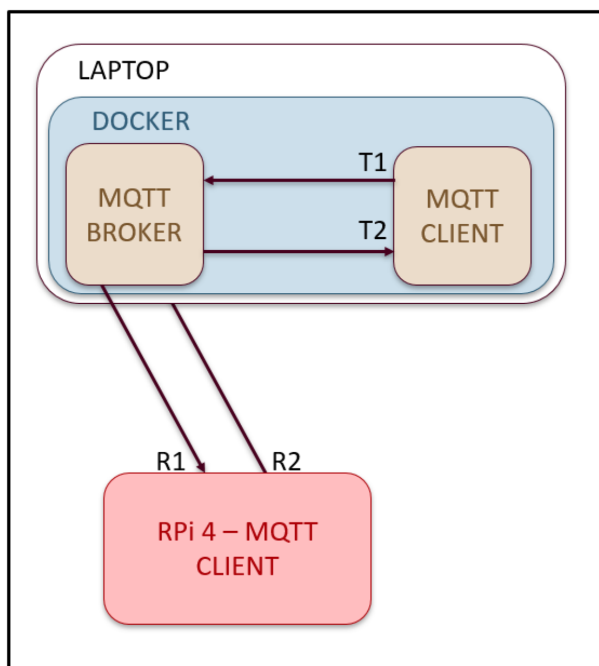


Figure 5.2: Diagram of the scenario without Arduino

This scenario, as shown in Figure 5.2, is composed of the MQTT Client hosted in Docker, the Broker hosted in a different Docker container and the MQTT Client hosted in RPi, which is the part of the system that works with the MQTT protocol.

Table 5.2: Table of the latencies gathered in Scenario 0

Interval time	100ms	50ms	20ms	10ms
T(min) [us]	8419	7337	7202	6740
T(max) [us]	21863	20245	26275	23668
T(avg) [us]	10265.67	9902.17	9777.08	10524.91
O(min) [us]	6684	7262	7141	6684
O(max) [us]	23610	20175	26207	23610
O(avg) [us]	10465.78	9827.79	9708.78	10465.78
R(avg) [us]	76.14	74.37	68.30	59.13

An overview of all latency times for this scenario is shown in the Table 5.2. The minimum total response time (T(min)) shows that the shorter the interval time between sending the publishing messages, the shorter the latency time. On the other hand, the average response time from MQTT protocol (O(avg)) represents the MQTT protocol latency time for this scenario is observed.

5.2 Evaluation Scenario 1 – Arduino 57600bps

The aim of Scenario 1 is to observe the latencies of the full setup. This scenario has been created in order to measure the latency referred to the time it takes a message to be published by the MQTT client hosted in Docker, to reach the sensor hosted in the Arduino passing through the Broker and until the response message makes its way back to the MQTT Client.

This scenario, as shown in Figure 5.3, it is composed by the MQTT Client hosted in Docker, the Broker, the MQTT Client hosted in the RPi and the Arduino with a baud rate of 57600bps.

An overview of all the evaluation metrics for this scenario is shown in Table 5.3.the average response time from Arduino R(avg) shows that the Arduino takes about 4ms to receive,

process and respond to the request issued by the Raspberry Pi. By comparing with the configuration that emulates the behaviour of the Arduino (see section 5.1), the contribution of the Arduino to the total latency can be seen.

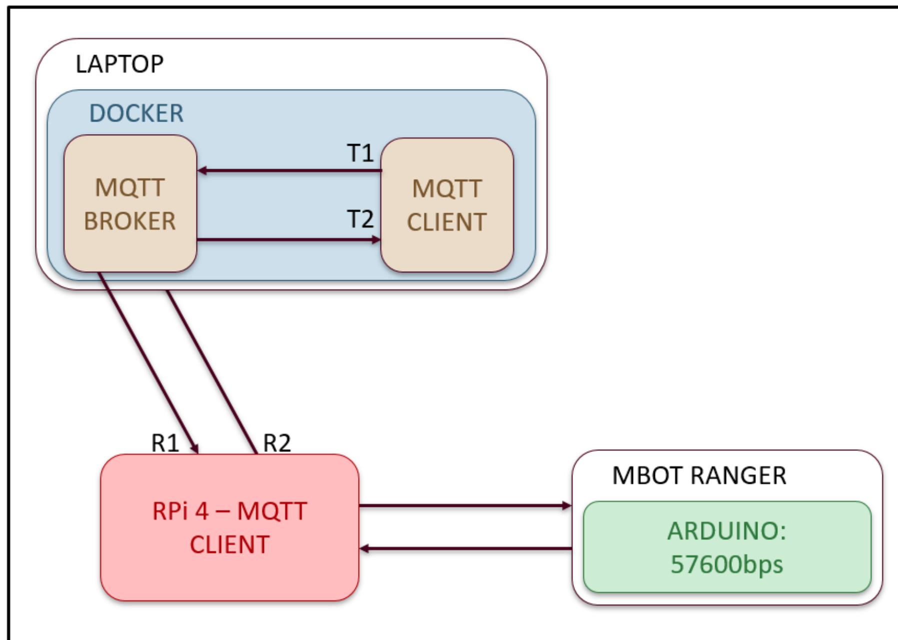


Figure 5.3: Diagram of scenario 1

Table 5.3: Table of the latencies gathered in Scenario 1

Interval time	100ms	50ms	20ms	10ms
T(min) [us]	11523	10974	11178	
T(max) [us]	27125	26636	29049	
T(avg) [us]	13865.49	13394.78	13024.8	Loss
O (min) [us]	7159	6385	6858	of
O(max) [us]	22856	22291	24667	packets
O(avg) [us]	9560.85	9105.1	8744.7	
R(avg) [us]	4304.64	4289.67	4280.1	

It has also been observed, that when 100 messages are published with a 10ms interval, the MQTT Client hosted in Docker receives fewer responses than sending out publishing

messages. By careful examination of the log files, we found that messages queued up in the system as the minimum processing and transmitting time for the whole control loop is larger than 10 ms. As messages are arriving faster than can be processed, the Arduino answered several times multiple requests, leading to the situation where we received fewer responses as some have been skipped. We indicate those in the Table 5.3 as packet loss. For this reason, it is concluded that in this situation, there is packet loss.

5.3 Evaluation Scenario 2 – Docker Stress

The aim of Scenario 2 is to see CPU stress on the MQTT Broker. Another container denoted as Stress has been deployed on the Docker host where the MQTT Broker is deployed. Therefore, the scenario has been created to be able to calculate the latency referred to the time it takes for a message to be published by the MQTT Client hosted in Docker, to reach the sensor hosted in the Arduino passing through the Broker and until the response message makes its way back to the MQTT Client when there is stress in the system.

This scenario, as shown in Figure This scenario, as shown in Figure 5.4, it is composed of the MQTT Client hosted in Docker, the Broker, the MQTT Client hosted in RPi, the Arduino with a baud rate of 57600bps and, in addition, another Docker container has been added that shares CPU with the rest of the containers. The impact that this new container generates is a pressure in the CPU of the Docker behaviour, causing stress in the system.

This stress container has been created by executing the command *docker run progrium/stress* and configuring the following options:

-cpu 3 : spawn 3 workers spinning on sqrt().

-io 3 : spawn 3 workers spinning on sync().

-vm 5 :	spawn 5 workers spinning on malloc()/free().
-vm-bytes 128M :	malloc 128M bytes per vm worker.
-timeout 4000s :	timeout after 4000s seconds.

It has therefore been generated as follows: *docker run –rm -it progrium/stress –cpu 3 –io 3 –vm 5 –vm-bytes 128M –timeout 4000s*

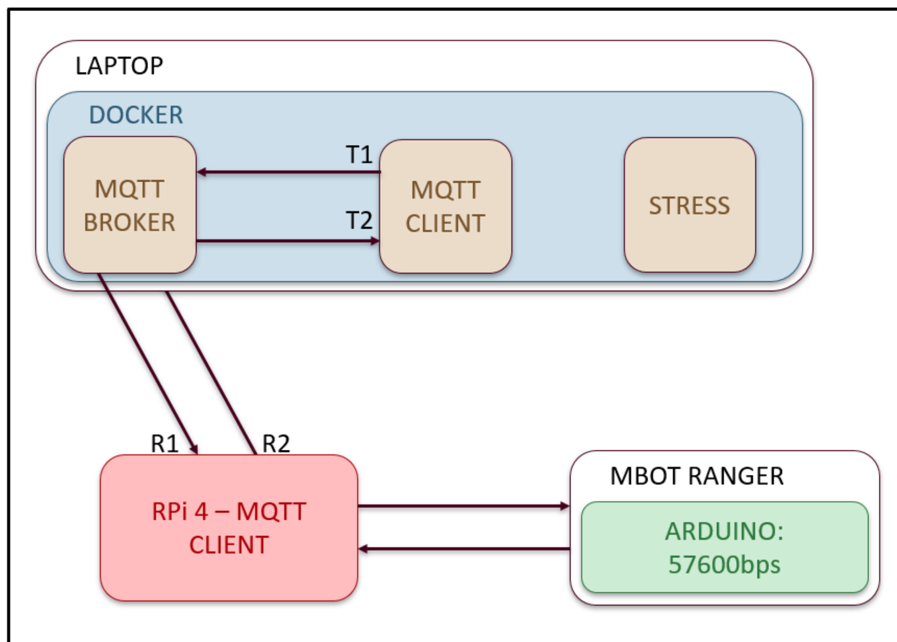


Figure 5.4: Diagram of scenario 2

Table 5.4: Table of the latencies gathered in Scenario 2

Interval time	100ms	50ms	20ms	10ms
T(min) [us]	12176	12547	11904	
T(max) [us]	85065	65884	50272	
T(avg) [us]	16780.93	17418.42	18032.59	Loss
O(min) [us]	7807	8254	7606	of
O(max) [us]	80739	61643	46028	packets
O(avg) [us]	12507.46	13124.32	13747.89	
R(avg) [us]	4273.46	4294.1	4284.7	

An overview of all evaluation metrics for this scenario is shown in the Table 5.4. By looking at the O(avg) time, it can be seen that the average latency of the processing and transmission of the MQTT publish and subscribe messages is 13ms. This latency time is very high, by comparing with the full setup, when not under CPU pressure (see section 5.2), and considering that this is a lightweight messaging protocol. The large jitter in the message processing is due to CPU stress on the Broker and the MQTT client, which leads to overall large and varying MQTT protocol processing latency. However, observing the R(avg) time, it can be seen that the time it takes Arduino to receive, process and respond to the request is 4ms, and therefore the stress, as expected, is not affecting it.

Besides, it has been observed that when 100 messages are published with a 10ms interval, the MQTT Client hosted in Docker receives fewer responses than sending out publishing messages. By careful examination of the log files, we found that messages queued up in the system as the minimum processing and transmitting time for the whole control loop is larger than 10 ms. As messages are arriving faster than can be processed, the Arduino answered several times multiple requests, leading to the situation where we received fewer responses as some have been skipped. For this reason, it is concluded that in this situation, there is packet loss.

5.4 Evaluation Scenario 3 – Docker Stress without Arduino

The aim of Scenario 3 is to see how stress affects the system when the processing of the Arduino is abstracted emulating its ideal behaviour, thus when the RPi receives the messages it directly returns a random message. Therefore, this scenario has been created to calculate the latency referred to the time it takes for a message to be published by the MQTT Client hosted in Docker, to reach the MQTT Client hosted in the RPi passing through Broker and until the response message makes its way back to the MQTT Client when there is stress in the system.

This scenario, as shown in Figure 5.5, it is composed of the MQTT Client hosted in the Docker, the Broker, the MQTT Client hosted in the RPi and, in addition, another Docker container has been added that shares CPU with the rest of the containers. The impact that this new container generates is a pressure in the CPU of the Docker behaviour, causing stress in the system. This stress container has been created as in Scenario 2 (section 5.3).

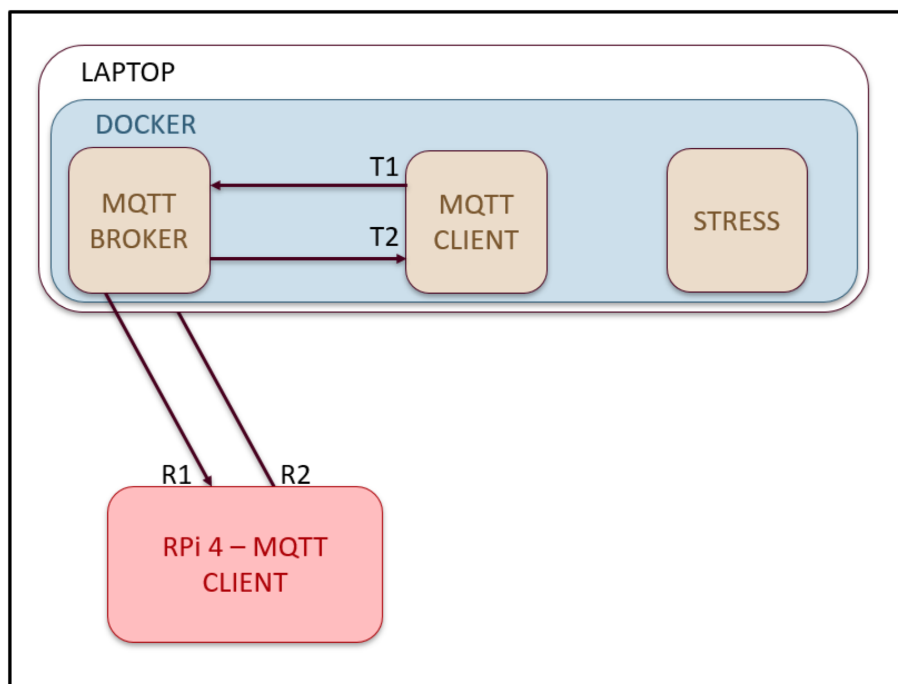


Figure 5.5: Diagram of scenario 3

Table 5.5: Table of the latencies gathered in Scenario 3

Interval time	100ms	50ms	20ms	10ms
T(min) [us]	9926	8535	7499	7550
T(max) [us]	115348	117048	50147	47490
T(avg) [us]	18171.98	12409.14	11597.95	13884.73
O(min) [us]	9847	8459	7431	7500
O(max) [us]	115269	116924	50072	47433
O(avg) [us]	18094.90	12334.70	11527.55	13825.51
R(avg) [us]	77.08	74.44	70.40	59.22

An overview of all evaluation metrics for this scenario is shown in the Table 5.5. By looking at the O(avg) time, it can be seen that the average MQTT processing and transmission time of the MQTT protocol reaches 18ms. This latency time is very high, by comparing with the configuration that emulates the ideal behaviour of the Arduino where there is no stress in the system (see section 5.1) and considering that this is a lightweight messaging protocol. The main reason for the large and varying latency is the concurrent stress container that issues CPU pressure on the MQTT broker and MQTT client in the docker system. The stress causes this that the Docker environment is submitted.

Besides, by looking at the R(avg) time, it can be seen that since there is no Arduino in this scenario, it takes about 0.7ms for the RPi to receive and respond with a random message to the request, and therefore the stress, as expected, is not affecting it.

5.5 Evaluation Scenario 4 – MQTT Broker Prioritisation

The aim of Scenario 4 is to see how the prioritisation of the Broker's container reacts to a system under stress. For this reason, the scenario has been created to be able to evaluate the latency referred to the time it takes for a message to be published by the MQTT Client hosted in Docker, to reach the sensor hosted in the Arduino passing through the Broker and until the response message makes its way back to the MQTT Client when there is stress in the system.

This scenario, as shown in Figure 5.6, is composed of the MQTT Client hosted in Docker, the Broker with priority in Docker, the MQTT Client hosted in the RPi, the Arduino with a baud rate of 57600bps and, in addition, another Docker container denoted as Stress has been added that shares CPU with the rest of the containers. The impact that this new container generates is a pressure in the CPU of the Docker behaviour, causing stress in the system. This stress container has been created as in Scenario 2 (section 5.3).

The prioritisation has been achieved configuring the following flags to the *docker run* command of the Broker container:

–ulimit rtprio=90 : Maximum real-time priority allowed for the container.

–cap-add=sys_nice : Allows the container to raise process *nice* values, set real-time scheduling policies, set CPU affinity and other operations.

It has therefore been achieved as follows: *docker run -it –ulimit rtprio=90 –cap-add=sys_nice -name priMQTTB eclipse-mosquitto*

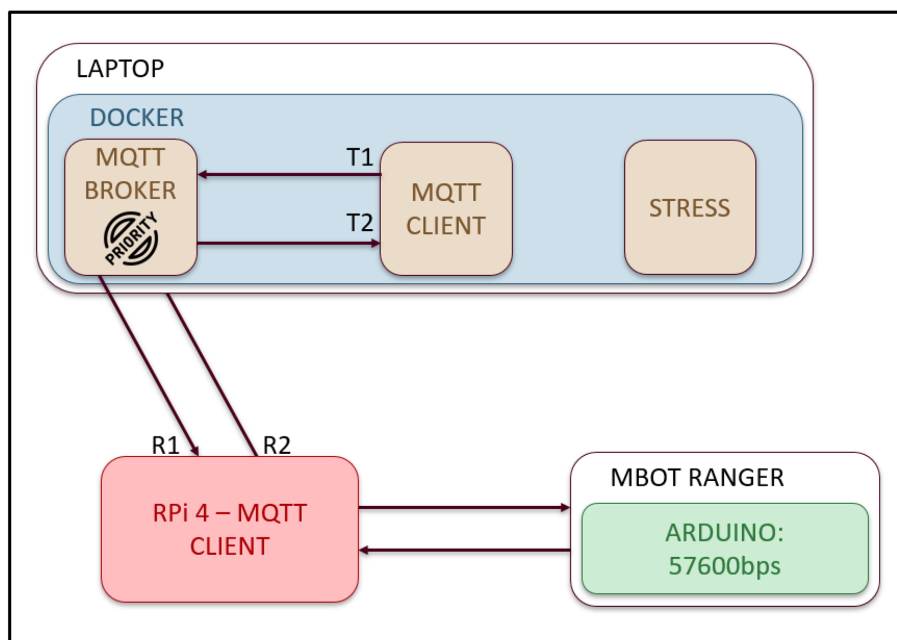


Figure 5.6: Diagram of scenario 4

An overview of all the evaluation metrics for this scenario is shown in Table 5.6. By looking at the O(avg) time, it can be seen that the average MQTT message processing and transmission are around 11ms. This latency time is high, by comparing with the full setup (see section 5.2) and considering that it is a light messaging protocol. Nevertheless, by

comparing with the full setup with stress (see section 5.3), it can be seen that the system is more stable as the result of the average latency times in the three intervals are similar, thanks to the prioritisation of the Broker's container. Also, the maximum latency is still higher compared to when no stress is exercised, which may be due to the non-prioritisation of the MQTT client container.

Table 5.6: Table of the latencies gathered in Scenario 4

Interval time	100ms	50ms	20ms	10ms
T(min) [us]	11067	10712	10845	
T(max) [us]	46398	45772	41477	
T(avg) [us]	15400.99	15252.33	15448.03	Loss
O(min) [us]	6726	6372	6630	of
O(max) [us]	42097	41217	37184	packets
O(avg) [us]	11106.4	10964.30	11157.39	
R(avg) [us]	4294.59	4287.87	4290.64	

Moreover, it has been observed that when 100 messages are published with a 10ms interval, the MQTT Client hosted in Docker receives fewer responses than sending out publishing messages. By careful examination of the log files, we found that messages queued up in the system as the minimum processing and transmitting time for the whole control loop is larger than 10 ms. As messages are arriving faster than can be processed, the Arduino answered several times multiple requests, leading to the situation where we received fewer responses as some have been skipped. For this reason, it is concluded that in this situation, there is packet loss. However, observing the R(avg) time, it can be seen that the time it takes Arduino to receive, process and respond to the request issued by the Raspberry Pi is 4ms, and therefore the stress, as expected, is not affecting it.

5.6 Evaluation Scenario 5 – MQTT Broker and MQTT Docker Client Prioritisation

The aim of Scenario 5 is to see how if the MQTT Client and Broker prioritisation impact the latency of the control loop when the system is under stress. For this reason, the scenario has been created to be able to calculate the latency referred to the time it takes for a message to be published by the MQTT Client hosted in Docker, to reach the sensor hosted in the Arduino passing through the Broker and until the response message makes its way back when there is stress in the system.

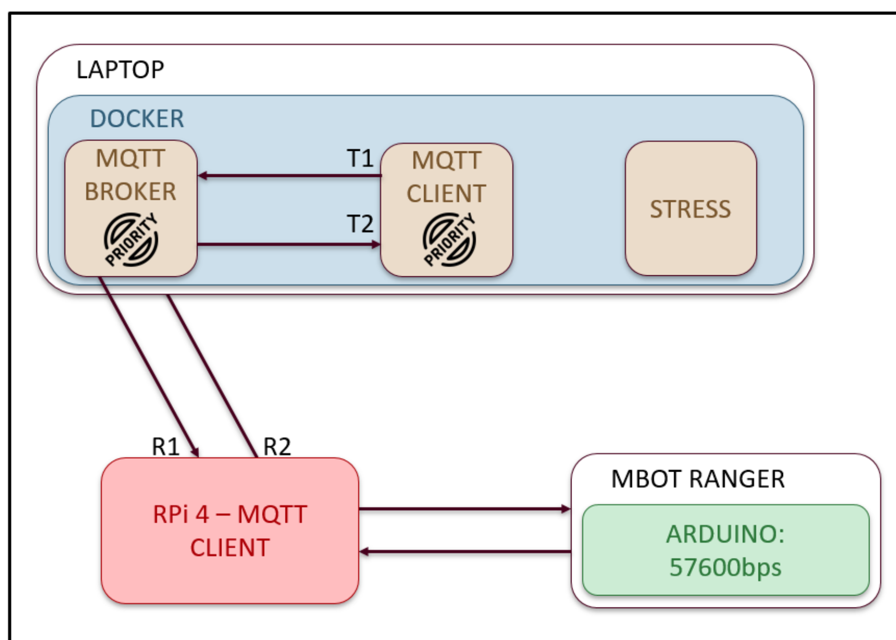


Figure 5.7: Diagram of scenario 5

This scenario, as shown in Figure 5.7, it is composed of the MQTT Client hosted in Docker with container prioritisation, the Broker with container prioritisation, the MQTT Client hosted in the RPi, the Arduino with a baud rate of 57600bps and, in addition, another Docker container has been added that shares CPU with the rest of the containers. The

impact that this new container generates is a pressure in the CPU of the Docker behaviour, causing stress in the system. This stress container has been created as in Scenario 2 (section 5.3).

The container prioritisation has been achieved following the explained flags in scenario 4 (section 5.5). It has therefore been achieved for the Broker as follows: *docker run -it –ulimit rtprio=90 –cap-add=sys_nice -name priMQTTB eclipse-mosquitto* . And has been achieved for the MQTT Client as follows: *docker run -it –ulimit rtprio=90 –cap-add=sys_nice -name priMQTTC efrecon-mqtt-client*

Table 5.7: Table of the latencies gathered in Scenario 5

Interval time	100ms	50ms	20ms	10ms
T(min) [us]	11307	5828	11203	
T(max) [us]	45534	57113	37628	
T(avg) [us]	15732.84	15264.91	15345.23	Loss
O(min) [us]	7052	1300	6910	of
O(max) [us]	41278	52784	33303	packets
O(avg) [us]	11439.44	10978.75	11059.27	
R(avg) [us]	4293.40	4286.26	4285.96	

An overview of all evaluation metrics for this scenario is shown in the Table 5.7. By looking at the O(avg) time, it can be seen that the average of the MQTT message processing and transmission time is around 15ms. This latency time is very high, by comparing with the full setup (see section 5.2) and considering that it is a light messaging protocol. Nevertheless, by comparing with the full setup with stress (see section 5.3), it can be seen that the system is more stable as the result of the average latency times in the three intervals are similar, thanks to the prioritisation of the Broker's and the MQTT Client's containers.

By comparing with the evaluation metrics with the ones in the Table 5.6 of Scenario 5, it can be seen that practically the same latency times are obtained. Therefore, no benefit

has been obtained by prioritising the MQTT Client container, since the maximum latency is still higher compared to when no stress is exercised.

Moreover, it has been observed that when 100 messages are published with a 10ms interval, the MQTT Client hosted in Docker receives fewer responses than sending out publishing messages. By careful examination of the log files, we found that messages queued up in the system as the minimum processing and transmitting time for the whole control loop is larger than 10 ms. As messages are arriving faster than can be processed, the Arduino answered several times multiple requests, leading to the situation where we received fewer responses as some have been skipped. For this reason, it is concluded that in this situation, there is packet loss. However, observing the R(avg) time, it can be seen that the time it takes Arduino to receive, process and respond to the request is 4ms, and therefore the stress, as expected, is not affecting it.

5.7 Evaluation Scenario 6 – MQTT Broker Prioritisation without Arduino

The aim of Scenario 6 is to see if the Broker prioritisation impacts the latency of the control loop when the system is under stress and, when it abstracts the processing of the Arduino emulating its ideal behaviour; thus when the RPi receives the messages, it directly returns a random message. Therefore, the scenario has been created to be able to measure the observed latency referred to the time it takes for a message to be published by the MQTT Client hosted in Docker, to reach the sensor hosted in the Arduino passing through the Broker and until the response message makes its way back when there is stress in the system.

This scenario, as shown in Figure 5.8, it is composed of the MQTT Client hosted in Docker, the Broker with container prioritisation, the MQTT Client hosted in the RPi and,

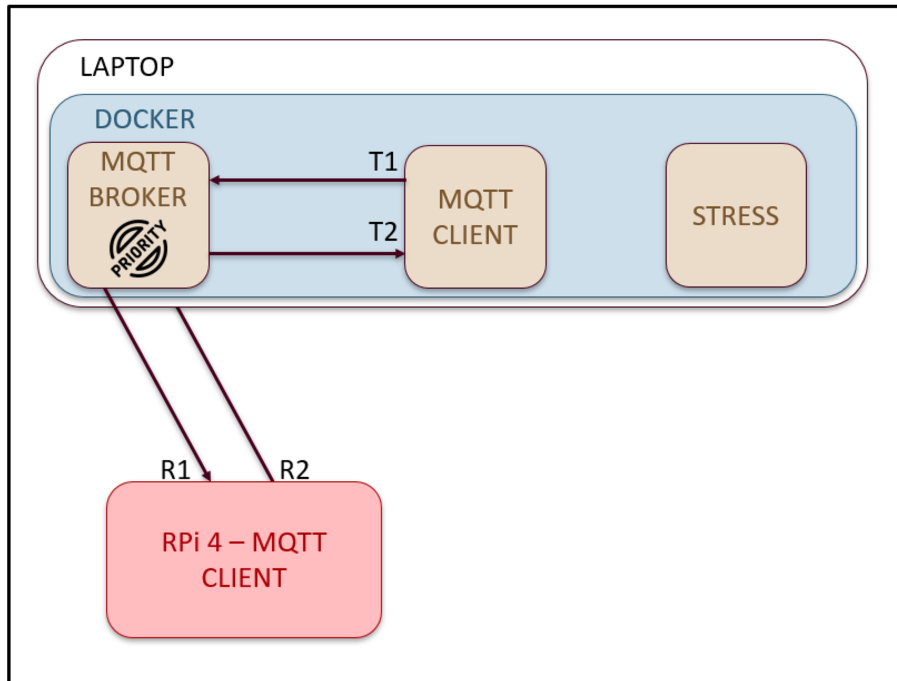


Figure 5.8: Diagram of scenario 6

in addition, another Docker container has been added that shares CPU with the rest of the containers. The impact that this new container generates is a pressure in the CPU of the Docker behaviour, causing stress in the system. This stress container has been created as in Scenario 2 (section 5.3).

Table 5.8: Table of the latencies gathered in Scenario 6

Interval time	100ms	50ms	20ms	10ms
T(min) [us]	7853	8667	7730	7249
T(max) [us]	36828	33758	32276	34296
T(avg) [us]	12073.55	13308.27	11271.43	11903.02
O(min) [us]	7789	8600	7655	7195
O(max) [us]	36701	33632	32196	34171
O(avg) [us]	11995.12	13235.7	11202.51	11845.06
R(avg) [us]	78.42	72.56	68.91	57.95

An overview of all latency times for this scenario is shown in the Table 5.8. By looking

at the O(avg) time, it can be seen that the MQTT protocol latency time is around 12ms. This latency time is high, by comparing with the same setup but without stress (see section 5.1) and considering that it is a light messaging protocol. Nevertheless, by comparing with the setup with stress but without Arduino (see section 5.4), it can be seen that the system is more stable as the result of the latency times in the three intervals are similar, thanks to the prioritisation of the Broker's container.

Besides, by looking at the R(avg) time, it can be seen that since there is no Arduino in this scenario, it takes about 70us for the RPi to receive and respond with a random message to the request, and therefore the stress, as expected, is not affecting it.

5.8 Evaluation Scenario 7 - MQTT Stresser

The aim of Scenario 7 is to see how the Broker's stress affects the system. For this reason, the scenario has been created to be able to evaluate the latency referred to the time it takes for a message to be published by the MQTT Client hosted in Docker, to reach the sensor hosted in the Arduino passing through the Broker and until the response message makes its way back to the MQTT Client when there is stress in the system.

This scenario, as shown in Figure 5.9, it is composed of the MQTT Client hosted in Docker, the Broker, the MQTT Client hosted in the RPi, the Arduino with a baud rate of 57600bps and, in addition, another Docker container has been added which generates stress to the Broker. In contrast to Scenario 2, where the stress container only created CPU pressure on the MQTT Broker and MQTT Client, this time we used a container which creates an MQTT stress test by sending many MQTT messages to the Broker in short time to stress the MQTT messaging system.

This new container that generates stress in the Broker has been created by executing the

command *run inovex/mqtt-stresser* and configuring the following options:

-broker tcp://192.168.0.107:1883 :	Define Broker URL (IP and access port)
-num-clients 100 :	Number of concurrent clients
-num-messages 10 :	Number of messages shipped by Client
-rampup-delay 1s :	Time between batch rampups
-rampup-size 10 :	Size of rampup batch
-global-timeout 180s :	Timeout spanning all operations
-timeout 20s :	Timeout for pub/sub loop

It has therefore been generated as follows: *run inovex/mqtt-stresser -broker tcp://192.168.0.107:1883 -num-clients 100 -num-messages 10 -rampup-delay 1s -rampup-size 10 -global-timeout 180s -timeout 100s*

In this set of experiments, the MQTT client sends messages every 20 ms only. What happened during the experiment is that the connection was lost before sending 100 consecutive messages. The reason is that the new container has caused the Broker to be under much stress and not responding.

5.9 Evaluation Scenario 8 - MQTT Stresser with MQTT Docker Client Prioritsation

The aim of Scenario 8 is to see how the prioritisation of the MQTT Client's container and the different QoS values affect the system when there is stress in the Broker. Therefore, the scenario has been created to be able to calculate the latency referred to the time it

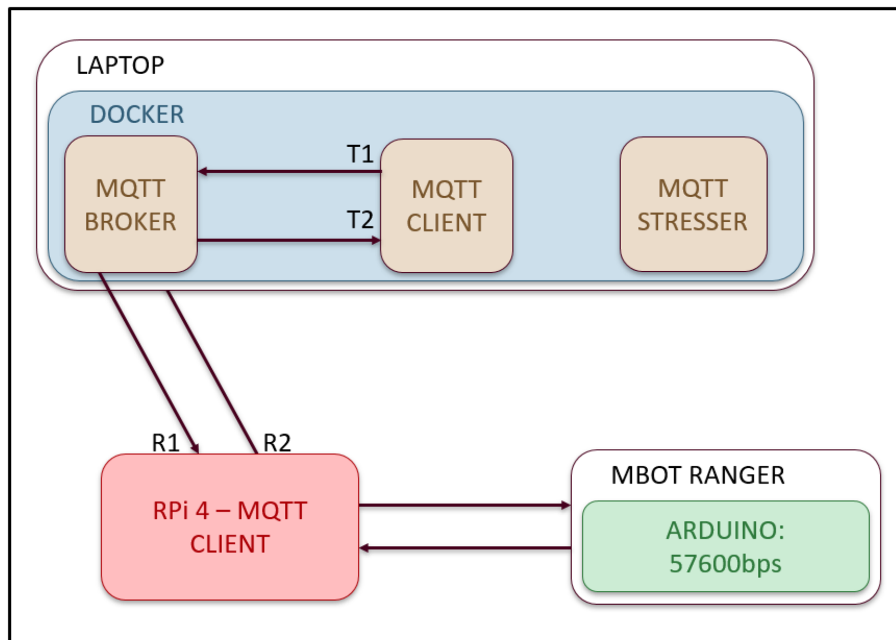


Figure 5.9: Diagram of scenario 7

takes for a message to be published by the MQTT Client hosted in Docker, to reach the sensor hosted in the Arduino passing through the Broker and until the response message makes its way back when there is stress in the system.

This scenario, as shown in Figure 5.10, is composed of the MQTT Client hosted in Docker with priority, the Broker, the MQTT Client hosted in the RPi, the Arduino with a baud rate of 57600bps and, in addition, another Docker container has been added which generates stress to the Broker. This stress container has been created as in Scenario 7 (section 5.8).

The experiment with this scenario has been to send messages to every 20ms and changing the QoS value. An overview of all of evaluation metrics for this scenario is shown in the Table 5.9. It can be seen that when QoS is set to 0, both the client and MQTT stress container issues messages having the same priority. Consequently, as the MQTT stress tester floods the Broker with messages, the MQTT Client, which implements the control loop, loses the connection. On the other hand, when the MQTT client prioritises

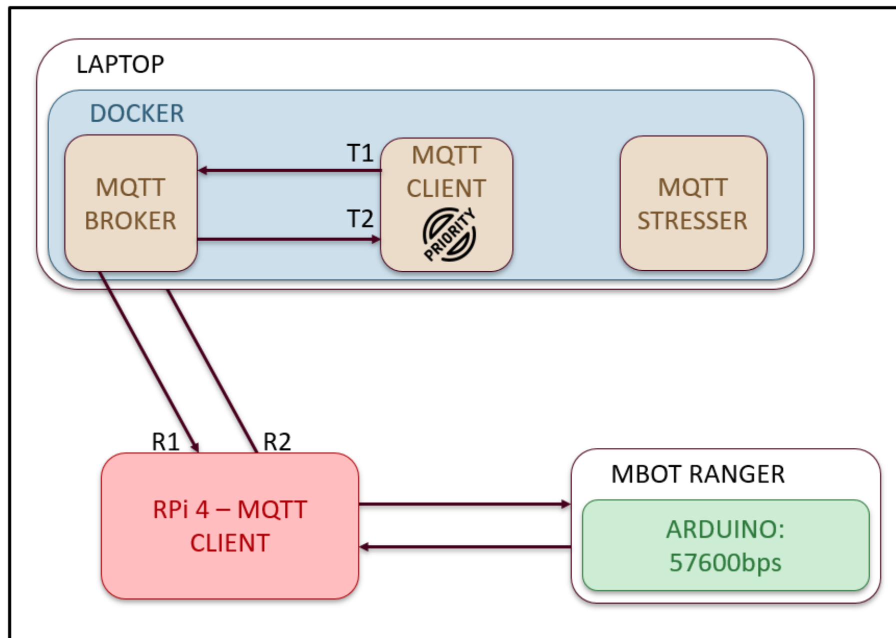


Figure 5.10: Diagram of scenario 8

the MQTT messages using QoS =1 or 2, the MQTT clients messages are prioritised at the brokers MQTT stack. Therefore, in those scenarios, the average MQTT message processing and transmission times are much smaller and stable, showing the benefit of QoS prioritisation.

Table 5.9: Table of the latencies gathered in Scenario 8

QoS	0	1	2
T(min) [us]		11336	22450
T(max) [us]		71455	89561
T(avg) [us]	Loss	16082.64	36286.04
O(min) [us]	of	7061	18124
O(max) [us]	packets	67217	85305
O(avg) [us]		11829.24	31997.53
R(avg) [us]		4285.52	4288.51

Additionally, as the Client waits to send the next message to receive the acknowledgement, much higher latencies are obtained. By looking at the O(avg) time, it can be seen that the

latency time when QoS is 2 is much higher than when it is 1. This delay is because with QoS 2, for each Publish message, a sequence of 4 messages is sent since is the highest level of service and the safest.

5.10 Summary

To summarise, nine different scenarios have been set up to evaluate the MQTT protocol.

In the scenarios that include Arduino, it can be seen that it has a latency time of about 4ms. Also, when publications are sent every 10ms, queues are building up leading to a situation where the MQTT messages are combined, leading to packet loss.

Furthermore, two different types of stress have been evaluated: stress for the Docker environment and stress that affects the MQTT system, more specifically to the Broker. On the one hand, it has been observed that in order to try to minimise a little the effect of stress in the Docker environment, priority should be given to the Broker's container or the MQTT Client's. On the other hand, in order for the system to continue working when the Broker is under much stress, QoS has had to be configured to prioritise MQTT messages of the Client implementing the control loop, and the MQTT Client hosted in the Docker must also be prioritised. This configuration has not reduced the latency times, due to the acknowledgement, but it provides reliability to the system, guaranteeing that no message is lost.

6 Results Analysis

The purpose of this section is to compare the different scenarios, explained in the Section 5. In this way, it will be possible to analyse whether Arduino's behaviour or prioritising Docker containers has an impact on the performance of the MQTT protocol.

In order to carry out the analysis, 5 cumulative distribution function (CDF) graphs have been created, in which it has been attempted to isolate these performance aspects.

6.1 Arduino 57600 vs No Arduino

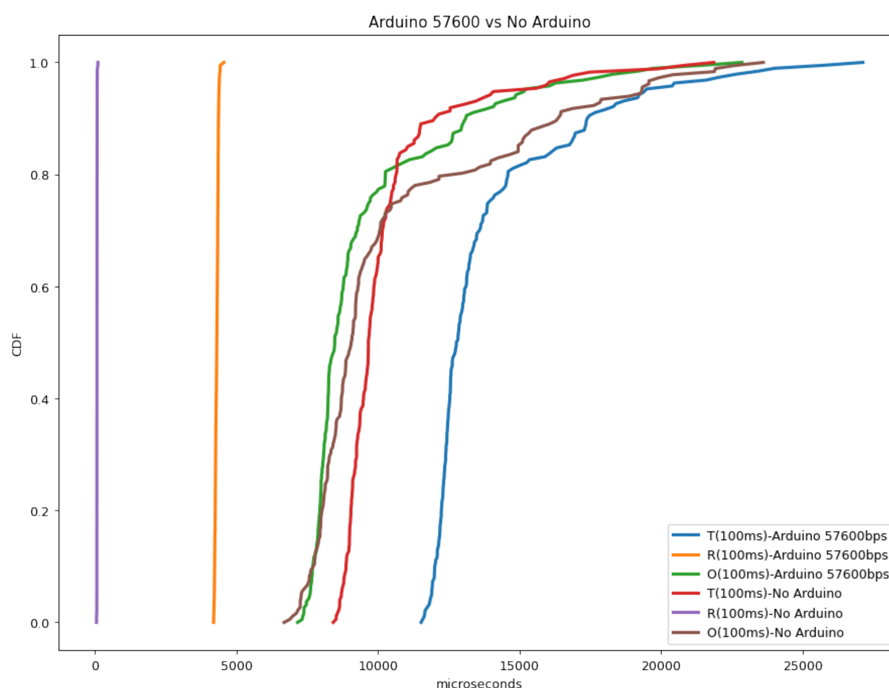


Figure 6.1: CDF graph comparing latencies of Scenarios 0 and 1

This analysis aims to compare scenario 0 (section 5.1) with scenario 1 (section 5.2) to be able to analyse the impact of the Arduino's contribution on the behaviour of the MQTT protocol.

By looking at the R metric of the scenario with Arduino in the Figure 6.1, observing the R metric of the scenario with Arduino, it can be seen how 100% of the messages the Arduino takes 4ms to receive, process and respond to the request issued by the Raspberry Pi. On the other hand, by looking at the R metric of the scenario without Arduino, it can be seen how 100% of the messages the Raspberry Pi takes about 70us in response to the request issued. Therefore, it can be seen how the Arduino contributes to the total latency

However, by looking at O metric, it can be seen how the MQTT protocol takes less than 8ms to process in 20% of the messages, for both scenarios. On the other hand, from this point, the observation that can be made is that the latency times of the MQTT protocol were generally lower in the scenario with physical Arduino compared to the scenario that simulates its ideal behaviour. This is because thanks to the processing time of Arduino consulting the sensor, the Broker has more time between each response message.

6.2 Arduino 57600 + Docker Stress vs No Arduino + Docker Stress

The objective of this analysis is to compare scenario 2 (section 5.3) with scenario 3 (section 5.4), in order to analyse the impact of the Arduino's contribution on the behaviour of the MQTT protocol when there is a new container that generates a pressure on the CPU from the Docker's behaviour, causing stress in the system.

By looking at the R metric of the Arduino scenario in the Figure 6.2, it can be seen how 100% of the messages the Arduino takes 4ms to receive, process and respond to the request issued by the Raspberry Pi. On the other hand, by looking at the R metric of the scenario without Arduino, it can be seen how 100% of the messages the Raspberry Pi takes about 70us in response to the request issued. Therefore, it can be seen how the Arduino contributes to the total latency.

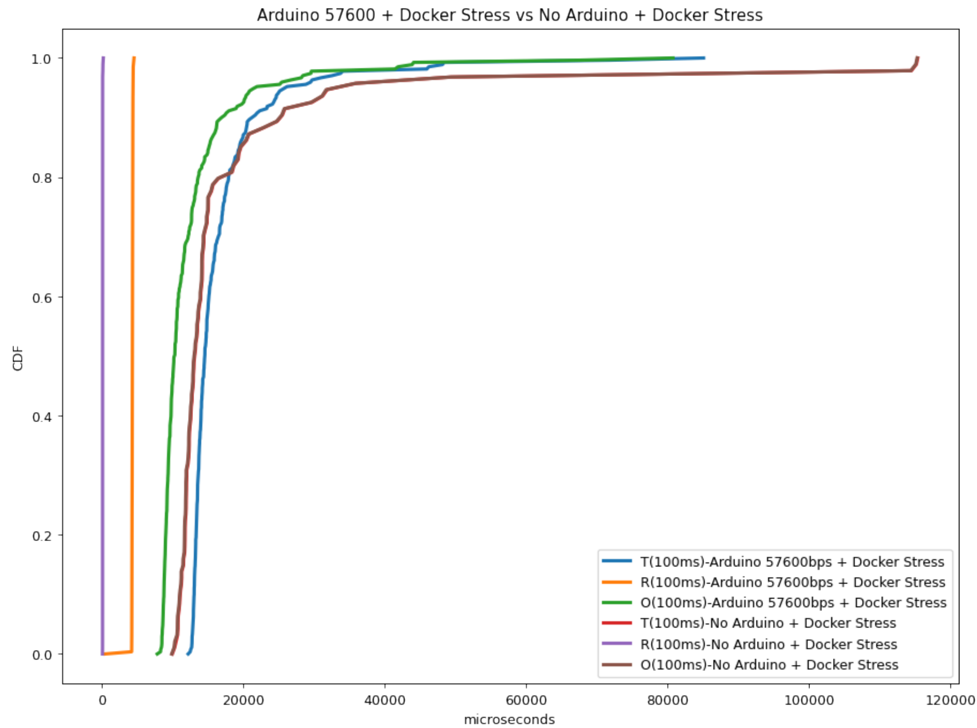


Figure 6.2: CDF graph comparing comparing latencies of Scenarios 2 and 3

However, by looking at the O metric, it is possible to see how the MQTT protocol takes less than 10ms to process in 60% of the messages, for the scenario with Arduino. On the other hand, in the scenario without Arduino, it can be seen how the MQTT protocol takes less than 15ms to process in 60% of the messages. From these data, the observation that can be made is that the latency times of the MQTT protocol were generally lower in the scenario with physical Arduino compared to the scenario that simulates its ideal behaviour. It is because thanks to the processing time of Arduino consulting the sensor, the Broker has more time between each response message.

6.3 Arduino 57600 vs Arduino 57600 + Docker Stress

This analysis aims to compare scenario 1 (section 5.2) with scenario 2 (section 5.3) in order to analyse the impact on the MQTT protocol of system stress caused by CPU pressure from Docker's behaviour.

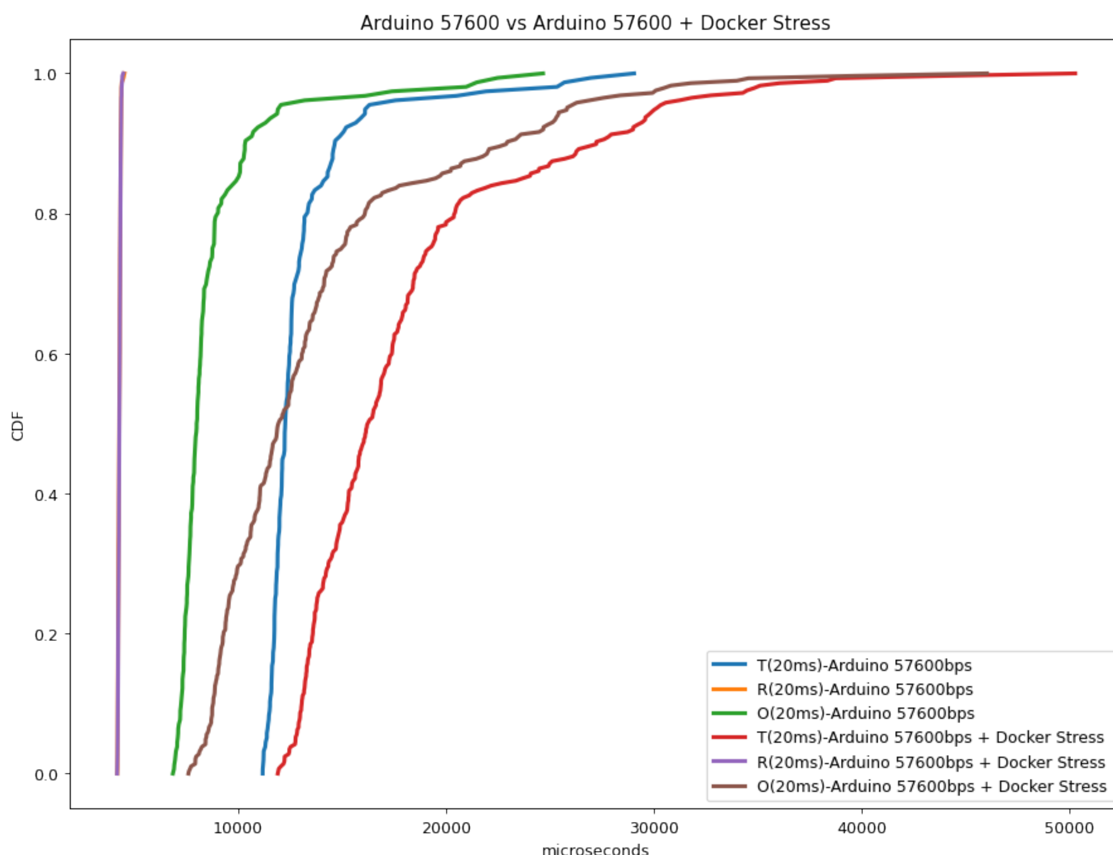


Figure 6.3: CDF graph comparing latencies of Scenarios 1 and 2

By looking at the O metric of the Arduino scenario in the Figure 6.3, it can be seen how the MQTT protocol takes less than 9ms to process in 80% of the messages, in the scenario that there is no stress. On the other hand, in the scenario with stress, it can be seen how the MQTT protocol takes less than 15ms to process in 80% of the messages. From these values and from the inclination of the graphs, the observation that can be made is that

the latency times of the MQTT protocol were generally lower in the no-stress scenario compared to the stress scenario. As in Docker the Broker's container and the MQTT Client's one are located, it happens that both are affected by stress, having a pressure in the CPU, causing them to be slower processing the messages.

6.4 Arduino 57600 + Docker Stress vs MQTT Broker Prioritisation vs MQTT Broker and MQTT Docker Client Prioritisation

This analysis aims to analyse the impact on the MQTT communication protocol of the prioritisation of the Broker and MQTT Client containers, when there is a new container that generates a pressure on the CPU of the Docker's behaviour, causing stress in the system. To this end, scenarios 1 (section 5.2), 2 (section 5.3), 4 (section 5.5) y 5 (section 5.6) have been compared.

By looking at the Figure 6.4, it can be seen how the MQTT protocol takes less than 9.5ms to process in 85% of the messages, in the scenario that there is no stress. In the scenario with stress, it can be seen how the MQTT protocol takes less than 19.5ms to process in 85% of the messages. On the other hand, in the scenario with stress but with the MQTT containers hosted in Docker prioritised, the MQTT protocol takes less than 14.5ms in 85% of the messages. In other words, when MQTT containers have prioritisation, the inclination in the graphic is more similar to when there is no stress action, meaning that the stress affectation is reduced and a faster protocol latency time is achieved.

On the other hand, it is also observed that prioritising the two MQTT containers at the same time, shows little difference to prioritising only one of the two containers. In other words, it means that prioritising both containers does not provide any more benefit than

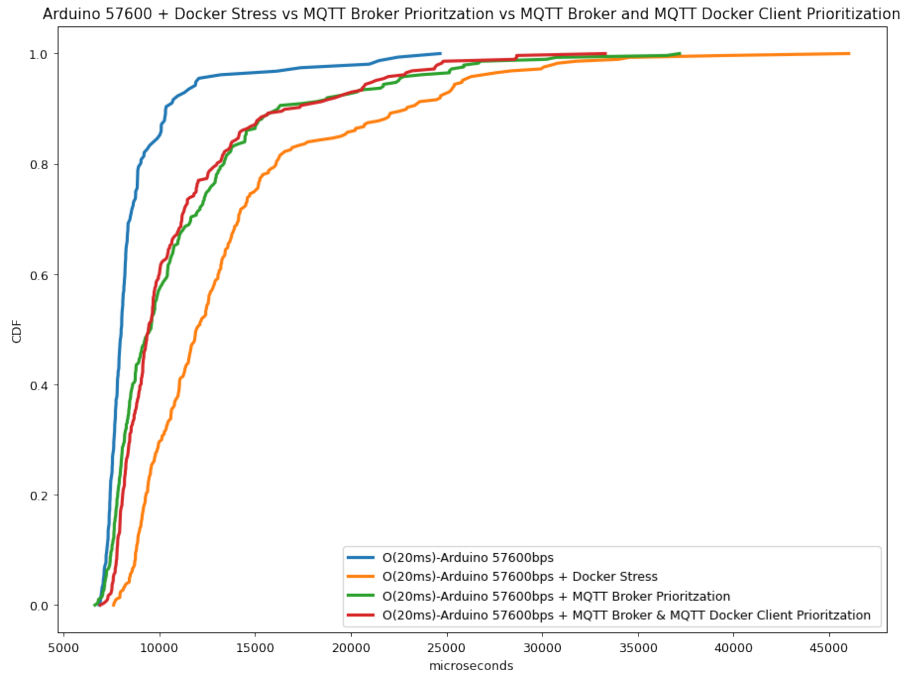


Figure 6.4: CDF graph comparing latencies of Scenarios 2, 4 and 5

prioritising only the Broker's container. Therefore, prioritising only the Broker's container is enough to improve the system's performance.

6.5 No Arduino + Docker Stress vs No Arduino + MQTT Broker Prioritisation

This analysis aims to compare scenario 3 (section 5.4) with scenario 6 (section 5.7) in order to analyse the impact on the MQTT communication protocol of the prioritisation of the Broker's container when there is a new container that generates a pressure on the CPU from the Docker's behaviour, causing stress in the system. Also, there is no contribution from the Arduino 6.5 is obtained.

By looking at the Figure 6.5, it can be seen how the MQTT protocol takes less than 10ms

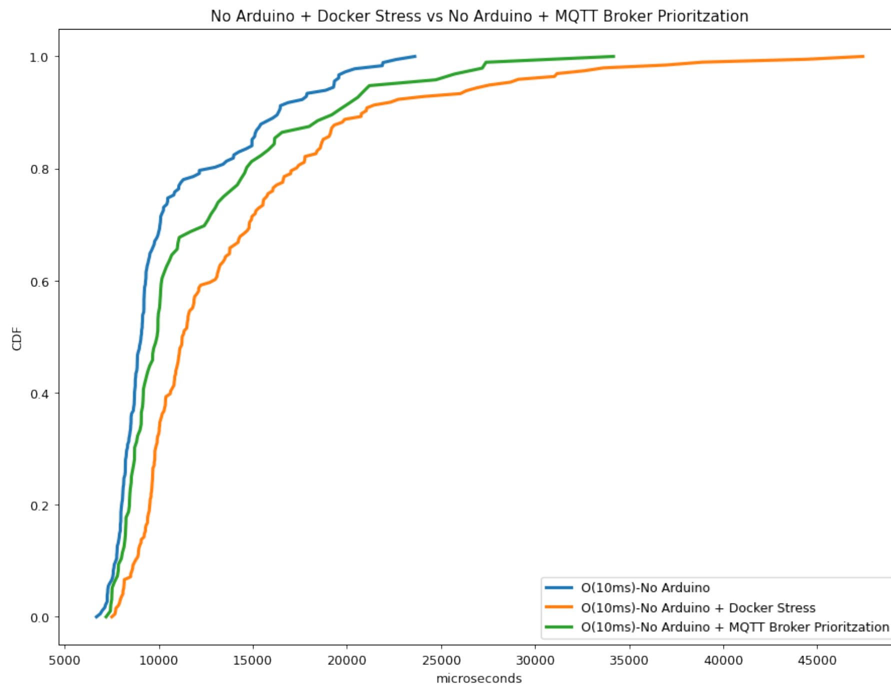


Figure 6.5: CDF graph comparing latencies of Scenarios 3 and 6

to process 70% of the messages, in the scenario that there is no stress. In the scenario with stress, it can be seen how the MQTT protocol takes less than 15ms to process in 70% of the messages. Moreover, on the other hand, in the scenario with stress but with the MQTT Broker's container hosted in a prioritised Docker, the MQTT protocol takes less than 12.5ms to process in 70% of the messages. In other words, when the MQTT Broker container has prioritisation, the inclination in the graph is more similar to when there is no stress action, meaning that the stress affectation is reduced and a faster protocol latency time is achieved.

In this analysis, the scenario with both MQTT containers prioritised has not been included since, as it has been observed in the Analysis 6.4, it does not show any benefit in the system behaviour.

6.6 Summary

To sum up, there have been five analyses of the system in this section, in which it has been possible to see that the MQTT protocol tends to act more quickly when the physical Arduino exists and does not emulate its ideal behaviour.

In addition, it has also been concluded that when there is a new container that generates a pressure on the CPU from the Docker's behaviour, causing stress in the system, it is possible to see how the MQTT protocol has higher latency times. However, in order to at least reduce the impact of stress on the system, it has been seen that the Broker's container must be prioritised and then a faster communication protocol latency time is achieved.

7 Conclusion and Future Work

Due to the importance of the Internet of Things and its impact on daily life, the different technologies applied to them are continually being investigated and how they can be most beneficial. It is becoming increasingly important to be able to consult/control electronic devices remotely.

In this thesis, a system was created to be able to control a robot through Docker remotely, and the different latency times in the different scenarios were evaluated.

In the initial scenario, the system gives reasonably good latency results, of which 35% represents the time taken by the robot to process the information. In other words, the communication protocol used is efficient and fast; however, the response latency of the scenario will depend on the device to which the request is executed (in this case, reference is made to the Arduino contained in the robot).

Besides, scenarios have been created by pushing the system to the limit, causing stress in Docker and also directly to the MQTT Broker. Nevertheless, finally, it has been concluded that to try to avoid this phenomenon, it is necessary to prioritise, at least, the Docker container of the MQTT Client. Also, depending on the importance of the data, the use of QoS, taking into account that sets the highest level, can cause a significant increase in the system's latency.

Finally, the utility of the system created has been very simple and, practically, only for the use of analysis. That is why, as future work, it would be interesting to see how the system acts if the robot subscribes to different topics and also, depending on the answers, acts with specific balancing movements or changes its speed.

References

[1] *Entendiendo Docker.* https://www.javiergarzas.com/2015/07/entendiendo-docker.html, Accessed: 2020-06-15.

[2] *What is Docker?* https://opensource.com/resources/what-docker, Accessed: 2020-06-15.

[3] *IT explained: mqtt.* https://www.paessler.com/it-explained/mqtt, Accessed: 2020-06-16.

[4] *Docker Official Images - eclipse-mosquitto.* https://hub.docker.com/$_/eclipse-mosquitto, Accessed: 2020-06-29$.

[5] *Docker Official Images - mqtt-client.* https://hub.docker.com/r/efrecon/mqtt-client, Accessed: 2020-06-29.

[6] *Eclipse Mosquitto^{TM}.* https://mosquitto.org/, Accessed: 2020-06-29.

[7] *Raspberry Pi 4 Tech Specs.* https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/, Accessed: 2020-06-29.

[8] *Runtime options with Memory, CPUs, and GPUs.* https://docs.docker.com/config/containers/resource$_constraints/, Accessed : 2020-06-29$.

[9] *MQTT man page.* https://mosquitto.org/man/mqtt-7.html, Accessed: 2020-06-30.

[10] *Qué son y cómo usar los Topics en MQTT correctamente.* https://www.luisllamas.es/que-son-y-como-usar-los-topics-en-mqtt-correctamente/, Accessed: 2020-06-30.

[11] *Understanding MQTT QOS Levels- Part 2.* http://www.steves-internet-guide.com/understanding-mqtt-qos-2, Accessed: 2020-06-30.

[12] *¿Qué es MQTT? Su importancia como protocolo IoT.* https://www.luisllamas.es/que-es-mqtt-su-importancia-como-protocolo-iot/, Accessed: 2020-06-30.

[13] Manuel Peuster Stefan Schneider Panagiotis Gouvas Daniel Behnke Marcel Müller Patrick-Benjamin Bök Anastasios Zafeiropoulos, Eleni Fotopoulou. *A scalable and low-cost MQTT broker clustering system.* IEEE, Published in: 2017 2nd International Conference on Information Technology (INCIT), 2018.

[14] Kevin Ashton. *The Internet Of Things.* https://medium.com/how-to-fly-a-horse/the-internet-of-things-e3050dd55556, Accessed: 2020-06-30.

[15] Lindsay Hiebert. *Public Safety Blog Series-Connecting the Unconnected in Public Safety Response.* https://blogs.cisco.com/government/connecting-the-unconnected-in-public-safety-response, Accessed: 2020-06-30.

[16] B. Ritesh Kumarn. *What are Internet of things (IoT) devices? A complete guide.* https://acquirehowto.com/what-are-internet-of-things-devices-complete-guide/, Accessed: 2020-06-30.

[17] Vasaka Visoottiviseth Ryousei Takano Jason Haga Dylan Kobayashi Pongnapat Jutadhamakorn, Tinnapat Pillavas. *A scalable and low-cost MQTT broker clustering system.* IEEE, Published in: 2017 2nd International Conference on Information Technology (INCIT), 2018.

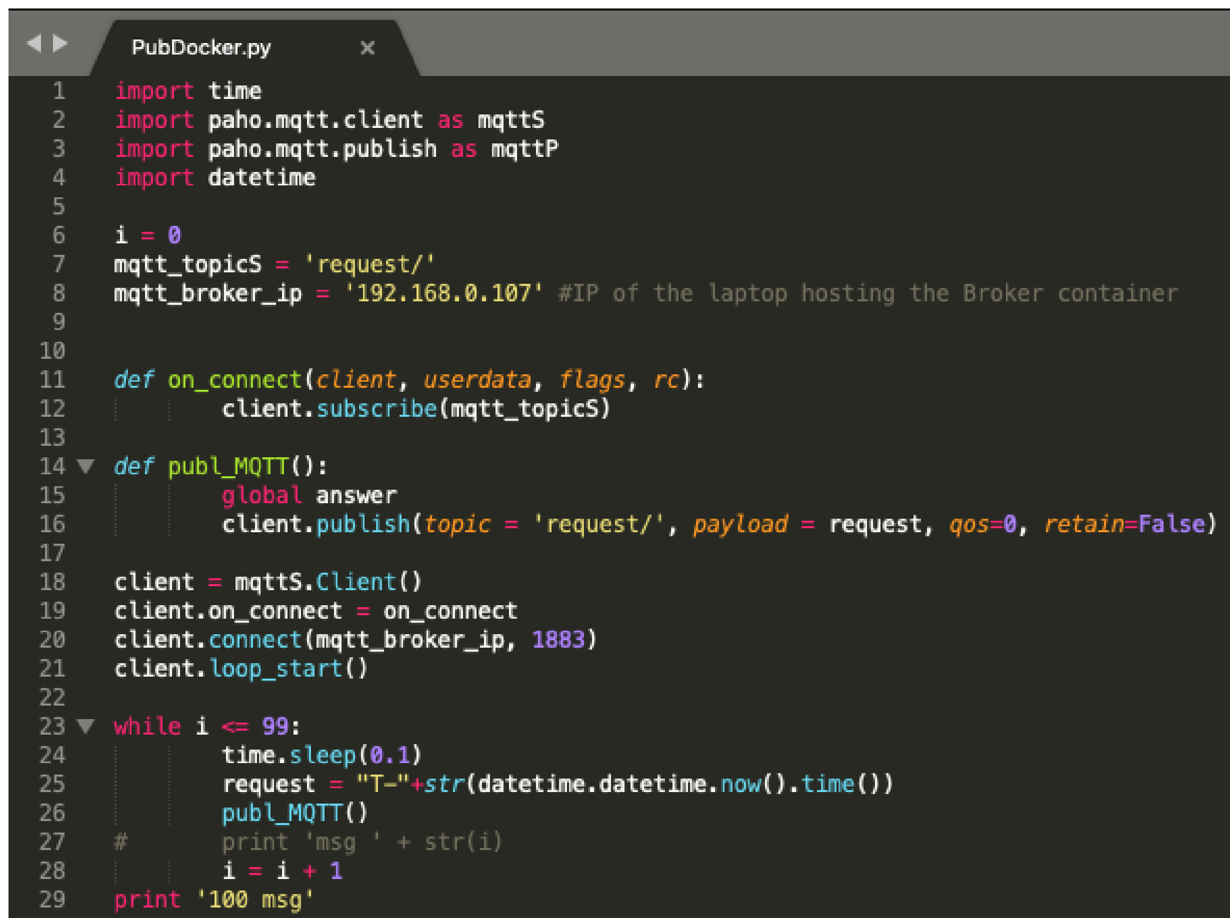
[18] G. Salgueiro D. Hanes R. Barton. *IoT Fundamentals: Networking Technologies, Protocols, and Use Cases for the Internet of Things.* Cisco Press, 2017.

[19] Margaret Rouse. *MQTT (MQ Telemetry Transport)*. https://internetofthingsagenda.techtarget.com/definition/MQTT-MQ-Telemetry-Transport, Accessed: 2020-06-16.

[20] The HiveMQ Team. *Quality of Service 0, 1 and 2 - MQTT Essentials: Part 6*. https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/, Accessed: 2020-06-30.

A Appendix - Scripts

In this appendix are attached all the scripts used in the project.



PubDocker.py

```
import time
import paho.mqtt.client as mqttS
import paho.mqtt.publish as mqttP
import datetime

i = 0
mqtt_topicS = 'request/'
mqtt_broker_ip = '192.168.0.107' #IP of the laptop hosting the Broker container


def on_connect(client, userdata, flags, rc):
        client.subscribe(mqtt_topicS)

def publ_MQTT():
        global answer
        client.publish(topic = 'request/', payload = request, qos=0, retain=False)

client = mqttS.Client()
client.on_connect = on_connect
client.connect(mqtt_broker_ip, 1883)
client.loop_start()

while i <= 99:
        time.sleep(0.1)
        request = "T-"+str(datetime.datetime.now().time())
        publ_MQTT()
#       print 'msg ' + str(i)
        i = i + 1
print '100 msg'
```

Figure A.1: Docker MQTT Client Publishing Script

```
PubDocker_QoS2.py

import time
import paho.mqtt.client as mqttS
import paho.mqtt.publish as mqttP
import datetime

i = 0
mqtt_topicS = 'request/'
mqtt_broker_ip = '192.168.0.107'


def on_connect(client, userdata, flags, rc):
        client.subscribe(mqtt_topicS)

def publ_MQTT():
        global answer
        client.publish(topic = 'request/', payload = request, qos=2, retain=False)

client = mqttS.Client()
client.on_connect = on_connect
client.connect(mqtt_broker_ip, 1883)
client.loop_start()

while i <= 99:
        time.sleep(0.02)
        request = "T-"+str(datetime.datetime.now().time())
        publ_MQTT()
        i = i + 1
print '100 msg'
```

Figure A.2: Docker MQTT Client Subscribing Script with QoS

SubDocker.py

```python
import time
import paho.mqtt.client as mqttS
import paho.mqtt.publish as mqttP
import datetime
import array as arr

mqtt_topicS = 'answer/'
mqtt_broker_ip = '192.168.0.107' #IP of the laptop hosting the Broker container
T = []
R = []
T1 = []
T2 = []
i = 0
a = 0

# CONNECTION MQTT
def on_connect(client, userdata, flags, rc):
     print "Connected!", str(rc)
     client.subscribe(mqtt_topicS)

# SUBSCRIBE MSG
def on_message(client, userdata, msg):
    global T
    global R
    global T1
    global T2
    global i
    T2.append(str(datetime.datetime.now().time()))
    msg1 = str(msg.payload) #recives T1,R
    answer = msg1.partition(",")
    T1.append(answer[0])
    R.append(int(answer[2]))
    i=i+1


# MQTT
client = mqttS.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect(mqtt_broker_ip, 1883)
client.loop_start()

while True:
        if ( i == 100):
        print len(R)
        print len(T2)
        while (a<len(R)):
        datetimeFormat = '%H:%M:%S.%f'
            diff = datetime.datetime.strptime(T2[a], datetimeFormat) - datetime.datetime.strptime(T1[a], datetimeFormat)
            T.append(diff.microseconds)
            a=a+1
        with open("R.txt", "w") as output:
            output.write(str(R))
        with open("T.txt", "w") as output:
            output.write(str(T))
        i=i+1
```

Figure A.3: Docker MQTT Client Subscribing Script

MQTTrpi-arrays (1).py

```python
import serial
import time
import paho.mqtt.client as mqttS
import paho.mqtt.publish as mqttP
import datetime
import array as arr

arduino = serial.Serial('/dev/ttyUSB0', baudrate=57600)
request = ''
answer = ''
ardu = ''
mqtt_topicS = 'request/'
mqtt_broker_ip = '192.168.0.107' #IP of the laptop hosting the Broker container
T1 = []
R1 = []
R2 = ''
R = ''
ID = 0

# CONNECTION MQTT
def on_connect(client, userdata, flags, rc):
#       print "Connected!", str(rc)
        client.subscribe(mqtt_topicS)

# SUBSCRIBE MSG
def on_message(client, userdata, msg):
        global T1
        global R1
        global ardu
        msg1 = str(msg.payload) #recibe T,ID-hh:mm:ss de T1
        request = msg1.partition("-")
        T1.append(request[2])
        R1.append(str(datetime.datetime.now().time()))
        ardu = request[0]
        arduino.write(ardu)

# PUBLISH MSG
def publ_MQTT():
        global answer
        client.publish(topic = 'answer/', payload = answer, qos=0, retain=False)

# MQTT
client = mqttS.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect(mqtt_broker_ip, 1883)
client.loop_start()
while True:
        while arduino.inWaiting() > 0:
                answer += arduino.read(1)
        if answer != "":
                R2 = str(datetime.datetime.now().time())
                datetimeFormat = '%H:%M:%S.%f'
                diff = datetime.datetime.strptime(R2, datetimeFormat) - datetime.datetime.strptime(R1[int(ID)], datetimeFormat)
                R = diff.microseconds
                answer = T1[int(ID)] + "," + str(R)
                ID = ID +1
                publ_MQTT()

        answer = ''

arduino.close()
```

Figure A.4: RPi MQTT Client Subscribing-Publishing Script