SILESIAN UNIVERSITY OF TECHNOLOGY

FACULTY OF AUTOMATIC CONTROL, ELECTRONICS AND COMPUTER SCIENCE

PROGRAMME: INFORMATICS

Master Thesis

Edge Class MQTT Broker Pipeline Performance Evaluation in IoT Network

Author: Brijesh Jagdishbhai Varsani

Supervisor: dr inż. Piotr Czekalski

Gliwice, November 2021

Index

Abstract

The Internet of Things (IoT) is developing at an incredible speed. For IoT applications, the Message Queuing Telemetry Transport (MQTT) protocol is a common communication protocol. Given the importance of Internet Protocol performance in IoT devices, this thesis intends to describe several elements that can affect the performance of the MQTT protocol as well as the various trade-offs that exist when using the MQTT protocol. The most common open-source MQTT implementation; the Mosquitto server is used to evaluate the performance of protocol on a private network so the results will not be disturbed by the other process. To verify system latency, pipeline performance is evaluated using MQTT parameters such as payload size, number of publishers/subscribers, and wildcards. The results indicate that changes or increments in the parameters resulted in the changing or increasing latency in MQTT while transmitting the message. The changes in latency often follow the linear and/or a logarithmic trendline.

Keywords: IoT, MQTT Protocol, MQTT Broker, ESP8266, Wildcards, Payload, Topics, Performance Evaluation.

1. Introduction

The Internet has evolved into the foundation for a wide variety of different and unique applications. As a consequence, IoT applications have reached every corner of the planet, beginning a new age of technological change and innovation. This modern revolution seeks to create complete communication between objects straight from any human contact. As a result, IoT may be defined as the interaction or communication of a set of components or devices that are equipped with embedded sensors or processors and can communicate remotely using an appropriate IoT communication protocol [1]. MQTT (Message-Queuing Telemetry Transport), HTTP (Hypertext Transfer Protocol), AMQP (Advanced Message-Queuing Protocol), and CoAP (Constrained Application Protocol) are the few to name in the Machine to Machine (M2M) Communication protocol segment [2]. Among these IoT Protocols, MQTT is free, simple to deploy, lightweight, and energy-efficient. It is a publish-subscribe model message transport protocol and is used in over 40% of existing IoT devices [2]. Despite the chip scarcity and the long-term effect of COVID-19 on the supply chain, the Internet of Things industry is expanding [3]. According to IoT Analytics, the worldwide number of connected IoT devices will increase by 9% to 12.3 billion active endpoints in 2021. More than 27 billion IoT connections are expected by 2025 and IoT devices are usually resource-constrained [4].

The MQTT protocol offers a wide range of applications, including healthcare, logistics, smart city services, and others [1]. Each application area needs special system requirements, more compatibility, scalability, and performance of individual devices, as well as the system as a whole, necessitates a unique approach or guideline to implement their system. This thesis seeks to uncover and elaborate on how the various parameters of the MQTT protocol affect latency and overall MQTT pipeline performance with regards to its payload length, numbers of subscribers\publisher to broker, and wildcard subscription (chapter 1.2.5). The output of it should be answered to the question like in the case of sensor network: Where there are multiple sensors, ability to keep reasonable transfer times and avoid dropping of messages? What level of delays\latency is to be expected regarding traffic in-network? What kind of growth or impact can be accepted with changes to MQTT parameters? Those answers may be very crucial in real-time IoT systems where delays need to be constant and very low.

This chapter describes the scope of the thesis as well as some definitions for terms that have been used. The remaining structure is as follows:

Chapter 2 would be the related work on the research area, research methods, and the limitations of the design of the experiment, in which it has thoroughly described the main area of the research part and important findings on the thesis.

Chapter 3 describes overall the experimental architecture. Starting from the setup of the experiments and the components that have been used during the research and the brief description of the experimental area that the research is focused on.

Chapter 4 would give the analysis and comparison of the results and possibly some discussion about the performance of the MQTT that are observed during and after the analysis.

Chapter 5 concludes this thesis with finding from the research experiments, the limitation of the results, and scope for the future research.

1.1. Scope of The Thesis

The main scope of the thesis is to perform experiments for evaluation MQTT broker performance. The scope details will be presented below.

The Pipeline performance experiments evaluate the MQTT broker which is regarding the various parameters of the MQTT communication protocol that includes Payload length, Number of concurrent subscribers, and Wildcard complexity in subscriptions.

Overall, the results would give the idea and/or guidelines for implementing and scaling IoT applications that use for MQTT communication.

1.2. Definitions

The definitions of some of the most important areas and frequent terminology used in this thesis are provided in this introductory subsection.

1.2.1. MQTT Protocol

MQTT is an Internet of Things (IoT) networking protocol that was created as a very lightweight publish and subscribe messaging transport. It is particularly beneficial in applications that demand a connection with a device in a faraway place. The protocols themselves must have low overhead and consume restricted bandwidth in these circumstances, consuming little fewer resources than other protocols.

MQTT protocol is a (publish/subscribe) messaging transport protocol for clients and servers. MQTT is a lightweight, open, straightforward, and easy-to-use messaging protocol. The protocol may be used with any protocol that supports ordered, lossless, bidirectional connections (most often TCP/IP) and supports ports 1883 by default and 8883 for non-encrypted and encrypted communication, respectively [5]. It has multiple Quality of Service(QoS) levels for different use cases, is data agnostic, and has a publish-subscribe architecture that allows applications to be decoupled and messages to be multicast. The low transmission overhead that provides for effective communication between devices, is its most significant characteristic.

Because of its potential to be fast and light-weight data transport, MQTT has been called the Internet of Things protocol. It becomes a standard for Internet of Things communications, which is standardized by OASIS and ISO organizations, offers a scalable and reliable means to link IoT devices over the Internet. However, because it has several configurable parameters, its performance can vary greatly. The goal of this thesis is to determine which parameters are significant in evaluating how much the configuration can impact performance in typical applications. This thesis examines many variable parameters that might influence MQTT's performance.

1.2.2. MQTT Broker

In MQTT, the clients do not communicate directly with one another, but through an intermediary, the MQTT broker. The broker provides a server to which clients connect to publish data and subscribe data. Its job is to take messages from publishing clients and transport them to the relevant subscribers, and eventually retain them if needed.

There are several MQTT brokers available openly for testing and real-world applications. There are several free self-hosted brokers available, the most popular of which is Mosquitto [6]. It is a free and open-source MQTT broker for both Windows and Linux operating systems. After the installation of a broker, it can be started automatically on system startup or with a manual approach. The broker works as an intermediate for both clients so, it can store messages only temporarily, once the messages have been sent to all subscribers they are then discarded unless marked to be retained.

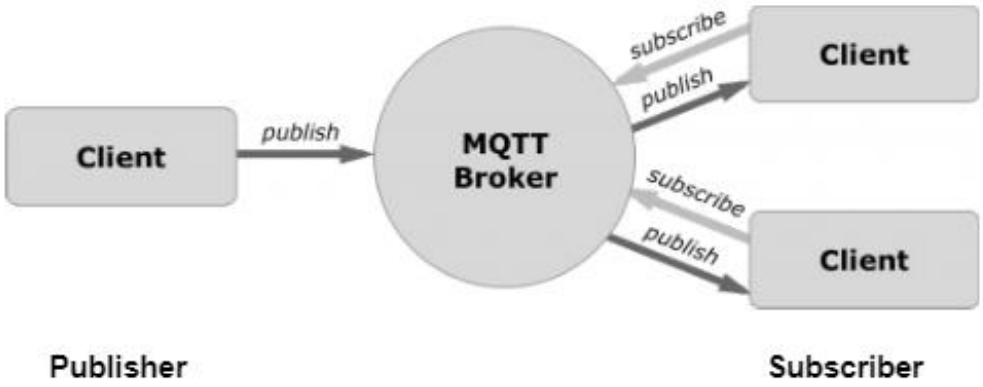

Figure 1: *MQTT Overview*

Figure 1 shows the MQTT overview of a client-server publish-subscribe-based messaging protocol work in general.

The maximum message size limit in MQTT is 268,435,456 bytes. But one can restrict the maximum messages size the broker will accept can be changed in the Mosquitto configuration file. The broker also discards messages received more than the limit.

1.2.3. MQTT Client

In MQTT, the clients include both publishers and subscribers. The publisher and subscriber labels indicate whether the client is currently publishing messages or is subscribed to receive messages. Both publishing and subscribing capabilities may be enabled by the same MQTT client. Any device from a microcontroller to a full-fledged server that runs the MQTT library and connects to the MQTT broker via a network is termed the MQTT client. The MQTT client, for example, may be a very little, resource-constrained device that connects over a wireless network and has a bare-bones library. For testing purposes, the MQTT client could instead be a standard Notebook/PC running a graphical MQTT client. MQTT client is any device that interacts with MQTT using a TCP/IP stack. The client implementation of the MQTT protocol is easy. One of the reasons why MQTT is best suited for tiny devices is its simplicity of deployment. Moreover, the MQTT client libraries are available for a broad variety of programming languages.

1.2.4. Payload

The MQTT protocol operates by sending and receiving a set of MQTT Control Packets in a certain order. The MQTT Control Packet is made up of three parts: a fixed header (which is present in all packets), a variable header (which is optional), and the payload (optional).

As shown in Figure 2, the Fixed header contains the Packet Type and Flags as a control header (1 byte) and Packet length which consists of 1 to 4 bytes. The Packet type is represented as a 4-bit unsigned value.

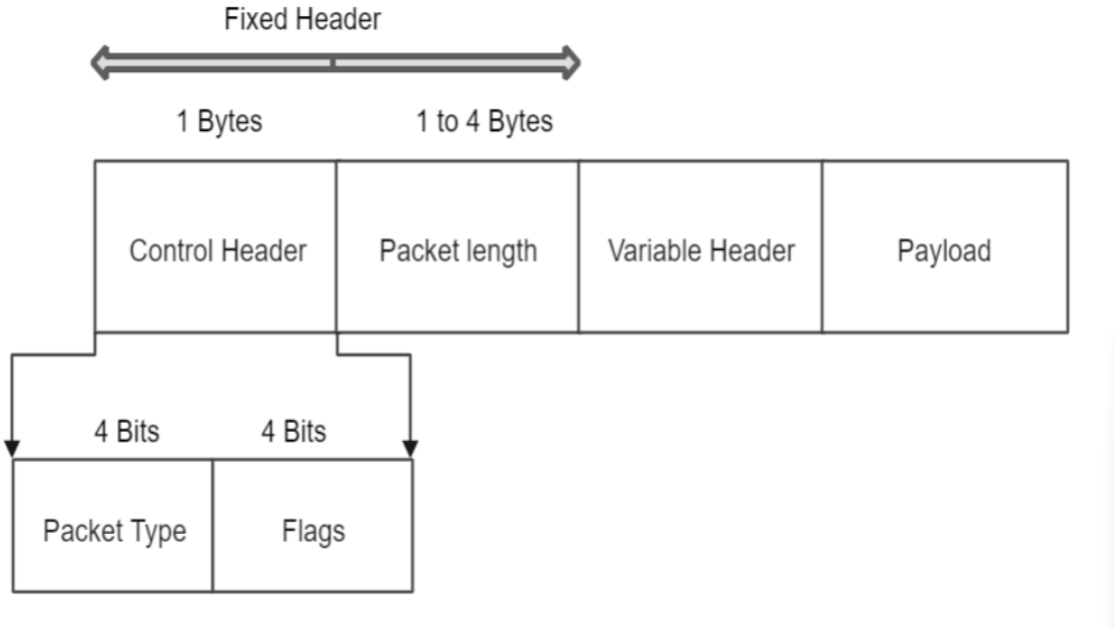

Figure 2: *A Standard Packet Structure of MQTT Protocol*

A variable header component is included in some MQTT Control Packets. It's in the middle of the packet, between the fixed header and the payload. The variable header's content varies based on the packet type. Several packet types use the Packet Identifier field of the variable header.

Application Message that is being published is contained in the Payload. MQTT Control Packets contain a Payload as the final part of the packet. The data's content and format are application-specific. The length of the payload may be determined by subtracting the variable header's length from the Fixed Header's Remaining Length field. And it is acceptable for a PUBLISH Packet to include a payload of zero length.

The presence of one or more length-prefixed fields in the CONNECT Packet's payload is determined by the flags in the variable header. If present, these fields MUST exist in the correct order: Client Identifier, Will Topic, Will Message, User Name, Password.

1.2.5. Wildcards

When a client subscribes to a topic, it can either subscribe to the specific subject of a published message or utilize wildcards to subscribe to many topics at the same time. A wildcard may only be used to subscribe to topics; it cannot be used to post a message. Wildcards are classified into two types: Single-level and Multi-level.

Example of normal topic:

(1) home/floor1/livingroom/temprature

(2) home/floor2/kitchen/humidity

- Single-Level: '+'

A single-level wildcard, as the name implies, replaces one topic level. In a subject, the '+' symbol represents a single-level wildcard. If a subject has an arbitrary string instead of a wildcard, it matches a topic with a single-level wildcard.

Examples for single-level topic:

(1) home/floor1/+/temprature

(2) home/+/livingroom/temprature

- Multi-Level: '#'

The multi-level wildcard covers a wide range of topic levels. The hash symbol '#' denotes the topic's multi-level wildcard. The multi-level wildcard must be placed as the last character on the topic and preceded by a forward slash '/' (topic separator) for the broker to determine whether topics match [5].

Examples for single-level topic:

(1) home/floor2/#

(2) home/#

- Topics start with '$'

The subject name can be anything. However, there is one exception: topics beginning with a ‘$’ symbol serve a different purpose. When subscribing to a multi-level wildcard as a subject ‘#’, these topics are not included in your subscription. The ‘$’ symbol topics are reserved for MQTT broker internal statistics logs. Clients are unable to post messages to these subjects. There is currently no recognized standardization for such topics. ‘$SYS/’ is commonly used for all of the following information, but broker implementations vary.

Examples: (1) $SYS/broker/clients/connected,

(3) $SYS/broker/clients/disconnected

1.2.6. Latency

Latency is a time-based statistic for assessing a system's performance. Wherever feasible, a good message provider should optimize and decrease delay. For a published-subscribed service, the latency metric may be defined as the time it takes the service to acknowledge a sent message or the time it takes the service to transmit a published message to its subscriber. Latency is also defined as the amount of time it takes a messaging service to deliver a message from the publisher to the subscriber [6].

2. Related Work

This chapter explored the novelty in the domain of MQTT performance assessment. There will be a discussion about the newest literature review in the area of MQTT. In the research domain, the current edge class advancement in protocol performance benchmarking is described.

2.1. Literature Review

When developing an IoT system, one of the most important aspects to consider is performance. It is quite important when it comes to the scalability and reliability of IoT applications because that it is necessary to have prior knowledge of latency and its configurations. Because the thesis's scope is focused on the MQTT protocol, this chapter reviewed some of the most significant MQTT performance-related research published in the literature in the recent 5 years.

The MQTT-related latency experiments in [7] revealed that QoS level 0 had the lowest latency of all the QoS levels. However, the data also revealed that QoS 1 and 2 get almost comparable latency. Furthermore, the use of environmental experiments revealed appropriate findings. The statistics revealed that the latency effect varied very little across the different use conditions. Furthermore, the statistics indicate that CoAP and MQTT had shorter latencies in a high Wi-Fi signal strength environment than in a low Wi-Fi signal strength environment.

According to the research paper [8], during message bursts, the delivery follows a LIFO (last-input-first-output) sequence, resulting in messages being digested in reverse order. However, this is not done using the MQTT protocol, because delivery is always in order. Message losses occur only when the load on the producer side exceeds the system buffer capacity. This discovery gives the knowledge required to carry out the thesis experiment.

In the research paper [9], Mosquitto [10] takes the least amount of time to transmit messages between publisher to subscribers which are followed by HiveMQ [11] and finally

the MQTT Server [12]. At the QoS 1 level, the MQTT Server performed the best for transmitting messages, the Mosquitto came in second, and HiveMQ came in third. Mosquitto topped the list once again at the QoS 2 level. Furthermore, it is assumed that MQTT offers the benefit of utilizing fewer data to transmit the same payload, as well as having somewhat shorter transmission times. The most significant difference was in the transmission of a message to numerous clients, where the MQTT protocol provided a significant benefit in transmitting messages to a large number of subscribers.

The research paper [13] offered an in-depth examination of the four frequently used messaging protocols for IoT systems: MQTT, CoAP, AMQP, and HTTP. This paper compares the properties of various protocols in general. Furthermore, by performing an in-depth and relative analysis based on various associated criteria, this study sheds light on the strengths and limits of different IoT protocols.

The author of the [14] concludes that the semantic data extraction is facilitated by several of the MQTT protocol's characteristics, such as message retention, which allows for the simple addition of additional devices. The availability of topics for subscription and publication makes data routing processes essentially redundant and eliminates the need for heavy mechanisms at the program level for directing data to certain buffers. QoS and last will improve MQTT's dependability and make it appropriate for use in constrained environments.

To analyze MQTT, it is also necessary to investigate the security and encryption of the payload in the system. This research paper [15] demonstrates a lightweight MQTT design paradigm that eliminates data transmission encryption. Encryption applied as a distinct feature through TLS by various MQTT brokers raises overhead. From a security standpoint, MQTT's default plain-text data exchange presents a significant risk.

2.2. Research Areas

This section will consist of multiple areas of different parameters and their workings, as well as how those parameters affect the MQTT performance.

2.2.1. Current Advance in Protocol Performance Benchmark

Over the last decade, there has been a substantial increase in the global use of IoT. As the number of linked devices grows by the billions year after year, the capacity and operational costs of IoT networks and related communications software become critical. Manufacturers, software developers, integrators, telecom operators, and business-end customers are all in desperate need of a benchmarking reference that encompasses the performance elements of IoT transport protocols.

Chapter (2.1) literature reviews demonstrate the most recent advancement in evaluating the performance of IoT transport protocols, which compare the performance of various IoT protocols such as CoAP, AMQP, and HPPT [13], MQTT power consumption [7], the performance impact analysis of securing the MQTT [16], Performance authentication in MQTT [17].

Furthermore, article [17] presents a performance benchmarking methodology as well as examples for the creation of MQTT protocol performance testing. The implementation work was completed under the Eclipse Foundation's open-source project IoT Testware.

There are some researches on accurate benchmarking of the MQTT protocol on the basis of its various configuration parameters, and it is required to comprehend the whole pipeline performance of the MQTT and generalize the conclusions acquired from the experiments.

As a state-of-the-art solution, this thesis proposes a solution for evaluating MQTT performance on the basis of various configurations, As a result, the objective is to utilize these methods to measure how effectively the MQTT broker is performing its functions. Because MQTT is a transport protocol, the measurements will concentrate on how quick, dependable, and efficient data transmission is handled. The measurements are intended to

serve this function while also addressing a wide range of use-case situations. For the acquired measurement data to be usable, the test system configuration must be carefully considered.

At the present, it is quite difficult to obtain clear instructions for anyone interested in designing and implementing an IoT ecosystem on how MQTT protocol settings affect latency. This understanding is critical in the case of many IoT applications.

In this thesis, research was performed on MQTT pipeline performance, and many experiments were carried out to elaborate on benchmarking and assessing MQTT performance on different configuration parameters.

2.2.2. MQTT Connection

The TCP/IP is the foundation for the MQTT protocol. A TCP/IP stack is required on both the client and the broker. Figure 3 describes the protocol stacks in the IoT network.

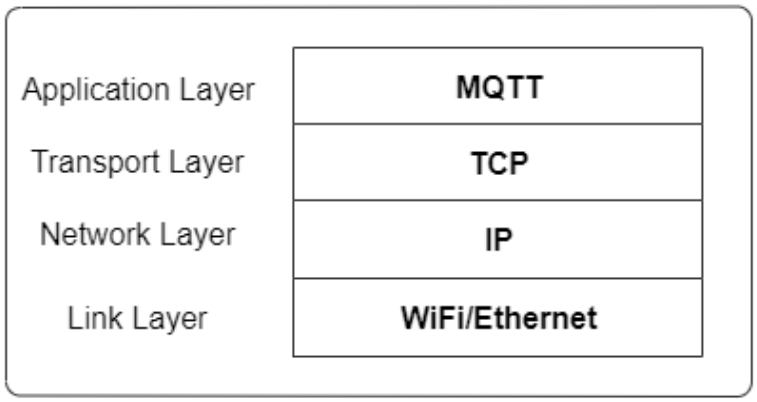

Figure 3: *Protocol Stacks*

The MQTT connection is always established between a single client and the broker, which results in clients never communicate directly with one another. As shown in Figure 4, The client sends a CONNECT message to the broker to establish a connection. The broker replies with a CONNACK message and an indication of the status code. Once the connection is established, the broker maintains it until the client sends a disconnect signal or the connection is broken.

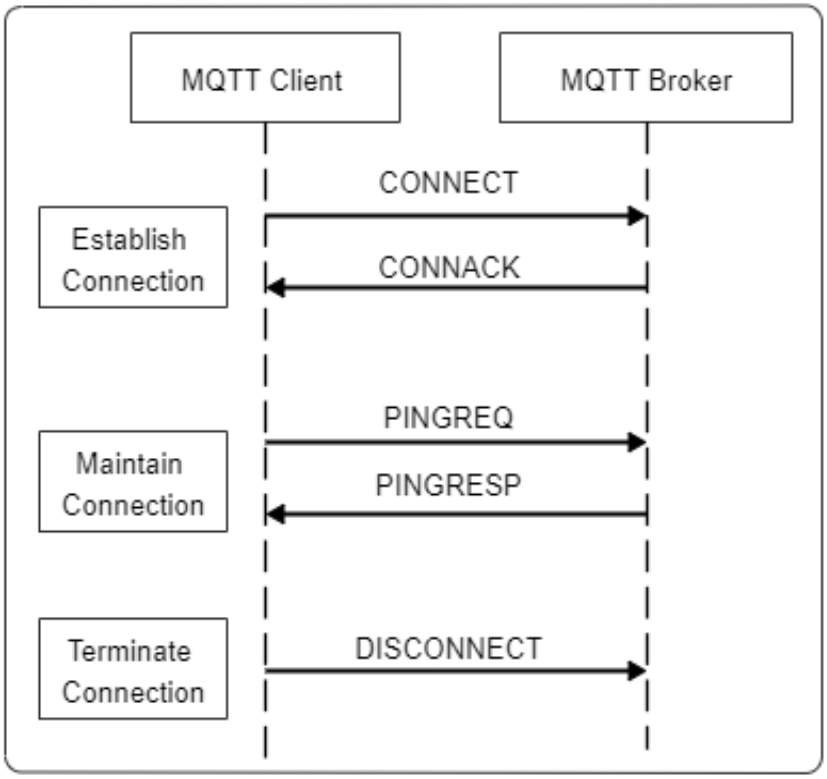

Figure 4: *MQTT connection flow for Establish, Maintain, and Terminate.*

The connection between the client and the server is ended after a set amount of time in MQTT. To sustain the connection, the client sends a PINGREQ packet to the server to signal that it is alive. The MQTT server sends a PINGRESP packet to the client with the specified identification in order to keep the connection alive till the disconnection.

To disconnect from the server, the MQTT client sends a DISCONNECT packet. This packet is not acknowledged by the server. However, all client-related application messages are flushed and the client is disconnected from the server.

2.2.2.1. MQTT connection through a Network Address Translation (NAT)

In many circumstances, the MQTT client is placed behind a router that employs network address translation (NAT) to convert a private network address (such as 192.168.X.X or 10.0.X.X) to a public-facing address. As previously stated, the MQTT client establishes the

connection by sending a CONNECT message to the broker. There is no difficulty with clients situated behind a NAT since the broker has a public address and maintains the connection open to enable bidirectional sending and receiving of messages which are after the very first CONNECT.

2.2.2.2. The client initiates the connection with a CONNECT message

The client sends a command message to the broker to begin a connection. The broker cancels the connection if this CONNECT message is faulty [5] or if too much time occurs between creating a network socket and sending the connect message. This approach discourages malicious customers from slowing down the broker. Some information in a CONNECT message is presumably more valuable to MQTT library implementers than to MQTT library consumers.

2.2.2.3. ClientID

Each MQTT client that connects to the MQTT broker is identified by the client identifier (ClientId). The ClientId is used by the broker to identify the client and the client's current state. As a result, this Id should be unique for each client and the broker.

If one does not require the broker to hold a state, you may transmit an empty ClientId in MQTT. The absence of the ClientId results in a connection with no state. In this situation, the clean session flag must be set to true or the connection would be rejected by the broker.

2.2.2.4. Clean Session

The clean session flag indicates to the broker whether or not the client wants to create a persistent session. A persistent session means CleanSession set to “false”, preserves all subscriptions for the client as well as any missed messages for the client who subscribed with a QoS level 1 or 2. If the session is not persistent means CleanSession is “true”, the broker does not save anything for the client and deletes all data from any prior persistent session.

Moreover, according to IBM Documentation [18], the MQTT client and service keep track of session status information. The state information can be utilized to assure "at least once" and "exactly once" delivery, as well as "exactly once" publishing confirmation.

Subscriptions made by a MQTT client are likewise included in the session state. An MQTT client may be executed with or without retaining state information between sessions.

When using the MqttClient.connect method to connect a MQTT client application, the client identifies the connection by utilizing the client identification and the server address. The server determines if session information from a previous connection to the server has been stored. If a prior session still exists and cleanSession is set to "true," the previous session information is erased at both the client and the server. The previous session is continued if cleanSession is set to "false." A new session is established if no prior session exists.

2.2.2.5. Username/Password

For client authentication and approval, the MQTT may communicate a user name and password. If this information is not encrypted or hashed through implement or TLS, the password is delivered in plain text. One can strongly advise using user names and passwords in conjunction with the secure transmission.

Brokers such as the Mosquitto may authenticate clients using an SSL certificate, eliminating the requirement for a username and password.

2.2.2.6. Keep-Alive

When the connection is established, the client specifies and transmits to the broker a time interval in seconds. This interval specifies the maximum amount of time that the broker and client may go without sending a message. The client agrees to send PING Request messages to the broker regularly. The broker sends a PING answer, that strategy enables both client and broker to know if the other is still accessible.

Furthermore, Individual libraries also provide extra configuration choices. For example, how queued messages are stored in a particular implementation.

2.2.3. Published Messages

When a client publishes a message on a topic then the broker will distribute that message to any connected clients that have subscribed to that topic. Once the message has been sent to those clients it will be removed from the broker unless it is not marked for retained. If no clients have subscribed to the topic or they are not currently connected, then the message is removed from the broker.

To conclude, the Broker does not store messages but retained messages, Persistent connections, and QoS levels can result in messages being stored temporarily on the broker or server.

2.2.4. Publishing Messages and The Retain Flag

Generally, if a publisher publishes a message to a topic, and nothing is subscribed to that topic the message is simply rejected or discarded by the broker. However, the publisher may request the broker to preserve the last message on that subject by setting the retained message signal. This might be highly important, for example, if the sensor publishes its status only when modified. Without retained messages, the subscriber would have to wait for the status to change before it received a message. However, with the retained message, the subscriber would see the current condition of the sensor.

What is important to understand is that just one message is retained per topic. The next message published on that topic replaces the previous retained message for that topic.

The retained message feature is a particular advantage for retaining the last state of an object and is particularly beneficial when the state doesn’t change regularly and Quality of service (QoS) parameters do not affect retained messages.

2.2.5. Subscribing to Topics

When one subscribes to a topic, it needs to set the quality of service for the topic subscription. The levels of QoS and their meaning are the same as for published messages. When subscribing to one or more topics, the broker needs to send messages on that topic. To send messages to a client, the broker uses the same posting mechanism that the client uses. You can subscribe to multiple topics using the two wildcards (+ and #), as shown in (chapter 1.2.5).

All subscriptions are confirmed by the broker using a subscription confirmation message that includes a packet identifier that can be used to verify the success of the subscription.

2.2.6. Multiple Subscribers for Same Topic

The Client is free to publish a message on any topic it chooses. Currently, there is no reserved Topic name. However, MQTT brokers can restrict access to topics.

The client cannot directly publish a message to another client and does not know whether the client receives that message. Clients can publish messages only to one topic and not to a group of topics. However, a message can be received by a group of subscribers if they subscribe to the same topic.

2.2.7. Shared Subscription by Load Balancing

The messages can be sent to an endless number of subscribers. However, sometimes simple PubSub is insufficient, and some type of client load balancing is required to accommodate some of the more complicated use cases involving several MQTT subscribers.

Shared Subscriptions are a technique that enables a subscription group's messages to be distributed to the group's members.

There is no limit to the number of Shared Subscriptions that may be purchased. When a publisher publishes a message with a relevant topic, the message is received by one (and only one) client from each group.

Shared Subscriptions are a non-standard MQTT feature that enables MQTT clients to share the same broker subscription. Each subscribing client gets a copy of the message when implementing standard MQTT subscriptions. When using Shared Subscriptions, all clients that share the very same subscription with a subscription group will get messages in an alternating pattern. Because the messaging burden of a single topic is dispersed across all subscribers, this technique is usually referred to as client load balancing.

MQTT clients may use standard MQTT protocols to subscribe to a shared subscription. This is particularly beneficial for backend systems or "hot-topics" that might rapidly overwhelm a single MQTT client.

2.2.8. Quality of Service (QoS)

The Quality of Service level is a consensus between a message sender and recipient regarding the assurances of receiving the message. These warranties may cover the number of times a message will arrive and whether duplicates will occur or not. The publishing process between the publisher and the MQTT broker has a QoS level, while the publishing process between the MQTT broker and the subscriber has a different QoS level. The following three different QoS levels are supported by MQTT.

2.2.8.1. QoS “0”

This QoS level provides the same guarantee as to the underlying TCP protocol. The receiver or the destination does not acknowledge the communication. Nothing happens once the publisher delivers the message to the receiver. If communication fails to reach its intended destination, the sender neither stores nor schedules new transmission of a message. When compared to the other QoS levels, this one has the lowest overhead.

2.2.8.2. QoS “1”

This QoS level adds a confirmation requirement to the destination that must receive the message at least once. In this approach, QoS level 1 guarantees that the message will be delivered to the subscriber at least once. One of the major drawbacks of this QoS level is that it may result in duplicate messages, meaning that the same message may be delivered to the same destination several times.

The publisher keeps the message until the subscriber responds with an acknowledgment. If the sender does not get an acknowledgment within a certain amount of time, the sender will resend the message to the receiver. The final receiver must be equipped with the appropriate logic to identify duplicates.

2.2.8.3. QoS “2”

Exactly once delivery: This QoS level ensures that the message is delivered to the intended recipient just once. In comparison to the other QoS levels, QoS level 2 has the largest overhead.

This degree of QoS necessitates two flows between the sender and recipient. When the publisher is certain that a message broadcast with QoS level 2 has been successfully received once by the destination, it is considered effectively delivered.

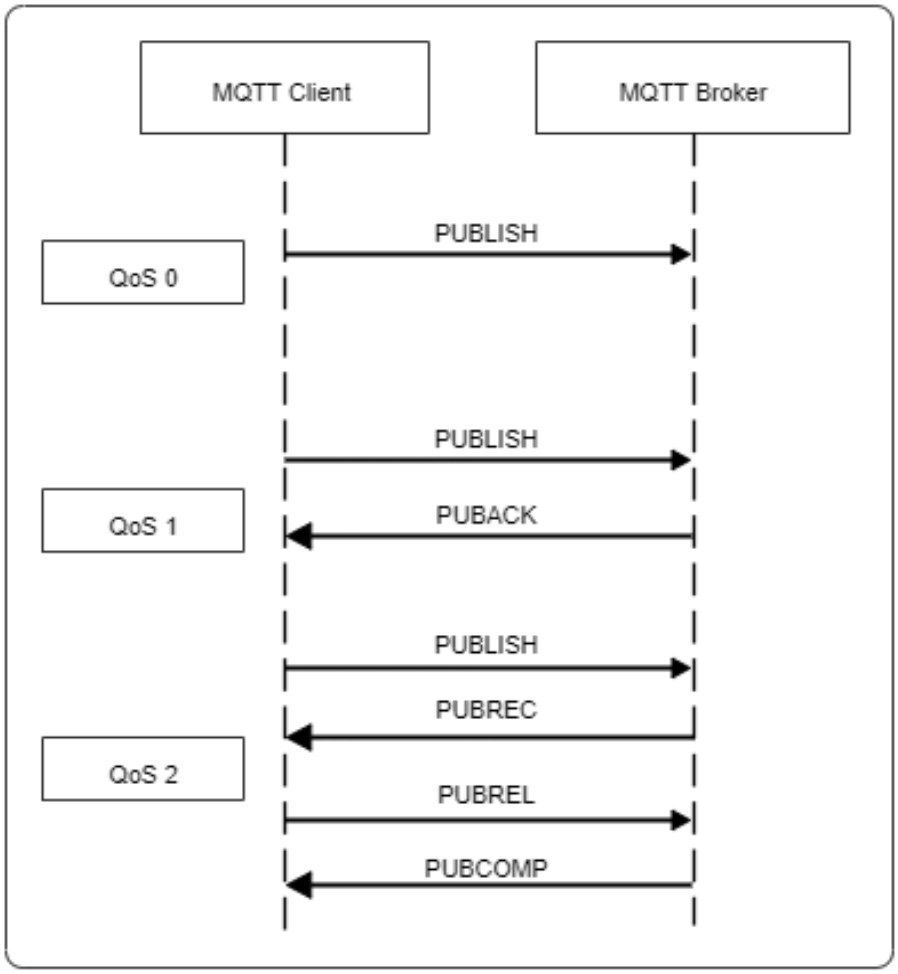

Figure 5: Publishing messages to the broker with different QoS

From Figure 5, if the application messages are sent at QoS 0, the client receives no acknowledgment for the published packet. In the case of QoS 1, the broker acknowledges the published packet with PUBACK, which includes the packet identification.

In QoS 2 however, four packets are exchanged. With the PUBREC packet, the server confirms receipt of the PUBLISH packet. The MQTT client then sends a PUBREL packet

to release publish. The server then delivers the fourth packet, PUBCOMP, indicating that the application message on the provided subject has been published.

If a publisher uses a higher QoS level than the subscriber specifies, the MQTT server must lower the QoS level to the lowest level the subscriber is using when publishing the message from the MQTT server to this subscriber. For example, If one utilizes QoS level 2 to publish a message from the publisher to the MQTT server, but one subscriber specified QoS level 1 when subscribing, the MQTT server will publish to this subscriber using QoS level 1.

Furthermore, for all Experiments environments, there is a use of the QoS level "0". The reason behind it is, first our research is based on parameters such as payload, topic name, and wildcards rather than QoS levels. Second, the system is connected to a highly reliable and secure private network. Finally, there are no issues with message duplication or loss because it get published when needed and the message is confirmed at the subscriber level.

2.2.9. Persistent Connections and Clean Sessions

When a client connects to a broker, it may use either a non-persistent (clean session) or a persistent connection. The broker does not store any subscription information or undelivered messages for the client while using a non-persistent connection.

When the client just transmits messages, this mode is suitable. It can also connect as a long-term client via a persistent connection. In this mode, the broker/server will hold messages for the client if it is disconnected depending on the QoS of the published messages and the subscribing client.

2.2.10. The Last Will

The last Will message is used to inform subscribers of an unexpected shutdown of the publication. The publisher directs the broker to notify all subscribers of a subject, using the last will message, in the case that the connection breaks. Second, if the broker detects a connection break it transmits the last will message to all subscribers of that subject.

2.2.11. MQTT Callback Function

The callback function work as it must be supplied in the constructor if the client is used to subscribing to topics. When new messages arrive at the client, this function is invoked. For both inbound and outbound messages, the client uses the same buffer internally.

The topic and payload values passed to the function will be overwritten once the callback function returns, or if a call to publish or subscribe is performed from within the callback function. If the values are required after the callback returns, the program should keep a copy of them.

Working of the callback function: It is just a function that triggered another function to call it if certain conditions are fulfilled. Typically, this provides a pointer to a function in a class someplace, and the class comes to the call-my-callback point that is a function to be called. It is similar to ISR described in [3.1.1] on the Arduino with *attachInterrupt()* but it is not an ISR. That just stores the function name you pass it. Later the real ISR calls that function (by reading the variable) when it is called.

It should also be mentioned that the underlying WiFi network is utilized in these experiments and it offered a steady connection but the packet losses can happen.

Finally, this thesis is confined to studying the performance of the MQTT broker solely. There is no other protocol that is investigated or compared.

2.3. Quantitative Research Method

In this thesis, the Quantitative research method is used which includes the empirical investigation of observable and measurable variables. It is used for concept testing, prediction of outcomes, and figuring out relationships between and amongst variables.

As per the research methods, first, the novelty in the research area is identified. The proper experiment configuration will be identified. Later, the data from the investigation has been collected for further observation and analysis.

2.4. Limitations

The limitation of this experiment's setup comes from the hardware part. The specification of the Microcontroller ESP8266 has been discussed in chapter (3.1.1.2) and it has 80 KB of RAM. As the experiment continues, one thing has been noticed that one can not generate a buffer size of more than 10000 bytes which makes it impossible to publish payload size over 10000 bytes.

However, In real-life scenarios, it is quite unusual that someone is willing to transfer more than a Kb per single payload. Moreover, due to the computational power capacity of the microcontroller, there are a few limitations for the experiments which would be described in the chapter (2.4).

As for the software part, there is one limitation in the PubSubClient library discussed in the chapter (3.1.2.5). As per the default value set for the payload is 256 bytes by the system, one can not publish payload over that size which later can be overcome by overriding that value.

3. Experiment Architecture

The architecture of the entire experiment environment and their work are explained in this chapter. In the experiment setup, all of the hardware and software components, as well as their descriptions and functionality, that will be utilized throughout the experiment are explained. The execution of all the experiments, as well as general features such as configuration, preparation, and a parameter that are necessary to get the results and data from the evaluations, are detailed in the area of the experiment's chapter (3.2).

3.1. Experiment Setup

The overall Experiment’s architecture diagram is shown in Figure 6, which illustrates the overall architecture of the experiment environment.

For the research experiments, a private network has been set up. The reason behind setting up a private network is discussed in chapters (2.2.1) and (1.1). There is a use of Raspberry Pi 4 with Raspbian OS installed. Later, the Mosquitto Broker is installed and set up its parameters. A private wi-fi network has been created separately for this project.

.

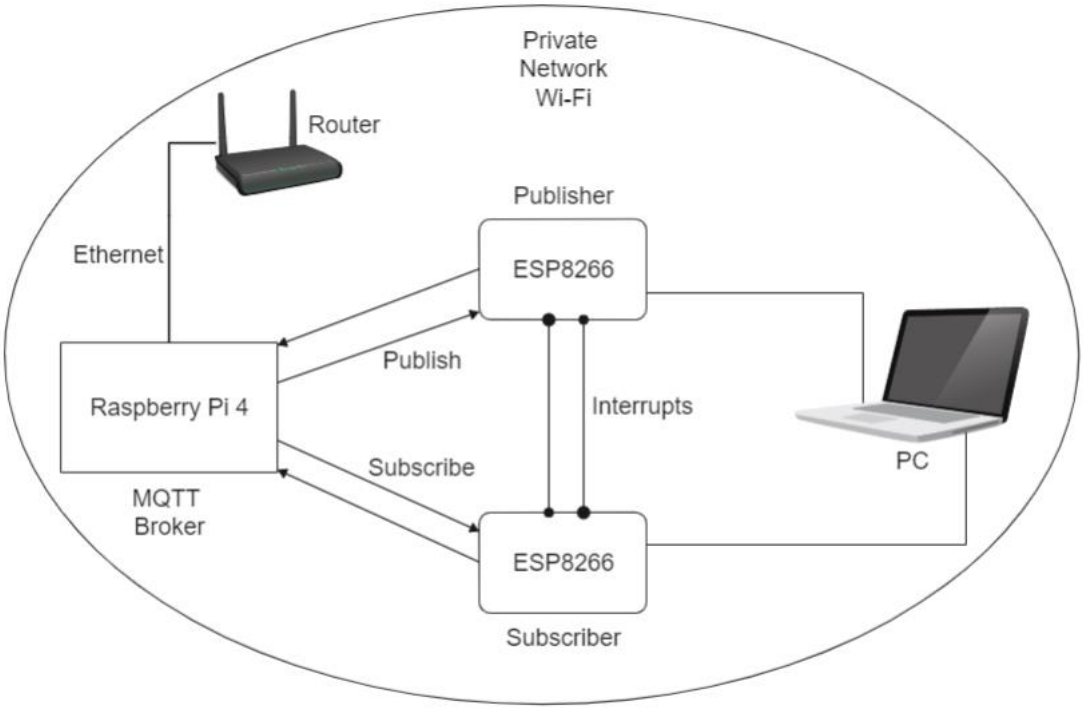

Figure 6: Overall Experiments Setup

The Raspberry Pi is connected to a private network using an ethernet interface. Two individual similar ESP8266 microcontrollers are used as Publishers and Subscribers, both of which are connected to Notebooks/PCs for Power and Instructions. For the experiments, the Interrupts has been introduced to both microcontrollers, which connect both microcontrollers with GPIO pin D2 and ground via wire (3.1.1.2). Both microcontrollers have an inbuild wifi module that is being used for connecting to a private network.

3.1.1. Hardware

All the necessary hardware components and their description will be discussed below which are used to develop an experimental environment to perform research.

3.1.1.1. Raspberry PI 4

The Raspberry Pi is a small, single-board computer, which has the functionality for use as a traditional computer/PC with the proper peripheral components. Despite being designed for a variety of purposes, the Raspberry Pi fits perfectly into the IoT environment due to its low power, low cost, and high potential for performing computing tasks and connecting to various types of sensors.

For Raspberry Pi to assist the MQTT protocol, there is a use of the server software program referred to as Mosquitto. It is a message broker and it has implemented various variations of the MQTT protocol, which includes the latest version 5.0. It is also a particularly lightweight software program, which makes Mosquitto the best preference for dealing with the MQTT protocol on our Raspberry Pi. The MQTT protocol works via way of means of having clients act as publishers and subscribers. The publishers send the messages to a broker which acts as an intermediary.

Operations performed on Raspberry pi:

1) Install Raspbian OS
2) Install mosquito MQTT broker
3) Configure properties for Ethernet connection to a router

3.1.1.2. Microcontroller ESP8266

The ESP8266 is a System on a Chip produced by the Chinese firm Espressif. It is made up of a Tensilica L106 32-bit microcontroller (MCU) and a Wi-Fi transceiver. It features 11 GPIO pins (General Purpose Input/Output pins) as well as an analog input. This means one may program it in the same way you would any other Arduino or microcontroller. It has a You also get Wi-Fi communication, so one will use it to connect to your Wi-Fi network, connect to the Internet, run a web server with genuine web pages, allow your device to connect to it. his chip has become the most popular IoT gadget on the market.

MQTT is lightweight so, it is perfect for running on small microcontrollers such as an ESP8266. For the experiment, there are two ESP8266 one as a Publisher and the second as a Subscriber has been used.

It has a set frequency of 80MHz, 80KB RAM Memory, and 4MB Flash Memory which is enough for the experiment.

Operations performed on both ESP8266:

1) Connect Interrupt wire to GPIO pin D2 and common grounds
2) Build the project and upload the firmware.

3.1.1.3. Interrupt in ESP8266

Interrupts are beneficial for making things happen automatically in microcontroller applications. With Interrupt, one does not need to continuously test the current GPIO pin value. When an alternate pin value is detected, an event is triggered, and a function known as Interrupt Service Routine (ISR) is called.

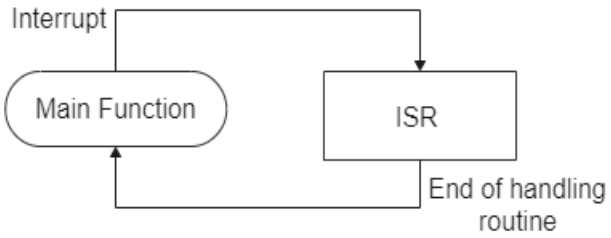

Figure 7: *Working of Interrupt in ESP8266*

When an Interrupt happens, the processor stops the execution of the main program to execute a task, after which it gets returned to the main program as shown in Figure 7. In our case, this is useful to trigger an action whenever the need for the time counter.

3.1.1.4. Wi-Fi Router

Operations performed on Wi-Fi Router:

1) Edit Router Configuration (Create a separate Network)
2) Gateway Configuration (Static IP to connect MQTT broker)

3.1.2. Software and Libraries

In this section, the software and the libraries used to develop the experiment system are described.

3.1.2.1. Mosquitto

Eclipse Mosquitto is an open-source MQTT message broker that implements the MQTT protocol versions 3.1, 3.1.1, and 5.0. It is installed on Raspberry Pi 4 board for our experiment environment.

The MQTT protocol offers a lightweight way of carrying out messages using a publish/subscribe structure. This makes it ideal for Internet of Things messaging such as with low power sensors or mobile devices such as phones, embedded computers, or microcontrollers.

The Mosquitto project also includes a C library for creating MQTT clients, including the widely popular mosquitto_pub and mosquitto_sub command line MQTT clients.

The MQTT broker/server will start automatically on system startup.

3.1.2.2. Platform IO

Platform IO is a cross-platform universal middleware interface tool to ease configuration of the SDKs, libraries, projects for writing embedded applications. It is used as a writing program and testing out the experiment for this project on Visual Studio Code editor. It is programmed with the Arduino Service Interface Programming (ASIP) model.

3.1.2.3. Python

The python script is used as a Subscriber from the notepad to receive the data generated by the ESP8622 over MQTT protocol, later the data will be stored and converted as a .xlsx file format.

3.1.2.4. Visual Studio Code

Visual Studio Code is a simple code editor that supports development tasks such as debugging, task execution, and version control. It seeks to give just the tools a developer needs for a short code-build-debug cycle, leaving more sophisticated processes to full-featured IDEs like Visual Studio IDE.

3.1.2.5. Libraries

In this section, all libraries that are used to development of the software processing for the experiment environment will be described here.

1) **Arduino**:

   This is the main included file for the Arduino SDK. This library is free software. One may redistribute and/or modify it under the term of the GNU Lesser General Public License as published by the Free Software Foundation.

2) **ESP8266WiFi**:

   It is a Wi-Fi library and provides a wide collection of C++ methods and properties to configure and operate ESP8266.

   The ESP8266 Wi-Fi library was created using the ESP8266 SDK and the naming conventions and general functionality philosophy of the Arduino WiFi library. The variety of Wi-Fi functionality translated from ESP8266 SDK to esp8266 / Arduino exceeded the Arduino WiFi library over time.

3) **PubSubClient**:

   This library allows you to send and receive MQTT messages. It is accomplished by merely implementing the protocol features. Because of the low memory and lack of a standard persistence method, it only supports Clean Sessions and does not support QoS 2 messages.

For network client libraries to implement, the Arduino platform specifies a common API. The PubSubClient supports a broad variety of Arduino-compatible devices out of the box by enabling sketches to pass in any implementation of the API.

It is utilized in a variety of production systems and was recently modified to enable MQTT 3.1.1. Other constants specified in that file regulate the version of MQTT utilized, the keepalive duration, and the various connection states the client might be in.

4) **ArduinoJson**:

It is a C++ JSON library for Arduino board that allows various operations for JSON data format.

5) **Xlsxwriter**:

It is a Python library that allows changing data format from JSON to .xlsx for easy reading of the results and data.

3.2. Experiment Areas

This chapter describes the functioning of the area of the entire experiment environment. There is an explanation of the working principle including all experiment's testing devices, their configuration, elements, and the parameters that are considered while conducting experiments. There is also mentioned about, how the data and results are generated and gathered for future analysis and storage.

Figure 8 shows the testing environment for this research, it consists of software and hardware part. The hardware part will use for testing the MQTT protocol for tramming messages from the publisher to the MQTT broker and from the broker to the subscriber. The interrupt has been connected with the GPIO pin of the both publisher and subscriber microcontroller for the feedback loop and resets the internal clock timer.

The software part in the experiments is used as the configuration of every MQTT parameter and publishing messages from the publisher and receiving it from the subscriber. As a part of the software, the Mosquitto has been installed on the Raspberry Pi 4 as an MQTT broker. Moreover, software processing is also used as a collection of the experiment results' data which is used for further analysis.

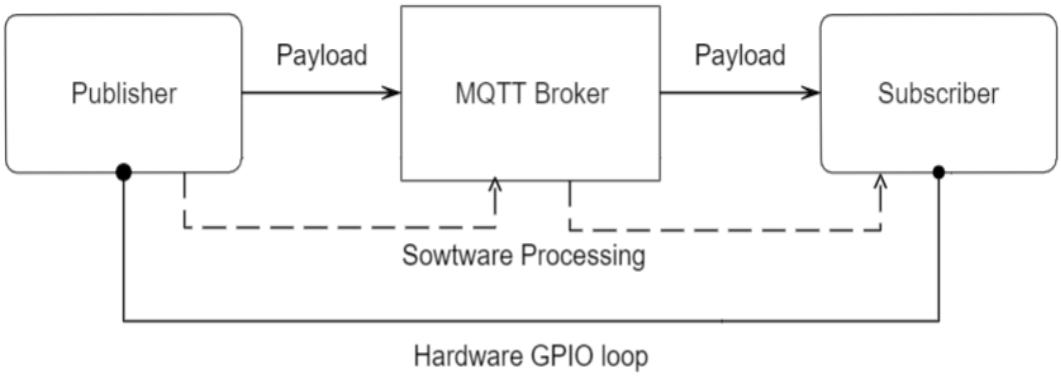

Figure 8: *MQTT testing environment*

The entire MQTT pipeline performance testing ecosystem measures the latency of various experiments in relation to parameters. The data generated from the testing is stored as a buffer under the software processing Later, the collected results/data are published in a separate topic for storage and analysis for further analysis.

To better understand the experiments on various parameters and their relations to each other that were carried out to evaluate MQTT performance, the subsections of this chapter explain the areas that are interested in the experiments and their setup, measurements, and functionalities.

3.2.1. Payload Size for Single Publisher

The MQTT protocol allows messages with a maximum size of 268435455 bytes, which equals approximately 260 MB. This is a very large message size that is restricted by most brokers, especially public and open software-based ones. Furthermore, this experiments area only focuses on a single publisher and single subscriber for message delivery.

The experiments consist of different payload sizes like 1, 64, 128, 512, 10224, 2048, 4096, 10000 bytes. The idea is to publish different payload sizes over a single Topic name. The program will send over 50 messages within a 1000 ms delay in between and the QoS level is set to be zero '0'. The process will be repeated 5 times for sample collection and an average would be considered as further analysis. Moreover, the program changes the Interrupt's pin value so, the subscriber starts/notes its internal clock time when the publisher publishes a message.

In the subscriber part, it subscribes to a particular topic, and the code calls the function mqttCallback (2.2.9) and starts listening to incoming messages over a topic. Later the message will be verified with a topic and the duration of time to delivery will be noted. This process will be repeated several times to accurate the result.

The latency of each message will be stored in a JSON buffer message as a result and now the subscriber part works as a publisher and it will publish the results over another topic to save the data for further analysis. Moreover, the process will be repeated several times to collect and compare the number of samples.

3.2.2. Multiple Publisher and Subscriber

Multiple sensors trying to publish data and also multiple subscriber topics to a broker are common in wireless sensor networks and IoT environments. thus, it is worth investigating the MQTT performance when multiple publishers and subscribers are being used.

In this experiment, publishers publish constant payload data across multiple topics and subscribers subscribe to all published topics. The idea is to check the latency of data transfer

for a higher number of topics like 1, 10, 100, 1000, 10000. The program will send over 20 messages within a 1000 ms delay in between and the QoS level is set to be zero ‘0’.

The experiments will be also done for each number of topics for payload size like 1, 128, 1024, 4096 bytes, and the process will be repeated 2 times for sample collection and an average would be considered as further analysis.

The program works as the publisher changes the Interrupt pin value to ‘low’ and starts publishing data to all the topics after completion of the publishing process it changes again interrupt pin value to ‘high’. In the subscriber part, it notes the changes in interrupt pin value and starts its internal clock for measurements, it also verifies the incoming topic and payload for the experiments. The results of the experiments are stored in JSON buffer and later they will be published over another topic to save data for further analysis.

3.2.3. Wildcard Subscription

When a client subscribes to a topic, it has the option of either subscribing to the exact topic of a published message or using wildcards to subscribe to multiple topics at once. Both strategies and also the two different kinds of wildcards can play an important part in the performance of MQTT. Thus, it is necessary to check how the wildcards affect the performance and latency for message transfer. These experiments consist of two separate methods of investigation described below.

3.2.3.1. Latency when Subscribing to MQTT broker

Since wildcards are only used during subscribing to multiple topics simultaneously, there is a need to understand how this affects the overall performance of MQTT and latency for data transfer.

In these experiments, there is a focus on the main 4 types of subscriptions to topics like subscribe without wildcards, subscribe with Single-level “+” wildcard, subscribe with Multi-level”#” wildcard, and combination of a Single-level and Multi-level wildcard.

All different types of subscription methods will be tested on a different number of topics and the results will be published on another topic for analysis purposes. The analysis of the experiments will only be based on, the time the subscriber takes to subscribe all topics to the MQTT broker not an overall message transfer time for the data.

3.2.3.2. Message latency on multiple topics

The analysis of the experiments will only be based on the time the publisher publishes the message till the end subscriber receives it. Instead of the only subscription to the MQTT broker. This will be tested for all 4 types of subscriptions mentioned in the above chapter (3.2.3.1) with consideration of multiple topics. Each type of subscription will be tested on a larger number of topics such as 1, 100, 10000, 1000000, 100000000 topics to subscribe.

The program works as, Publisher publishes the message over serval multiple topics, the publisher changes the Interrupt pin value to "low". Later the subscriber part notes the Interrupt value alongside subscribing to topics with one of the kinds of subscription methods.

The callback function is called due to Interrupt and starts receiving messages. It verifies around 50 samples and their topic. At last, the time latency will be saved on the JSON buffer, later it will be saved to the disk.

4. Experiment Results

This chapter is focused on the interpretation of the data fining from the experiment's setup which has been established for the research environment. The analysis will be undertaken concerning the hypotheses and will visualize the data to make it easier to understand. This section will interpret latency findings. Lastly, the outcomes will be compared with the hypotheses.

4.1. Payload Size For Single Publisher

The essential details of this experiment are discussed in the chapter (3.2.1). The primary purpose of this section of the experiment was to know how the payload size impacts the latency of the MQTT.

Throughout the initial experiment, It was determined that the overall time required in the system from publishing payload by the publisher to receiving payload in subscribers appears to be altered with different delay interval durations for publishing payload. In another word, the interval time of continuous publication of the message seems to affect the overall latency of message delivery of MQTT. There is numerous elements have been discovered during the early stages of the investigation. Those elements required an explanation to do an additional evaluation of the research and ensure the authenticity of the outcome. Those two elements' conclusions are provided in the chapter (4.1.1) and (4.1.2). Chapter (4.1.3) gives the outcome and analysis of the main research which is to analyze the MQTT, configurated on the different payload sizes.

4.1.1. Constant Payload with Different Delay Interval

To evaluate this section of the experiment 1 byte of the payload has been picked for constant payload which brings the overall minimum packet size to 3 bytes explained in

the chapter (1.2.4). The various time delay interval like 1ms, 100ms, 500ms, 1000ms, 2000ms, 10000ms was determined for publishing a continuous message.

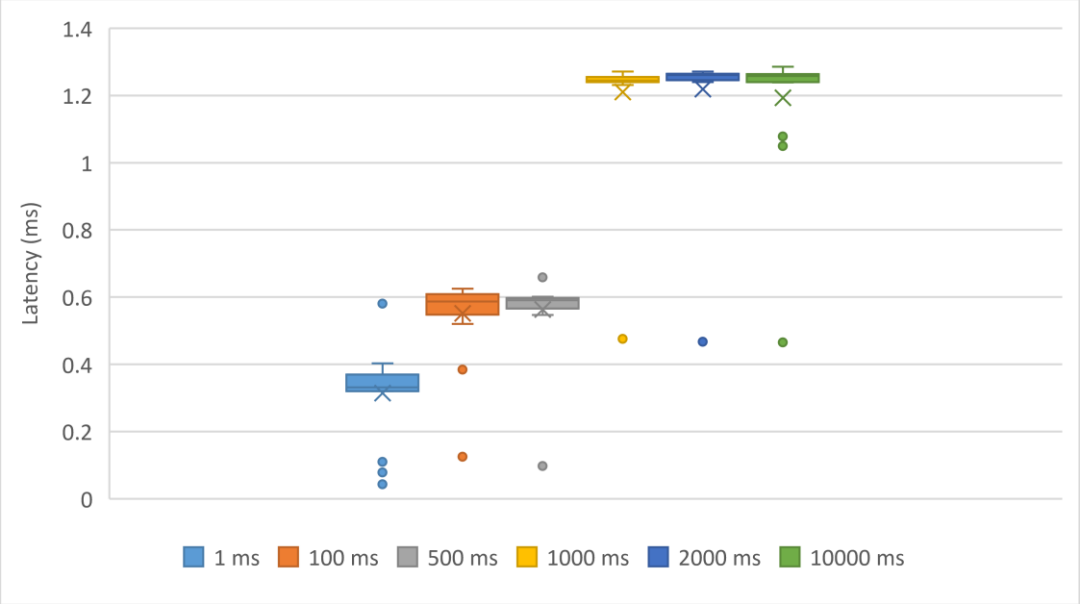

Figure 9*: Latency when publishing constant payload with different time delay intervals*

The results indicated that the amount of time required by the publisher for publishing the message during the varied time delay between two publishing messages appears to vary. The accompanying box diagram illustrates the sample of 20, throughout the various time delay periods.

As shown in Figure 9 when the interval is 1 ms, the time delay of publishing the message is the lowest ass 0.3-0.4 ms (To easily read data converted from microsecond to millisecond). And it appears stable at higher delay intervals like 10000ms. There is just one noticeable growing trend in the graph, the latency increased as the time of delay in each interval increased. Moreover, the growth is exponential in general and it appears steady if the delay is higher than 1000ms.

After evaluating this experiment and studying about the capabilities of the microcontroller and MQTT, a delay of 100ms to 1000ms looks more reliable and acceptable for the research's further progress, since a 1ms delay is relatively minimal in the presence of heavy payload size. because the publisher must send numerous packets. It is unnecessary if the delay is higher than 2000ms since the publisher is ideal throughout the delay time.

4.1.2. Constant Delay Interval with Different Payload

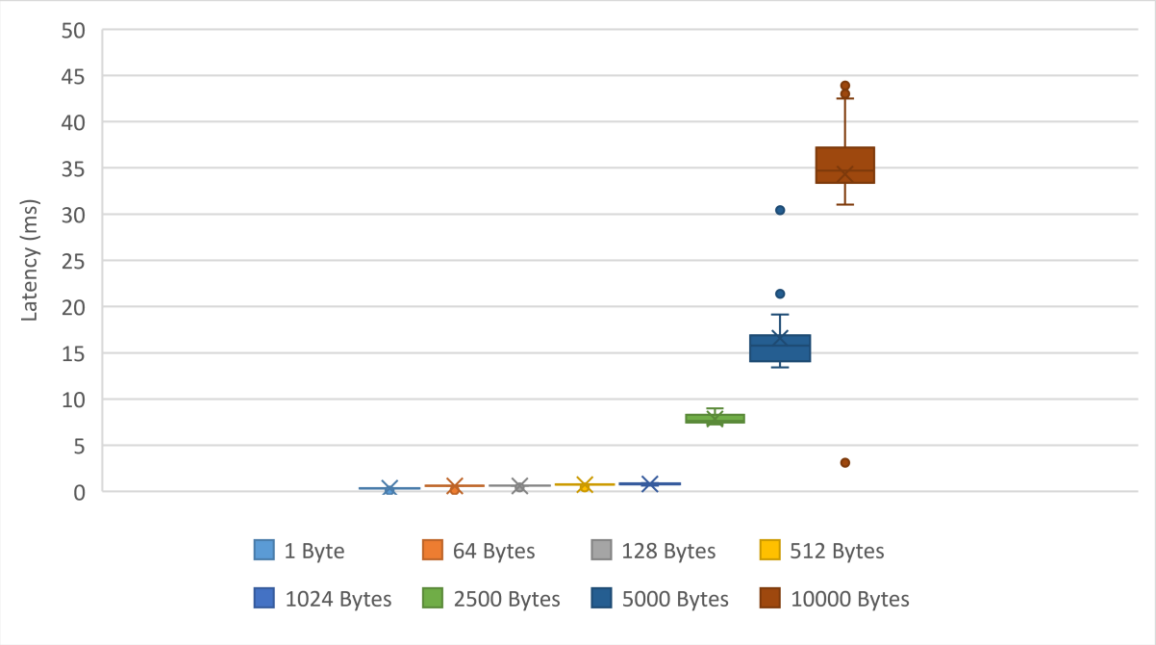

Figure 10: *Latency when publishing different payload with constant time delay intervals*

This experiment is continued from the prior experiment 4.1.1. Figure 10 illustrates that there is no other component that affects the change in overall latency for transmitting a message, only by delay in interval rather than payload size published by the publisher. both investigations confirmed that there is not any factor that affects the latency in MQTT.

After examining these two experiments, the main experiment can be configurated to ensure the accuracy of the results obtained.

4.1.3. Different Payload Size for Single Publisher

After analysis of the result from the chapter (4.1.1) and chapter (4.1.2), The time delay interval while publishing messages is set to 1000 ms for the following experiments.

The preparation of these experiments is discussed in the chapter (3.2.1). As an outcome, The time delay concerning the different configurations of the Payload size on MQTT is given extensively by Figure 11 and Figure 12. And the Behaviour of overall and initial responses of publishing messages are visualized in Figure 13 and Figure 14 respectively.

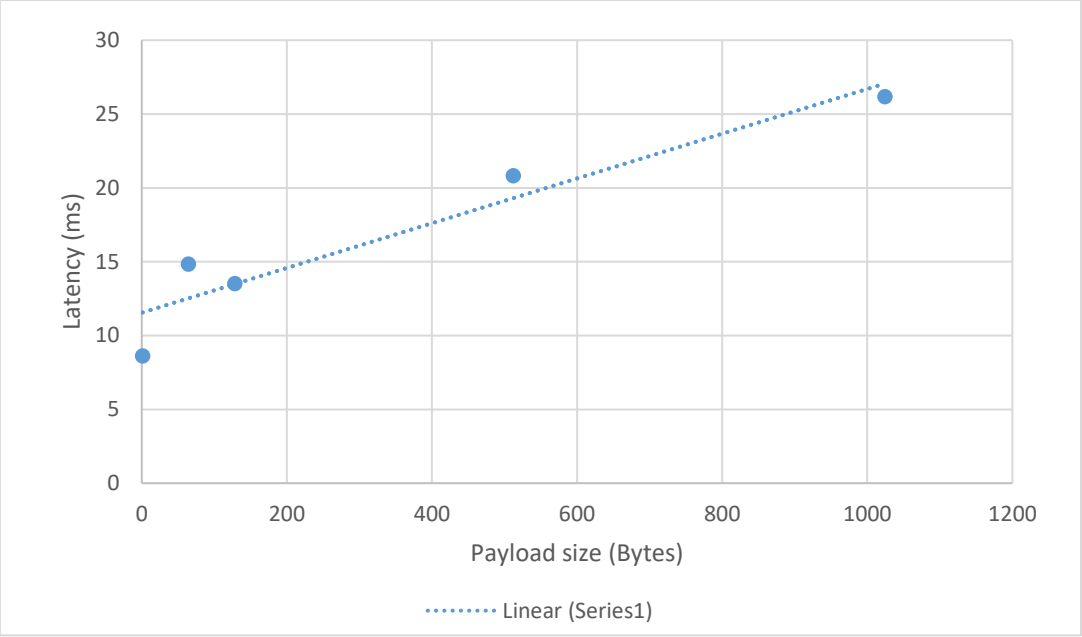

Figure 11: *Average Time Latency for Various Payload Size (1-1024 bytes)*

According to Figure 11, the Average latency for sending a message using MQTT with payload size from 1 to 1024 bytes illustrates the increasing the payload size leads to increasing the delay and trends to follow the linearity.

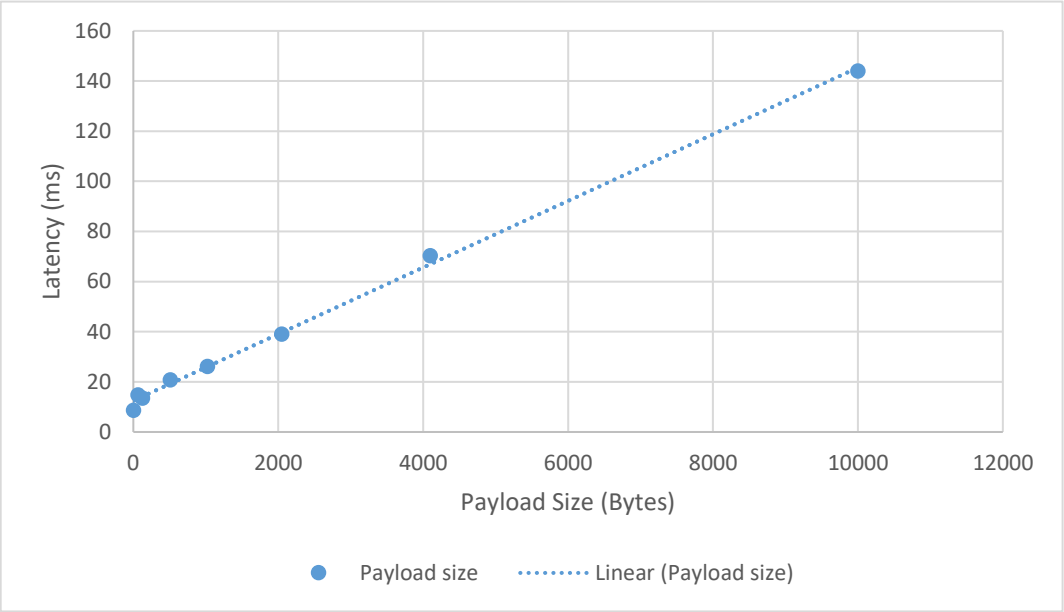

Figure 12: *Average Time Latency for Various Payload Size (1-10000 bytes)*

The initial research findings demonstrate that the results are given in chapter (4.2) for higher payload size exhibit different behavior for a different parameter like the number of publishers and subscribers. To confirm that concept, the data of these experiments analyze independently, one with a smaller payload size(1-1024 bytes) and the second with (1- 10000 bytes) of payload size.

Figure 12 illustrates the time latency graph with payload size 1-10000 bytes similarly following the linear trendline. One can conclude that when the parameter is configurated as a single publisher, without wildcard subscriptions, QsS 0, the payload size linearly influences the performance latency of the MQTT.

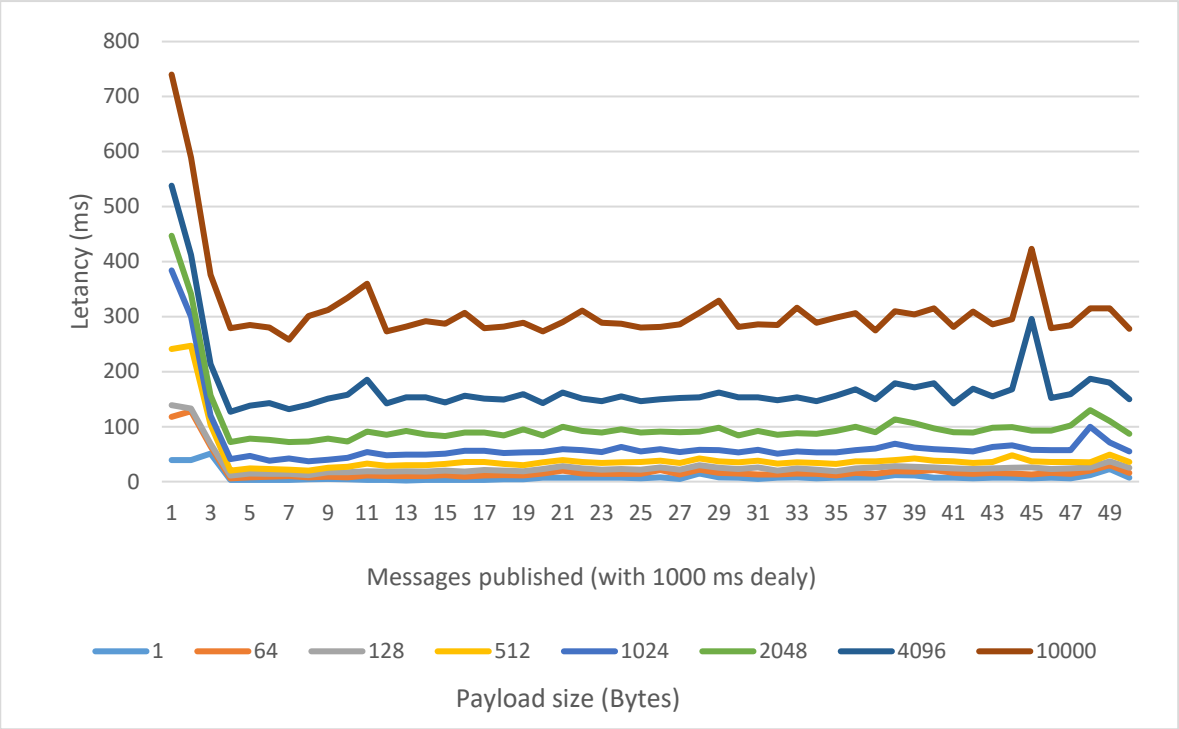

Figure 13: *Time latency for different payload sizes for initial 50 messages*

Figure 13 shows the time latency for different payload sizes for the initial 50 messages which are published with a 1000 ms delay interval. The latency graph appears to follow steadily after a few initially published messages for sample numbers 1 to 4 of all various payload sizes.

Figure 13 clearly demonstrates that when the payload size increases from 2048 bytes to 10000 bytes, the error or margin increases in the graph. This result analysis supports the findings in chapter (4.1.2).

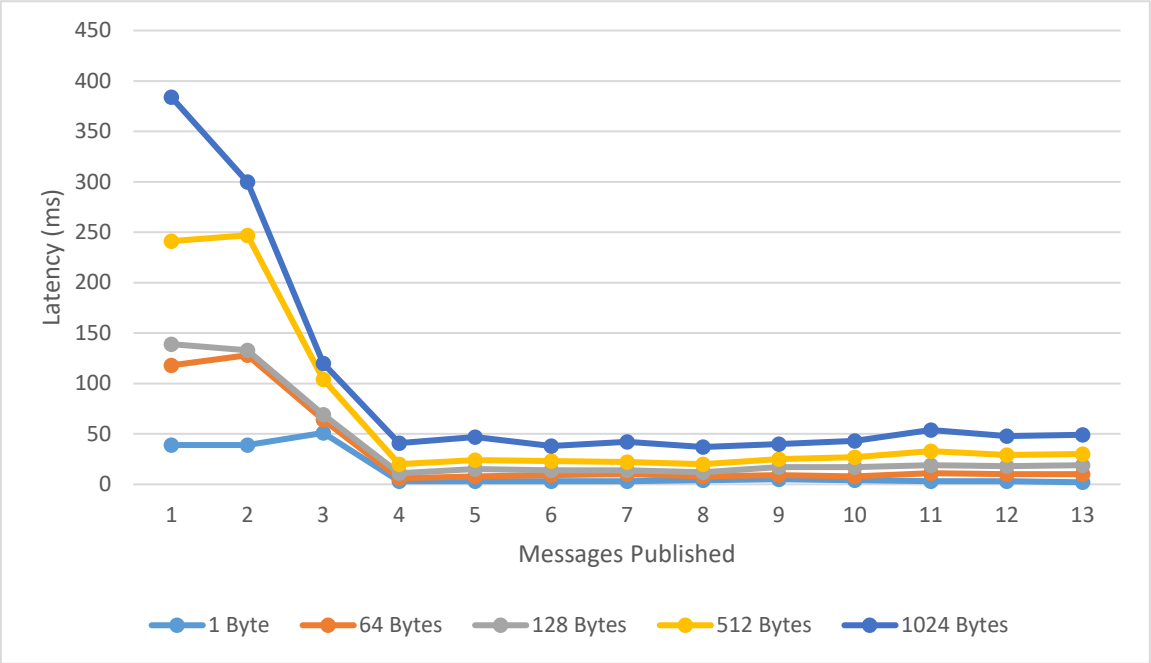

Figure 14: *Initial time latency for different payload sizes on published message*

Figure 14 shows the enlargement of Figure 13 during the initial phase for payload size 1-1024 bytes. This implies the first 3 to 4 published messages have higher latency than it appears to decrease and become more steady and have low latency. This type of behavior did not anticipate for the research, from a theoretical point of view as the clients(subscriber) have already established the connection before receiving any packet from the MQTT broker. The initial explanation would be, the broker is taking initial time to establish a strong connection with the subscriber. However, these findings required additional investigation which belongs to the section of future work for this experiment.

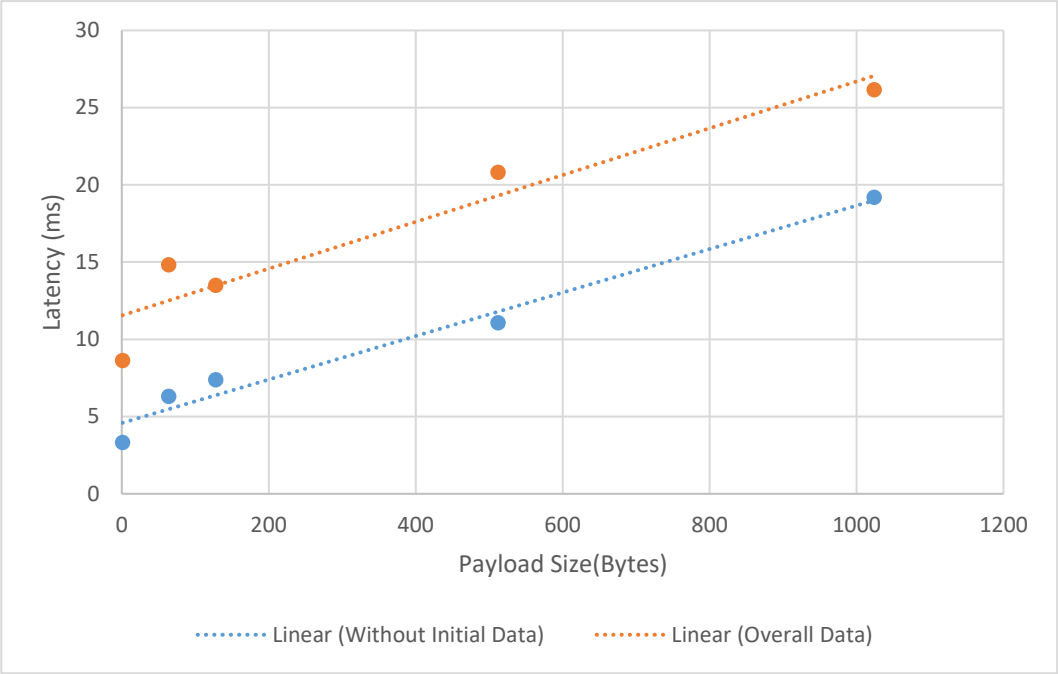

Figure 15: *Average latency with vs without initially published messages over different payload sizes*

Figure 15 demonstrates the comparison of average latency for with and without the initially published messages for 1-1024 bytes of payload size. The latency trendline of both with and without initially published messages shows a ~5 ms difference in latency delay.

To conclude this experiment, If the parameter like the number of the publisher, wildcard subscription, and QoS is set to constant and evaluate the latency in MQTT only base on payload size, the latency in MQTT shows the linear increments with respeccing to linear increment in payload size.

4.2. Multiple Publisher and Subscriber

The research and discussion of these experiments are covered in the chapter (3.2.2). The key purpose of this section is to illustrate how the multiple publisher and subscribers to that particular published topic influence the performance of the MQTT by overall transmission latency of the constant payload size.

In the initial stage of this experiment, it has been observed that The results appear to be pretty varied with different parameters, especially when the number of publishers varies. After some research (try and error), the conclusion comes with an explanation that, the result of the experiments should be examined independently, one with 1 to 128 number of publishers and second as a 128 to 8192 number of a publisher the result is displayed in Figure 16 and Figure 17 respectively.

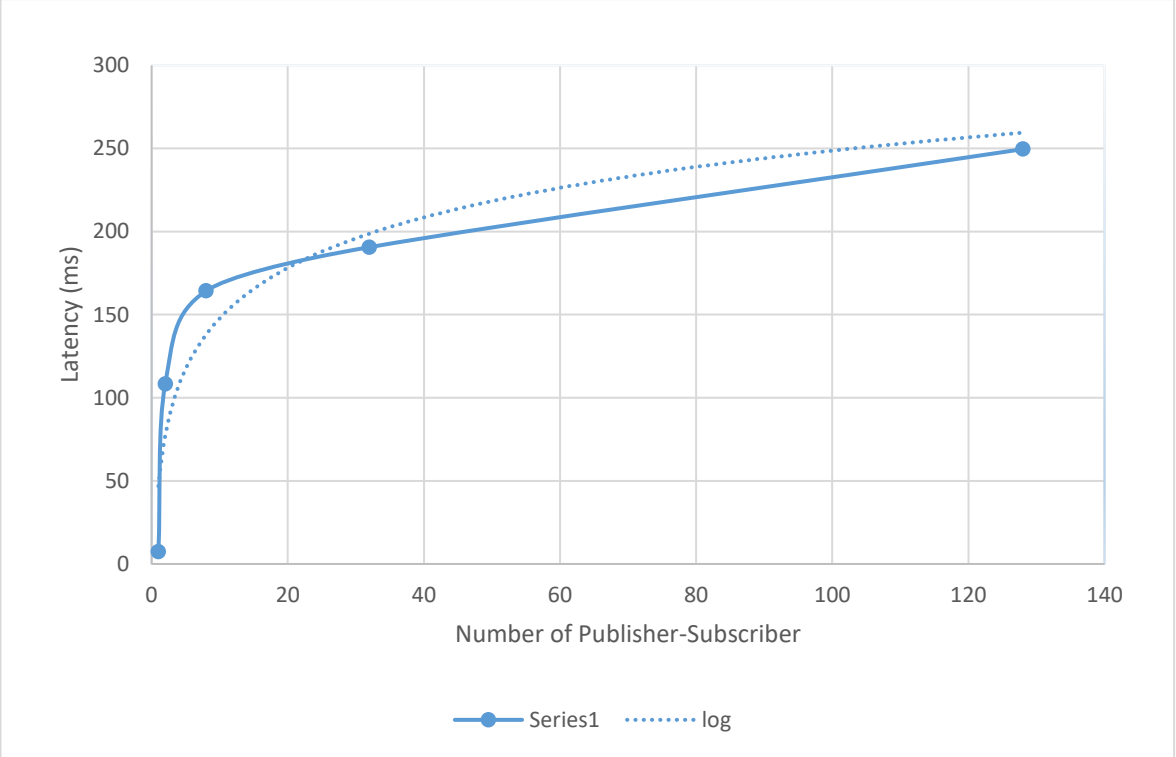

Figure 16: *Average latency of 1 to 128 numbers publisher/subscriber on the constant payload (1 byte)*

Figure 16 shows that when the number of publisher-subscribers increases from 1 to 128 the latency for MQTT transmission increases logarithmically. As a consequence, the QoS configuration is set to "0," and the retained messages are set to "false." Furthermore, the experiment was carried out while keeping the payload size constant and at a minimum of 1 byte.

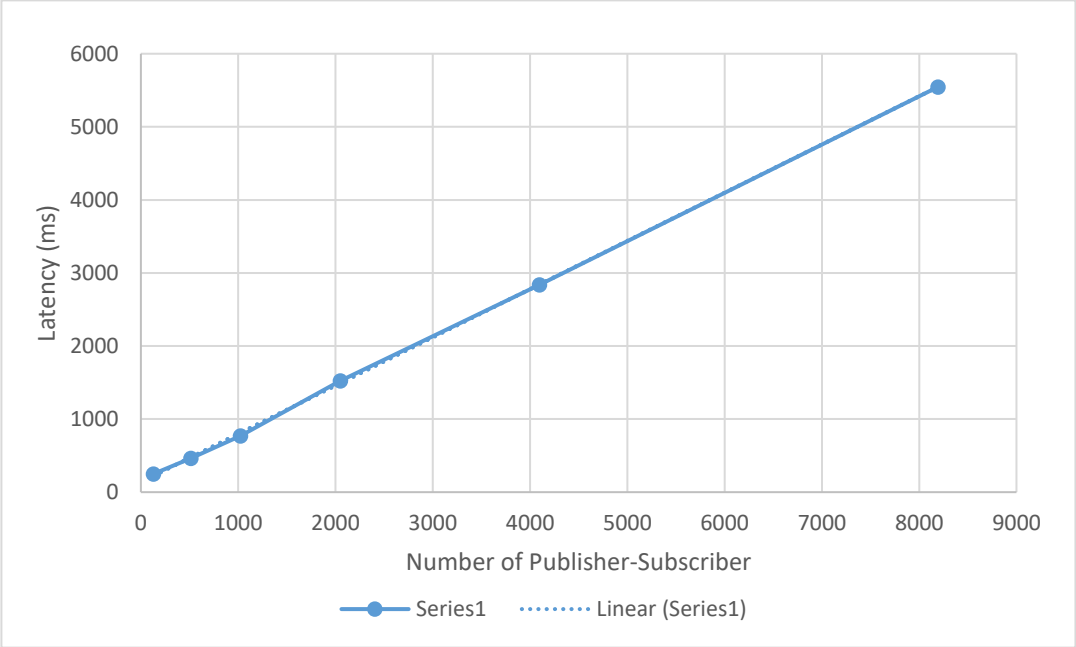

Figure 17: *Average latency of 128 to 8192 numbers publisher/subscriber on the constant payload (1 byte)*

Based on the early experiments, it has been observed that the higher the number of publish-subscribers, the different the behavior than the lower the number of publish-subscribers. When a consequence, Figure 17 illustrates that as the number of publisher-subscribers increases between 128 to 8192, latency increases, and the curve follows the linear extrapolation. In comparison, Figure 16 demonstrates logarithmic progression as numerous topics span between 1 to 128.

Payload (Bytes) / Number of Topics	1	128	1024	4096
1	8 ms	11 ms	26 ms	72 ms
10	24 ms	47 ms	132 ms	418 ms
100	57 ms	173 ms	1040 ms	
1000	315 ms	1588 ms	10903 ms	
10000	3083 ms	16096 ms		

Table 1: *Latency concerning payload size and number of published topics*

Analisis from Table 1 reveals that the color gradient in the table indicate where each cell value falls/increase within that range. As a consequence, it can observe that increasing the payload size resulted in an increase in the latency in MQTT as demonstrated in the chapter (4.1.3) the increase in the payload size results in an increase in the latency in MQTT as demonstrated in chapter (4.1.3).

Moreover, increasing the number of publisher-subscriber also resulted in increasing the latency of MQTT as shown in Figure 16 and Figure 17. Table 1 further illustrates that, if the number of publisher and payload increases, latency also get higher. One may remark that there is some empty value for certain parameters leading to higher latency time and/or the buffer size of the microcontroller which function as a publisher and not be sufficient to interact up with 5000-10000 delay intervals when publishing messages.

To conclude the analysis of these experiments, the latency of MQTT for transmitting the message from publisher to subscriber also depend on the number of publisher-subscriber as discussed above latency increase logarithmically when the number of publishers is between 1 to 128 and it linearly increases when the number of publishers is between 128 to 8192 when the payload size is constant and QoS set to 0.

4.3. Wildcard Subscription

The experiment area of these sections is mentioned in the chapter (3.2.3). This experiment is focused on major two research finding first is to investigate the latency while subscribing to the MQTT broker with a different kind of subscription like without wildcards, subscribe with Single-level “+” wildcard, subscribe with Multi-level ”#” wildcard have been examined and compared with each other with the multiple numbers of the topics and combination of a Single-level and Multi-level wildcard and the second is message transmission latency from publisher to subscriber with a different kind of subscription.

4.3.1. Latency when Subscribing to MQTT broker

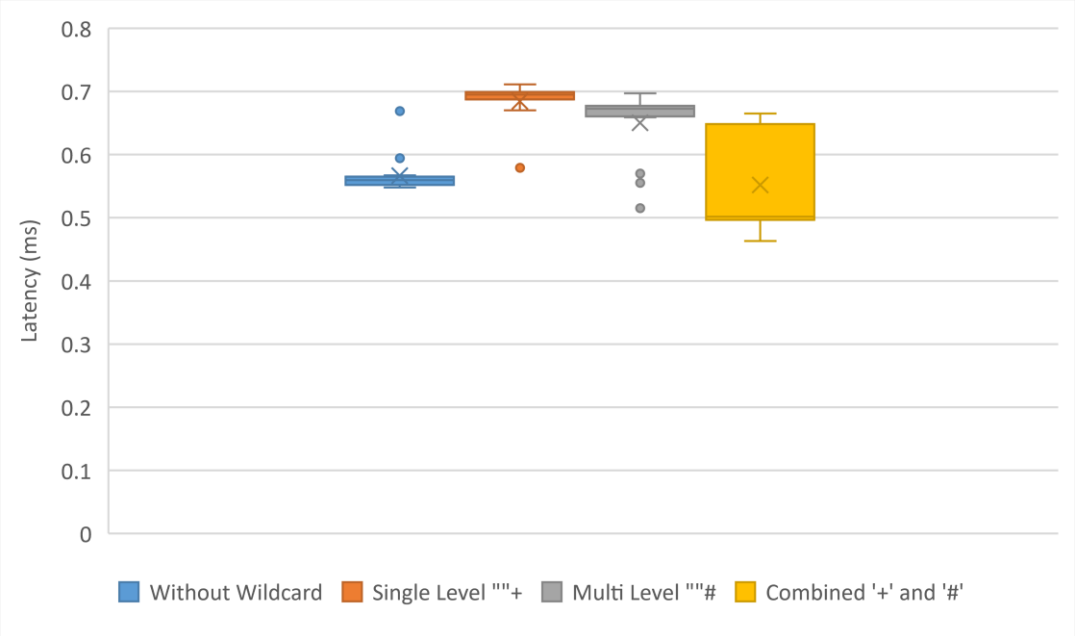

Figure 18: *Latency when subscribing to MQTT broker with and without wildcards*

Figure 18 box graph shows the latency to subscribing to the topic by a subscriber with a different configuration. The graph is made of 20 individual samples and time latency measure in a microsecond (In graph converted to a millisecond).

Without a wildcard subscription which subscribes to only one topic at a time seems to have a low latency time for subscribing to a topic. However, the single and multi-level wildcard subscriptions to topic subscribe 100 and 10000 topics respectively but show the almost same latency time. Moreover, the comparison of without and multi-level subscription difference in latency is just ~0.2ms which seems quite low if one can compare the number of topics subscribed by them is 1 and 10000 respectively.

From the graph, it is also noticed that the interquartile range of configuration of without, single-level, and multi-level wildcard subscriptions are very small compared to a combination of single and multi-level wildcard subscriptions. However, the range is between ~0.5ms and ~0.7ms from which it can be analyzed that the latency of subscribing to topic/topics with or without any wildcards does not affect much in MQTT performance.

4.3.2. Message latency on multiple topics

The results of the following experiment show how latency varies with a different kind of subscription across a variety of subjects to the subscriber. This experiment is based on two payload sizes: 1 byte and 128 bytes.

As illustrated in Figure 19, the lack of a wildcard subscription appears to produce the same results as indicated in the chapter (4.2), namely an increase in latency as the number of publishers increases. The single and multi-level subscriptions appear to follow the increasing trend, but the latency is slightly lower than without wildcard subscriptions.

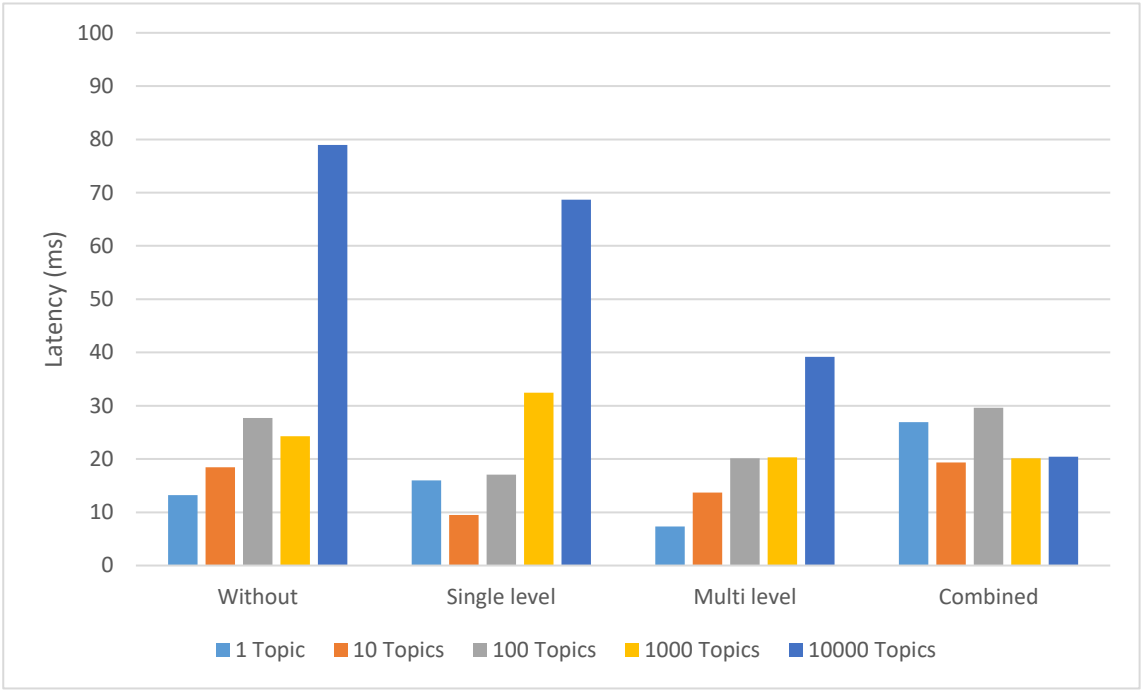

Figure 19: *Comparison of a different kind of subscription concerning a various number of the publishers with constant payload 1byte*

When comparing single and multi-level wildcard subscriptions, the results reveal that when the number of subscribing topics increases from 1000 to 10000, latency during multi-level subscriptions decreases considerably.

Furthermore, the graph with the lowest latency may be seen with a combination of single and multi-level wild card subscriptions. Therefore, if the number of subscription subjects increases, latency does not change significantly.

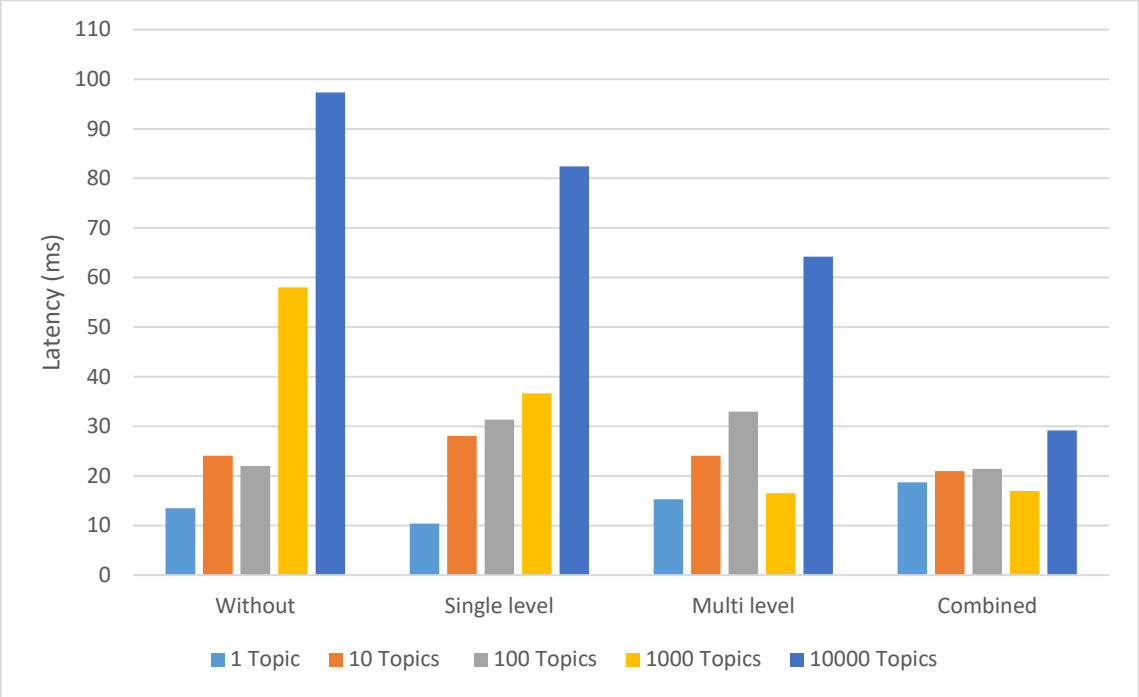

Figure 20: *Comparison of a different kind of subscription concerning a various number of the publishers with constant payload 128 bytes*

In order to compare the data from Figure 19 and Figure 20, The overall delay of any subscription appears to increase as the payload size increases from 1 byte to 128 bytes.

To summarize this portion of the experiment, wildcard subscription to multiple topics has an impact on MQTT performance, particularly when the subscriber connects to the MQTT broker and the transmission time of the message from the publisher to the subscriber.

5. Conclusion

When it comes to IoT research, there are several areas to explore. This thesis focuses on the state-of-the-art performance evaluation of the MQTT protocol, considering it is the most used protocol in IoT applications. Several experiments were carried out in this paper to examine the performance of MQTT in various configurations such as payload, number of publisher-subscribers, and wildcard subscription.

According to the results of the first experiment, increasing the payload size for transmission from publisher to subscriber increased the latency in MQTT performance. Especially, the other configurations such as multiple topics and QoS set to constant.

The second experiment's results indicate that increasing the number of topics to publish results in an increase in MQTT latency. The increment can be divided into two categories: one for 1 to 128 number of topics, where the increment is logarithmic, and another for 128 to 8192 number of topics, where the increment in latency follows a linear trendline.

Tirth, various kinds of subscription methods have been tested, such as without wildcard, single-level wildcard, multi-level wildcard, and single and multi-level wildcard combination for a serval number of topics to be subscribed. The results show that the latency when subscribing with multi-level and combination of single and multi-level wildcard has lower latency compared to other kinds of subscription.

When it comes to the thesis's purpose, several domains have been examined and extensively investigated in order to evaluate the performance of the MQTT. The results of the experiment and their interpretation demonstrate the satisfaction of achieving the aim. However, there are a few constraints and configurations that need to be considered to completely evaluate the MQTT.

Finally, this research thesis provides a broad understanding of how various configurations impact MQTT performance and may be used as a benchmark assessment for IoT system applications, and as a helpful reference for system designers.

5.1. Future Work

As for the future work for this thesis, numerous factors need to be researched and experimented with for the proper evaluation of the MQTT. Since this research thesis focuses on a specific configuration of MQTT, the number of configurations might be extended and studied for more comparison of findings. There are a few elements that may be required for additional investigation, which are listed below.

First and foremost, this thesis is primarily concerned with QoS level 0 and the retain message set to false. The same experiments can be conducted on this thesis for QoS levels 1 and 2 .which will use in the future to compare and generalize the findings.

Second, the experiments on wildcard subscriptions in this thesis need further exploration and a different approach to fully grasp how the wildcard works.

Third, throughout the thesis, all experiments were performed on a specific broker Mosquitto, and when comparing the findings of this research with other brokering software, it might acknowledge or reject the relationships observed so far. These aspects could also be taken into account for the future development of performance research.

Finally, a proper explanation is required for, why the delay in an interval of publishing the continuous message impacts the latency in publishing the message, which is described in the experiment performed on the payload size.

Bibliography

[1] P. Sethi and S. R. Sarangi, “Internet of Things: Architectures, Protocols, and Applications,” *Journal of Electrical and Computer Engineering*, vol. 2017, 2017, doi: 10.1155/2017/9324035.

[2] B. Mishra and A. Kertesz, “The use of MQTT in M2M and IoT systems: A survey,” *IEEE Access*, vol. 8, pp. 201071–201086, 2020, doi: 10.1109/ACCESS.2020.3035849.

[3] J. Voas, N. Kshetri, and J. F. Defranco, “Scarcity and Global Insecurity: The Semiconductor Shortage,” *IT Professional*, vol. 23, no. 5, pp. 78–82, 2021, doi: 10.1109/MITP.2021.3105248.

[4] “State of IoT 2021: Number of connected IoT devices growing 9% to 12.3 B.” https://iot-analytics.com/number-connected-iot-devices/ (accessed Oct. 26, 2021).

[5] “mqtt-v5.0-os Specification URIs,” 2019, Accessed: Nov. 08, 2021. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.htmlhttps://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.pdfhttp://docs.oasis-open.org/mqtt/mqtt/v5.0/cos01/mqtt-v5.0-cos01.docx

[6] “Eclipse Mosquitto.” https://mosquitto.org/ (accessed Nov. 08, 2021).

[7] B. Mishra, B. Mishra, and A. Kertesz, “Stress-Testing MQTT Brokers: A Comparative Analysis of Performance Measurements,” *Energies 2021, Vol. 14, Page 5817*, vol. 14, no. 18, p. 5817, Sep. 2021, doi: 10.3390/EN14185817.

[8] “IoT Latency and Power consumption”.

[9] J. E. Luzuriaga, M. Perez, P. Boronat, J. C. Cano, C. Calafate, and P. Manzoni, “A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks,” *2015 12th Annual IEEE Consumer Communications and Networking Conference, CCNC 2015*, pp. 931–936, Jul. 2015, doi: 10.1109/CCNC.2015.7158101.

[10] B. Mishra, “Performance Evaluation of MQTT Broker Servers,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10963 LNCS, pp. 599–609, May 2018, doi: 10.1007/978-3-319-95171-3_47.

[11] “MQTT Protocol | Messaging & Data Exchange for the IoT.” https://www.hivemq.com/mqtt-protocol/ (accessed Nov. 08, 2021).

[12] “MQTT - The Standard for IoT Messaging.” https://mqtt.org/ (accessed Nov. 08, 2021).

[13] N. Naik, “Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP,” *2017 IEEE International Symposium on Systems Engineering, ISSE 2017 - Proceedings*, Oct. 2017, doi: 10.1109/SYSENG.2017.8088251.

[14] S. Wagle, “Semantic data extraction over MQTT for IoTcentric wireless sensor networks,” *2016 International Conference on Internet of Things and Applications, IOTA 2016*, pp. 227–232, Sep. 2016, doi: 10.1109/IOTA.2016.7562727.
[15] D. Dinculeană and X. Cheng, “Vulnerabilities and Limitations of MQTT Protocol Used between IoT Devices,” *Applied Sciences 2019, Vol. 9, Page 848*, vol. 9, no. 5, p. 848, Feb. 2019, doi: 10.3390/APP9050848.
[16] O. Sadio, I. Ngom, and C. Lishou, “Lightweight Security Scheme for MQTT/MQTT-SN Protocol,” *2019 6th International Conference on Internet of Things: Systems, Management and Security, IOTSMS 2019*, pp. 119–123, Oct. 2019, doi: 10.1109/IOTSMS48152.2019.8939177.
[17] I. D. Gheorghe-Pop, A. Kaiser, A. Rennoch, and S. Hackel, “A Performance Benchmarking Methodology for MQTT Broker Implementations,” *Proceedings - Companion of the 2020 IEEE 20th International Conference on Software Quality, Reliability, and Security, QRS-C 2020*, pp. 506–513, Dec. 2020, doi: 10.1109/QRS-C51114.2020.00090.
[18] “What is MQTT? Why use MQTT? Why MQTT is one of the best network protocols for the Internet of Things – IBM Developer.” https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/ (accessed Nov. 08, 2021).

Index of symbols and abbreviations

IoT	Internet of things
MQTT	Message-Queuing Telemetry Transport
CoAP	Constrained Application Protocol
AMQP	Advanced Message-Queuing Protocol
HTTP	Hypertext Transfer Protocol
M2M	Machine To Machine
TCP	Transmission Control Protocol
IP	Internet Protocol
QoS	Quality of Service
OASIS	Organization for the Advancement of Structured Information Standards
ISO	International Organization for Standardization
HiveMQ	Enterprise for MQTT broker
TLS	Transport Layer Security
RAM	Random Access Memory
OS	Operating System
JSON	JavaScript Object Notation
API	Application Programming Interface
IDE	Integrated Development Environment
SDK	Software Development Kit
GNU	An operating system that is free software
DOI	Digital Object Identifier

CD/DVD content

CD/DVD attached to this work contains the following:

- Master_Thesis_BV.pdf
- Source_codes
    - Collect_data
    - MQTT_publisher.
    - MQTT_Subscriber
    - Scripts_for_different_experiments
- Datasets

Figure index

Table index