Universidade de Aveiro
2021

Pedro Afonso
Lopes Costa

# Suporte a Comunicações em Tempo Real usando MQTT com base em SDN

# Supporting Real-Time Communications using SDN-based MQTT

**Universidade de Aveiro**
**2021**

**Pedro Afonso Lopes Costa**

# Suporte a Comunicações em Tempo Real usando MQTT com base em SDN

# Supporting Real-Time Communications using SDN-based MQTT

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica do Doutor Paulo Bacelar Reis Pedreiras, Professor Auxiliar da Universidade de Aveiro, e do Doutor Luís Emanuel Moutinho da Silva, Professor Adjunto Convidado da Escola Superior de Tecnologia e Gestão de Águeda.

Este trabalho é financiado pela FCT/MCTES através de fundos nacionais e quando aplicável cofinanciado por fundos comunitários no âmbito do projeto UIDB/50008/2020-UIDP/50008/2020.

Dedico esta dissertação à minha família e amigos pelo apoio que me deram.

## **agradecimentos** / **acknowledgements**

Gostaria de agradecer, em primeiro lugar, aos meus orientadores, Professor Doutor Paulo Pedreiras e Professor Doutor Luís Moutinho, pela excelente orientação e dedicação que proporcionaram ao longo deste percurso. Obrigado por estarem disponíveis sempre que precisasse, pela paciência em tolerar os meus erros e pela forma como me trataram. Espero que tenham tido tanto gosto em trabalhar comigo como eu tive com ambos.

**Palavras Chave**

Software-Defined Networking, MQTT, Internet das Coisas, Indústria 4.0.

## Resumo

A disseminação de dispositivos tipicamente utilizados no âmbito da Internet das Coisas no meio industrial levou à adoção neste ambiente dos protocolos típicos dessas aplicações. No entanto, esses protocolos não foram projetados para o contexto industrial e como tal não suportam os níveis de qualidade de serviço exigidos neste tipo de ambiente, nomeadamente no que toca à fiabilidade e pontualidade das comunicações. Um dos protocolos tipicamente utilizados é o MQTT que, apesar de apresentar alguns mecanismos de qualidade de serviço, não oferece as garantias necessárias em ambientes industriais. No entanto, a utilização de protocolos de rede definida por software pode ajudar a mitigar as limitações destes protocolos, complementando-os com os serviços de qualidade disponíveis nos protocolos como OpenFlow.
O MQTT v5.0 abriu a possibilidade de estender o protocolo através propriedades definidas pelo utilizador, as chamadas *user properties*. Nesse sentido, este trabalho propõe tirar partido dessas propriedades e integrar o protocolo MQTT com o protocolo OpenFlow, permitindo aos equipamentos de Internet das Coisas tirar proveito dos mecanismos de qualidade de serviço existentes no OpenFlow.

**Keywords** Software-Defined Networking, MQTT, Internet of Things, Industry 4.0.

**Abstract**

The dissemination of devices typically used in the context of the Internet of Things in the industrial environment led to the adoption of protocols typical of these applications. Such protocols, however, were not designed for the industrial context and consequently do not support the Quality-of-Service level typically required in this type of environment, namely in terms of reliability and timeliness of the communications. One of the protocols typically used is MQTT, which, despite providing some Quality of Services mechanisms, does not fulfill for the requirements demanded by industrial environments. However, the use of Software-Defined Networking protocols can help to mitigate the limitations of the typical protocols of such equipment, complementing them with suitable Quality-of-Service mechanisms.
MQTT v5.0 opened the possibility to extend the protocol through user defined properties. In this sense, this work proposes to use these properties and integrate the MQTT protocol with the OpenFlow protocol, allowing the Internet of Things devices to take advantage of the quality of services existing in OpenFlow to enable its use in industrial applications.

# Contents

# List of Figures

# List of Tables

# Glossary

| | | | |
|---|---|---|---|
| **API** | Application Programming Interface | **LAG** | Link Aggregation Group |
| **GE** | Gigabit Ethernet | **M2M** | Machine to Machine |
| **CLI** | Command Line Interface | **OASIS** | Organization for the Advancement of Structured Information Standards |
| **GUI** | Graphical User Interface | | |
| **CPS** | Cyber-Physical Systems | **ONF** | Open Network Foundation |
| **DSCP** | DiffServ Code Point | **QoS** | Quality of Service |
| **IIoT** | Industrial Internet of Things | **SDN** | Software-Defined Networking |
| **IoT** | Internet of Things | **TLS** | Transport Layer Security |

CHAPTER 1

# Introduction

**Contents**

*T*he introduction of concepts such as Internet of Things (IoT), Cyber-Physical Systems (CPS) and Cloud Computing in industry marked the beginning of the fourth industrial revolution; Industry 4.0. The previous industrial revolution, introduced, throughout the 1970s, digital electronics and communication technologies to the industry, whereas the current revolution aims to further digitalize the industry with increased automation[1].
In the context of the industry, (Industrial Internet of Things (IIoT)) is the interconnection of devices, equipment, and management platforms on a single distributed network. This means that every device can exchange information with each other, allowing more flexible, efficient, and autonomous communication networks [2].

## 1.1 Problem Statement

Over the past years, there has been a large growth in the use of IoT equipments in the industrial sector. The demand for such devices is motivated by the possibility of increasing the self-sufficiency of not only the process but also the systems in the industrial setting through machine-to-machine connections, with no external intervention [3]. However, the large-scale growth of IoT devices lead networks to become more complex, increasing the probability of congestion of network due to the great concentration of communicating appliances. The reliability of transmission is especially significant when it comes to safety-critical systems. These systems are real-time and have demanding timing requirements. The tremendous demand for IIoT has brought in new protocols, many of them not originally designed for this type of use. One of them, and perhaps the most popular, is MQTT. This protocol is largely used in the industry because of its advantages: which include being lightweight and offering an effective paradigm for many industrial use cases. On the other hand, it lacks real-time properties [4]. The latest version of the protocol, MQTT 5.0, presents a feature that allows the protocol to be extended through the use of “user properties“. With this, this work proposes a system that takes advantage of this feature, expanding the protocol to take advantage of

Software-Defined Networking (SDN), namely OpenFlow. Thus, MQTT's flexibility can be complemented with the quality of service mechanisms available in OpenFlow, allowing devices to dynamically and autonomously reserving the necessary bandwidth for the transmission of messages of given topics.
This dissertation build up on previous work carried out within the Telecommunications and Networking – Av research group of the Instituto de Telecomunicações, which resulted in a MSc dissertation [5] and a conference paper [6], in which some basic mechanism were evaluated and validated.

## 1.2 Objectives

The main objectives of this dissertation are:

- Study of SDN, emphasizing on the OpenFlow protocol;
- Study of the MQTT protocol for industrial environments;
- Implementation of a OpenFlow controller capable of prioritizing critical messages and its subsequent validation;
- Analysis of the results obtained;

## 1.3 Document Outline

This dissertation is organized as follows :

- ***Chapter 2***: starts by introducing SDN and the OpenFlow protocol. Then, Ryu, an SDN framework, is introduced. Lastly, the chapter ends with an overview of the MQTT protocol.
- ***Chapter 3***: describes the architecture of the system.
- ***Chapter 4***: describes the implementation of the controller.
- ***Chapter 5***: describes the experiments conducted to validate the controller and presents an analysis of its results.
- ***Chapter 6***: presents the conclusions of the dissertation and some possible lines for future research.

CHAPTER 2

# Theoretical concepts and protocols

**Contents**

*T*his chapter presents concepts that are fundamental for understanding the various topics discussed throughout this dissertation. It starts by presenting the concept of SDN, followed by introducing the OpenFlow protocol, a SDN protocol. The chapter ends with a description of the MQTT protocol.

## 2.1 Software-Defined Networking

Network devices, such as switches, can have their switching roles abstracted in multiple planes which consist of distinct categories of data that serve different purposes on the network. There are three planes: data, control, and management [7]. Figure 2.1 provides an overview of said planes and the interactions that they have with each other.

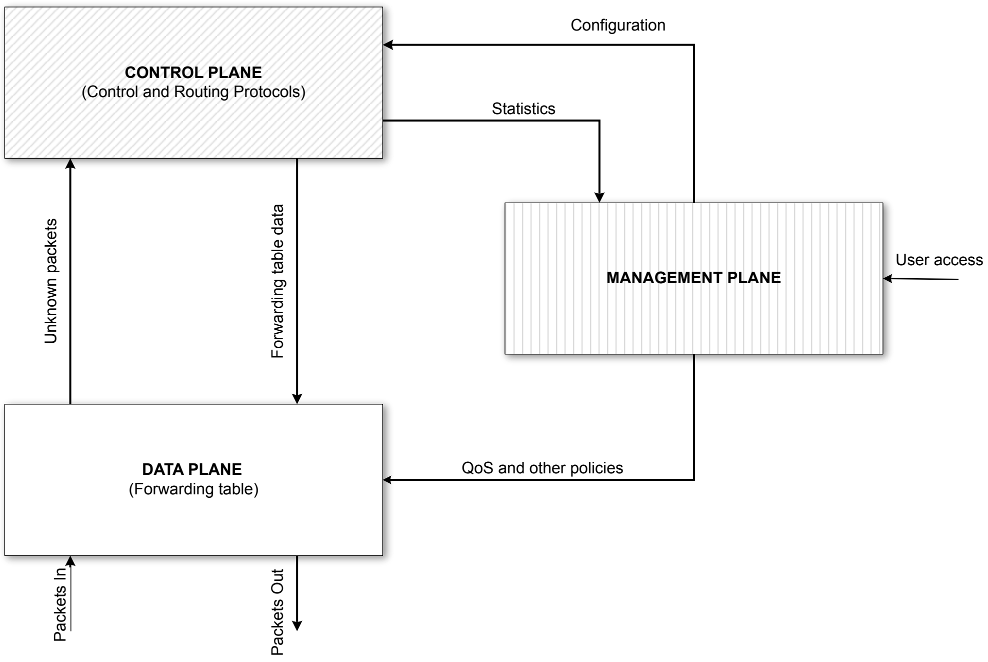

Figure 2.1: Data, control, and management planes.

The data plane is where packets are received and transmitted and where the forwarding rules are instantiated. Thus, this plane is responsible for operations such as forwarding, buffering and scheduling packets, and also modifying headers. The control plane keeps the forwarding table rules, located on the data plane, up to date. The management plane is responsible for configuring the other planes [8].
On traditional network devices, the aforementioned planes exist on every device. Thus, a traditional device has to, besides packet forwarding, do control and other management tasks such as prioritization and filtering. Since the control is done on the device itself, there is no concise method of distributing policy information and other configurations, such as Quality of Service (QoS), with other devices. As the industry progressed, networking devices became increasingly more complex, requiring higher computational capabilities. At the same time, as the management is on a per-device basis, it became more difficult to realize, specially in large-scale networks, due to the high number of devices. In addition, the coexistence of devices from multiple vendors results in different implementations that may not be compatible or lack common features altogether.
That being said, SDN is a paradigm in which the data plane and the control plane are decoupled from each other. Additionally, this architecture specifies that the control plane should be centralized and programmable using an open and standard Application Programming

Interface (API) [8]. The separation of the data and control planes and the centralization of the latter addresses the major limitations of the traditional devices. Foremost, it makes it possible to manage and control multiple devices from a single, centralized and standard software-based system instead of managing each device individually, running its own proprietary software. Additionally, since the control is performed by a separate, yet standard, entity, the compatibility of heterogeneous networks, that is network with devices of multiple vendors, is much higher than the traditional paradigm. Moreover, by offloading the control from the network devices to a controller, the processing overhead is reduced, allowing for much simpler devices. In addition, since the control is centralized, it has a complete overview of the network, allowing for more efficient routing and forward decision-making and superior distribution QoS policies and other configurations. This additional usability brought us protocols such as OpenFlow[9], an implementation of the SDN paradigm, that quickly became an industry standard in this field.

## 2.2 OpenFlow

OpenFlow is a specification by the Open Network Foundation (ONF) [10] for the SDN paradigm. The first version was released in 2009[11], and it has had multiple revisions since then, being the most recent the version 1.5.1, released in 2015[9]. Fundamentally, this specification defines three entities, a switch, a controller, and a protocol, and how they interact with each other. Figure 2.2 illustrates the two basic elements of the specification: the switch and the controller.

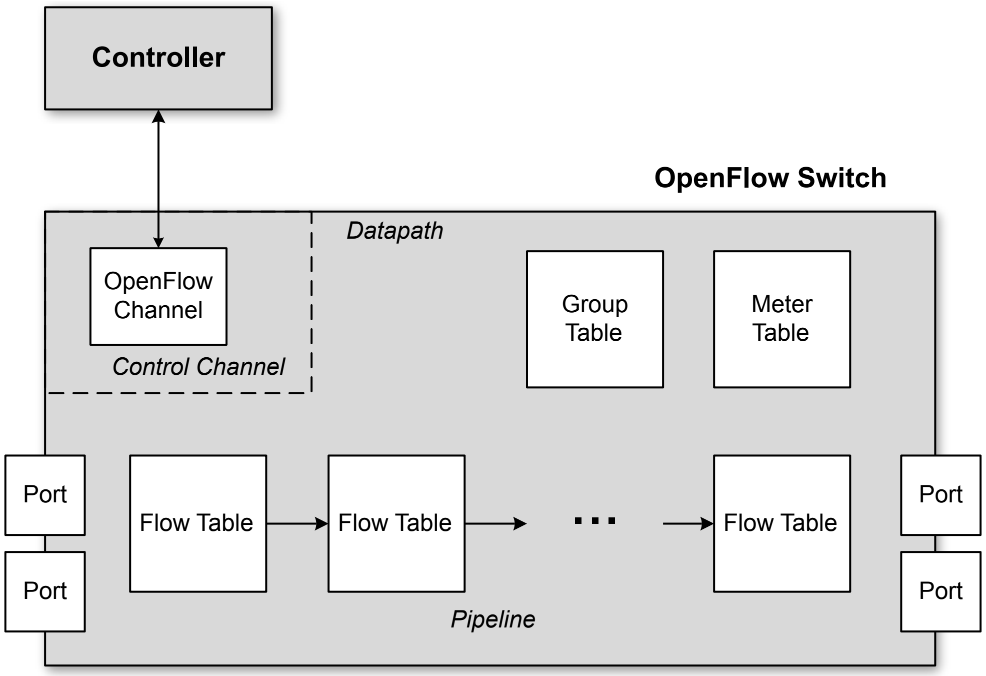

Figure 2.2: OpenFlow switch and controller (adapted from [9]).

The OpenFlow Switch communicates with one or more OpenFlow Controllers using the OpenFlow Protocol [9]. Hence, the decoupling of the control and data planes is attained, where the former resides on the controller and the latter on the switch. However, rather than specifying a strict architecture to be used by each of the previous entities, it describes the mechanisms and the behaviour that each of them should have to implement the specification itself. This ensures compatibility across devices from different vendors.

### 2.2.1 OpenFlow Controller

The OpenFlow Controller is the entity that interacts with one or more OpenFlow switches using the OpenFlow protocol. It is responsible for configuring the matching of packets and forwarding rules in the switch.

#### 2.2.1.1 Ryu controller

Ryu [12] is a component-based software defined networking framework, written in Python. It provides libraries and an API that can be used to create network management and control applications with protocols like OpenFlow [13]. It supports all releases of OpenFlow through version 1.5 [13]. Figure 2.3 depicts the architecture of the framework.

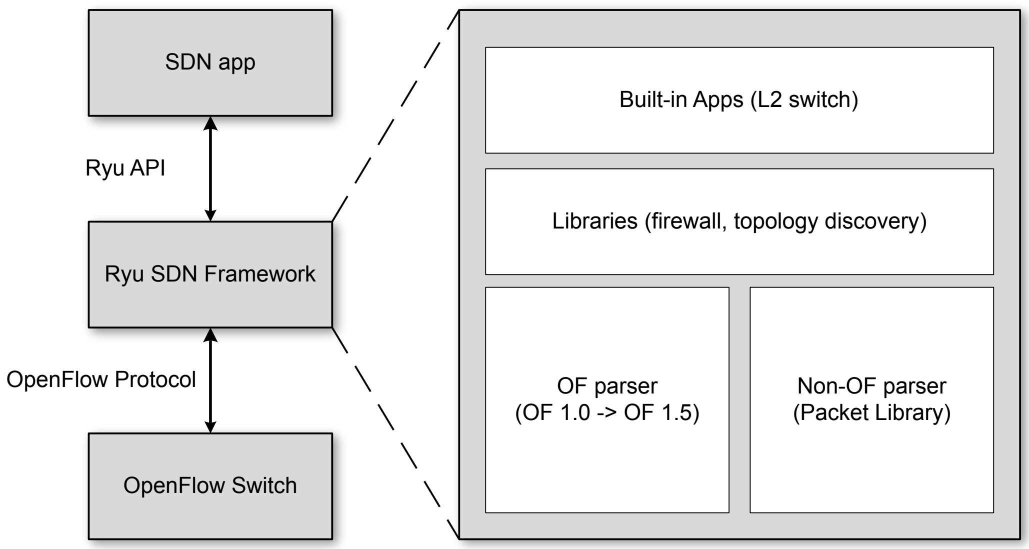

Figure 2.3: Ryu SDN Framework architecture (based on [14]).

To manage a switch it is necessary to write a specific application to implement any desired functionality. The application communicates with the Ryu controller, which will then create, send to, and listen for OpenFlow messages between it and the OpenFlow switch [14]. However, this framework also provides some fully functional SDN applications out-of-the-box, such as OpenFlow-compliant controllers that mimic the behaviour of a standard L2 switch.

Additionally, Ryu also contains libraries that can be used to handle network traffic and support for non-OpenFlow protocols. One such example is the *Packet Library* [15] which provides a set of APIs that can be used to build or parse network packets.

### 2.2.2 OpenFlow Switch

An OpenFlow switch consists of one or more flow tables, a group table, and a meter table, which are responsible for packet forwarding and processing — the **OpenFlow pipeline** [9]. The components of the switch are depicted in Figure 2.2. It is possible for an OpenFlow Switch to be OpenFlow-only, and OpenFlow-hybrid. OpenFlow-only switches only support OpenFlow, whereas OpenFlow-hybrid support both OpenFlow and traditional Ethernet switching. Additionally, in OpenFlow, ports can be physical, logical, or reserved. A logical port is a switch port that does not correspond directly to a hardware's physical port, e.g a Link Aggregation Group (LAG). A reserve port is a specific OpenFlow port used to perform certain actions, such as sending packets to the controller [9]. Any of these ports can receive or forward packets. A port that receives a packet is known as an ingress port and a port that forwards out a packet is an egress port [9].

#### 2.2.2.1 Pipeline processing

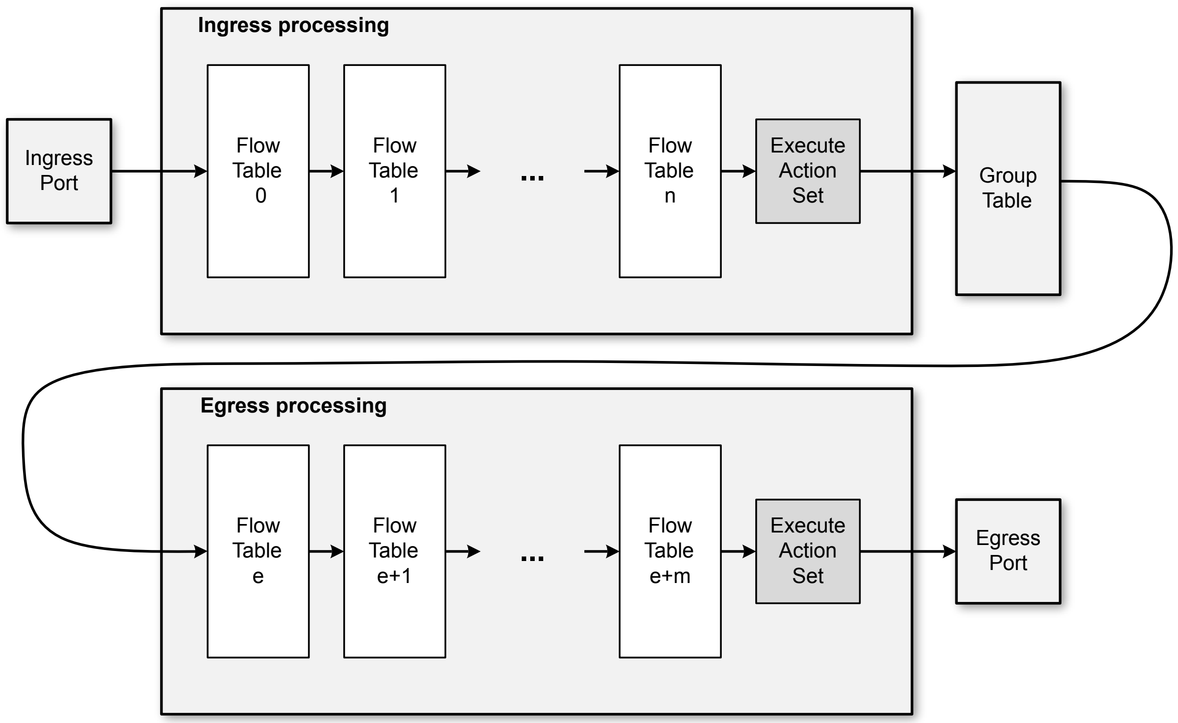

Figure 2.4: Overview of the processing pipeline (adapted from [9]).

The pipeline processing is a defining feature of OpenFlow. If the switch is OpenFlow-only, then it will use exclusively the OpenFlow pipeline for processing, but if it is a OpenFlow-hybrid switch it might do processing outside the OpenFlow pipeline [9]. In any case, the OpenFlow pipeline processing, as outlined by Figure 2.4, defines how packets interact with flow tables. The flow tables are numbered, starting at 0, and packets are processed in them. There may be multiple flow tables, but they must always have an increasing number, as forwarding can only be done to higher number tables. In other words, a packet cannot go backwards in the pipeline.

The processing happens in two stages: ingress processing and, optionally, egress processing. Ingress processing is done on packets arriving at an ingress port. Those packets will be matched against flow entries of the first ingress flow table [9]. Depending on the matching and configuration, the packet may continue to another flow table for further processing. If no matching occurs, the packet will be forwarded to an egress port. This marks the end of the first stage and the beginning of the second stage, the egress processing. In this stage, the packet will be processed similarly to the first stage, but it is optional and may not be supported on all devices. If it is not supported by the device or if it is not configured, the packet will be forwarded to the egress port. However, if it is configured, and supported by the device, the packet will be matched against the entries on the first egress flow table, and continue to the others depending on the outcome of the matching.

#### 2.2.2.2 Flow Table and Flow Entries

A flow table contains flow entries [9], and each flow entry consists of the headers found in Table 2.1.

A packet processed by a flow table is matched against the flow entries of the flow table. The matching is based on the precedence of the flow entry, meaning that the packet will be matched against higher priority flow entries first. Thus, the precedence and the matching fields are the identifying components of a particular flow entry. A flow entry also contains other components such as counters, instructions, timeouts, cookies, and flags:

- **Counters**: counts packets matching a flow entry
- **Instructions**: used to modify the action set or pipeline processing.
- **Timeouts**: maximum amount or idle time for the flow entry to expire.
- **Cookies**: an identifier used by the controller for filtering flow entry.
- **Flags**: used to change how flow entries are managed.

If a matching flow entry is found, the instructions associated with it are executed and the counter is updated. An instruction can be of the following types:

| | | | | | | |
|---|---|---|---|---|---|---|
| Match fields | Priority | Counters | Instructions | Timeouts | Cookie | Flags |

Table 2.1: Main components of a flow entry.

- **Meter**: limits the packet to a specified rate limit;
- **Apply-Action**: applies all actions of the action list;
- **Clear-Actions**: clears all actions of the action set;
- **Write-Metadata**: modifies all actions of the action set;
- **Goto-Table**: the next flow table of the pipeline;

If no match is found in a flow table, the outcome depends on the configuration of the Table-miss flow entry. The Table-miss flow entry is the last entry, with the lowest priority. It can be configured to forward the packet to the controller, to another flow table or to drop the packet.

#### 2.2.2.3 Meter Table and Meters

A meter table contains meter entries[9]. Meter entries are per-flow meters that allow OpenFlow to implement QoS operations, namely rate-limiting and DiffServ Code Point (DSCP) based metering. Rate-limiting is a simple bandwidth limiter that is applied for a set of flows. DSCP is a means of classifying and managing packets based on its rate.

| Meter Identifier | Meter Bands | Counters |
|---|---|---|

Table 2.2: Headers of a meter entry.

Table 2.2 shows the main components of the meter entry.

- **Meter Identifier**: identifier of the meter
- **Meter Bands**: a list of meter bands, where each meter band specifies its respective band and type of process to use on the packet (rate-limiting or DSCP). Table 2.3 contains the main components of the meter band.
- **Counters**: counts of packets processed by the meter

A flow entry can use multiple meters provided that is supported by the switch. Additionally, a packet can also go through multiple meters when using meters in successive flow tables. Each meter must have at least one band. The bands, whose components are depicted in Table 2.3, define the behaviour that meters have on packets. Each meter band specifies a target rate for that band and the way for packets to be processed.

| Band Type | Rate | Burst | Counters | Type specific arguments |
|---|---|---|---|---|

Table 2.3: Headers of a meter band.

The components of a meter band are as follows:

- **Band Type**: how packets are processed — rate-limiting (drop) or DSCP (dscp remark)
- **Rate**: target rate of the meter band
- **Burst**: granularity of the meter band
- **Counters**: number of packets processed by a meter band
- **Type specific arguments**: optional arguments specific to a type of band.

The support for multiple bands might depend on the switch.

#### 2.2.2.4 Group Table

A group table consists of group entries [9]. A group table facilitates operations that could not otherwise be performed through a flow entry such as multicast or broadcast forwarding. This is done by applying a set of actions to a particular flow entry. Therefore, group entries cannot perform instructions such as redirecting to another flow table.

| Group Identifier | Group Type | Counters | Action Buckets |
|---|---|---|---|

Table 2.4: Headers of a group entry.

Table 2.4 shows the headers of the group entry.

- **Group Identifier**: number that identifies the group.
- **Group Type**: type of the group. *Indirect* to execute the one defined bucket in this group and *All* to execute all buckets in the group
- **Counters**: counts of packets are processed by a group
- **Action Buckets**: a list of action buckets, where each action bucket contains a set of actions to execute and associated parameters band.

### 2.2.3 OpenFlow Protocol

The OpenFlow protocol defines a standard language for communication between an OpenFlow controller and an OpenFlow switch that runs on the OpenFlow Channel as seen in Figure 2.2. The protocol supports three messages types: controller-to-switch, asynchronous, and symmetric.

#### 2.2.3.1 Controller-to-switch

Controller-to-switch messages are initiated by the controller and used to directly manage or inspect the state of the switch[9].

- **Features**: a request to identify the capabilities of a switch

- **Configuration**: to set and query configuration parameters of a switch
- **Modify-State**: to manage state on the switches
- **Read-State**: to collect various statistics from the switch
- **Packet-out**: to send packets out of a port on the switch, and to forward packets received via Packet-in messages
- **Barrier**: to ensure message dependencies have been met, and receive notifications.
- **Role-Request**: to set the role of its OpenFlow channel
- **Asynchronous-Configuration**: to set an additional filter on asynchronous message that it wants to receive on OpenFlow Channel

#### 2.2.3.2 Asynchronous

Asynchronous messages are initiated by the switch and used to update the controller about network events and changes to the switch state[9].

- **Packet-in**: to send a packet to the controller
- **Flow-Removed**:to inform the controller that a flow has been removed
- **Port-status**: to inform the controller of change on a port
- **Role-status**: to inform the controller of a change of its role
- **Controller-Status**: to inform the controller when the status of an OpenFlow channel changes
- **Flow-monitor**: to inform the controller of a change in a flow table

#### 2.2.3.3 Symmetric

Symmetric messages are initiated by either the switch or the controller and sent without solicitation[9].

- **Hello**: messages exchanged between the switch and the controller upon startup
- **Echo**: sent from either switch or controller to verify the status of the connection
- **Error**: used by the switch or the controller to notify problems to the other side of the connection
- **Experimenter**: a standard way for OpenFlow switches to offer additional functionality within the OpenFlow message type space.

## 2.3 MQTT

MQTT is a standard[16] from the Organization for the Advancement of Structured Information Standards (OASIS)[17] that defines a messaging transport protocol that follows a publish/subscribe (also known as pub/sub) paradigm. It is designed to be used in applications that have limited communications and low computational resources, such as IoT and Machine to Machine (M2M)[18]. Therefore, the protocol is lightweight and simple, and thus requiring a small code footprint and network bandwidth. It typically runs over TCP/IP. MQTT v5.0 is the latest version of the protocol[16] and is used in many industries, such as automotive, manufacturing and telecommunications.
The main features of MQTT are as follows[16]:

- One-to-many message distribution and decoupling of applications
- Messaging transport is agnostic to the content of the payload
- Three QoS types: “at most once”, “at least once” and “exactly once”
- Small transport overhead and minimized network traffic
- Mechanisms to notify disconnections
- Client authentication and Transport Layer Security (TLS)-based cryptography

### 2.3.1 Overview of the protocol

MQTT follows a publish/subscribe architecture that is based on two different entities: one or more clients (publishers/subscribers) and a server, known as broker[18].
In this architecture, the clients who send messages (publishers) are decoupled from the ones who receive (subscribers). The decoupling is achieved by utilizing a server, known as broker, that receives and distributes the messages. For this purpose, MQTT uses the concept of topics, which are UTF-8 strings assigned to messages for filtering purposes. Clients can publish (send messages) or subscribe (receive messages) to one or more specific topics. The server uses the topic name contained in the received messages to filter and route them to the clients that are subscribed to that specific topic. The example presented in Figure 2.5 shows three clients (two subscribers and one publisher) and a single broker. Both clients on the right subscribe to topic A and so whenever the client on the left publishes to that topic, both of the subscribers will receive the data.

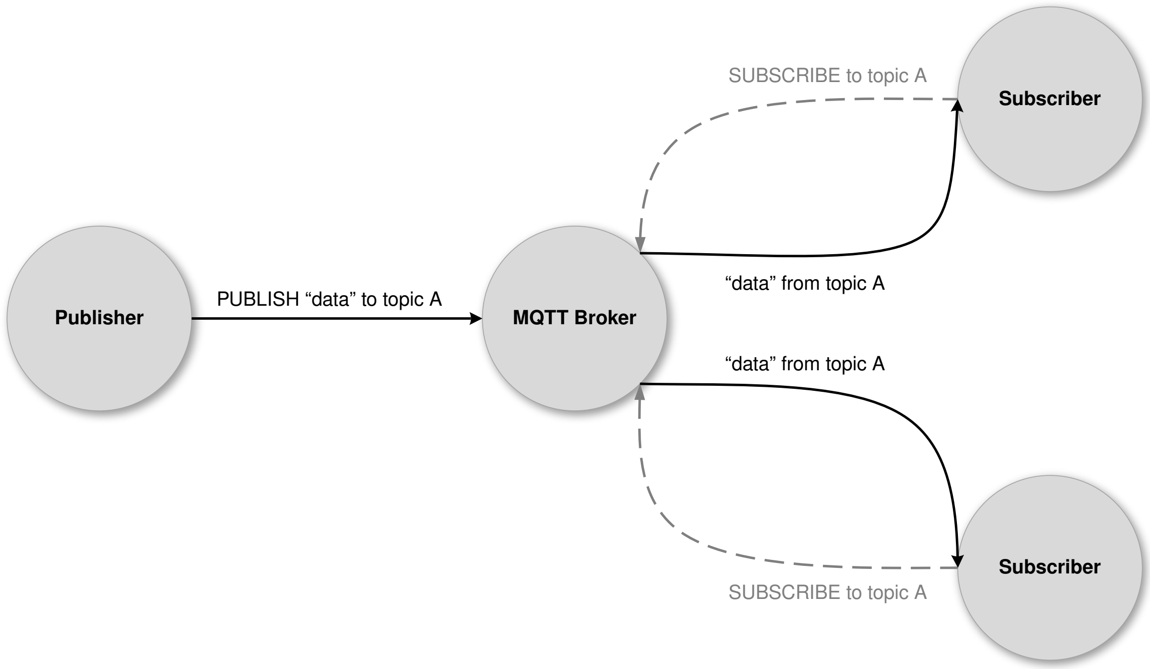

Figure 2.5: MQTT Publish/Subscribe architecture.

### 2.3.2 Publishing and subscribing to topics

The publishing and subscribing of topics in MQTT follows the Pub/Sub paradigm, and as such those transactions are both spacial and temporal decoupled. It is understood as spatial decoupling when clients know that the broker exists, but are not aware of each other. Also, it is said to be temporal decoupled if messages can be transmitted without both the subscribers and publishers being online at the same time [19].

That being said, to receive messages on topics of interest, the client sends a SUBSCRIBE message to the broker detailing a list of subscriptions [20]. Each subscription contains the topic and the desired QoS level. The topic may contain wildcards, allowing to subscribe to a topic pattern than a specific topic. To confirm each subscription, the broker sends a SUBACK message to the client, containing the list of return codes of each subscription. To cancel a subscription, the client can send a UNSUBSCRIBE message, containing a list of topics to unsubscribe. The broker confirms the unsubscription by sending a UNSUBACK message, containing the list of return codes of each cancelled subscription.

To send messages, a client sends a PUBLISH message to the broker. The message must contain the topic name, the payload which contains the data to transmit and other proprieties such as QoS. The broker reads the message, acknowledging it according to the QoS level, and determines which subscribed clients should the message be distributed to. Since clients do not know about each other, the publisher does not get any feedback about which clients received the message.

### 2.3.3 Quality-of-Service

The QoS level is an agreement between the sender of the message and the receiver of a message that specifies the guarantee of delivery level.
As such, the QoS level is set when the client publishes the message to the broker and when the client subscribes to a particular topic. Therefore, if the subscribing client defines a lower QoS level than the publishing client, the broker transmits the message with the lower level of QoS .

#### 2.3.3.1 QoS 0 — “at most once”

QoS 0 is the minimal and default level. It provides the same guarantees as the underlying TCP protocol.

#### 2.3.3.2 QoS 1 — “at least once”

QoS 1 guarantees that a message is delivered at least one time to the receiver. The sender sends a new PUBLISH message with an unused Packet Identifier number and waits for a PUBACK message with the same ID from the receiver. The sender may resend copies after a certain time until receiving a corresponding acknowledgement.

#### 2.3.3.3 QoS 2 — “exactly once”

QoS 2 guarantees that each message is received only once by the intended recipients. It is the safest and the slowest service level. The mechanism is similar to QoS 1, but it sends a four-part handshake between the sender and receiver. The first handshake confirms the reception of the PUBLISH message, acknowledging it with a PUBREC message. Then, after the sender receives the acknowledgment, it sends a PUBREL message signalling that the exchange was successful. Finally, the receiver responds with a PUBCOMP.

### 2.3.4 Message structure

The MQTT messages are known as MQTT Control Packets. These messages share a similar structure composed by a fixed header, variable header and, optionally, a payload.

#### 2.3.4.1 Fixed Header

The fixed header shown in Figure 2.6 exists on every MQTT Control Packet. The first four bits of the header indicate the packet type, e.g., PUBLISH.

| **Bit** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|
| *byte 1* | MQTT Control Packet Type | | | | Flags | | | |
| *byte 2* | Remaining Length | | | | | | | |
| … | | | | | | | | |

Figure 2.6: Structure of the fixed header.

The following four bits indicate flags specific to each MQTT packet type. As of MQTT v5.0 only PUBLISH uses flags. The last field, *Remaining Length*, contains the remaining bytes of the message.

#### 2.3.4.2 Variable Header and Payload

The variable header, as the name suggests, varies depending on the MQTT Control Type and might not even be present at all. If present, the variable header contains proprieties related to the command itself or the payload. The payload is always the last section of the packet.

CHAPTER 3

# Architecture

**Contents**

*T*he MQTT protocol is one of the most popular protocols for M2M and IoT applications, due to its small footprint and simplicity. The popularity of MQTT brought it to a wide variety of industries, such as automotive, manufacturing and telecommunications [17]. Recently, there has also been a growing tendency for the adoption of MQTT in applications in the industrial context, namely for intelligent industrial production environments [21].
Despite having characteristics suitable for IoT applications, MQTT is not suitable for the industrial context. These scenarios impose requirements regarding timeliness and reliability that cannot be guaranteed by the mechanisms of QoS found in MQTT.
One possible approach to solving this problem is to use software defined networks such as OpenFlow. Here, the network nodes would declare the requirements they need to perform the desired communications to another central node, which, in turn, would change the necessary configurations to guarantee the requested network requirements.
The MQTT protocol, as previously mentioned, does not provide mechanisms to guarantee the requirements imposed by industrial applications. However, MQTT v5.0 made possible to extend its functionality through the use of User Properties.
Therefore, this chapter proposes a network architecture based on SDN that overcomes the limitations of the native MQTT protocol. It starts by presenting the new extensions and services added to the MQTT messages that are used by the clients to declared their requirements. The chapter ends with alternative architectures.

## 3.1 Architecture of the network

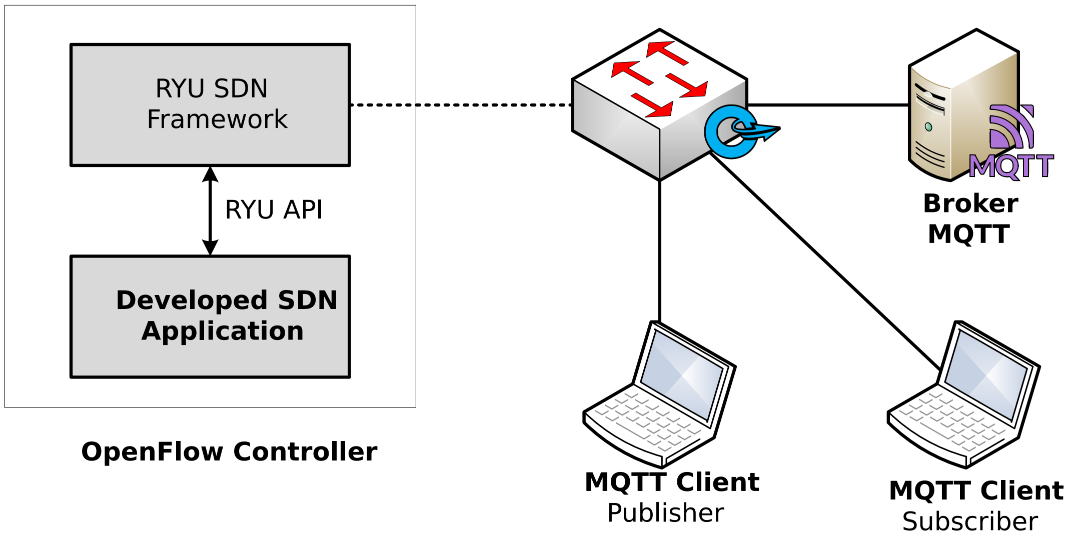

Figure 3.1: Architecture of the proposed network.

In the proposed architecture, shown in Figure 3.1, the MQTT clients and broker are part of a network connected by a SDN switch. The switch is managed by a SDN controller based OpenFlow. On this network, MQTT packets are sent both to the MQTT broker and to the OpenFlow controller. While the default behaviour of the broker remains unchanged, the duplicated packets sent to the controller will be used to make configuration changes to the network. The controller can dynamically add new devices to the network through native mechanisms found on OpenFlow. It runs a custom SDN application, based on RYU SDN Framework, which procedure modifies the necessary forwarding rules and other configurations autonomously.

### 3.1.1 Publishing and subscribing to topics

One of the aspects to consider with this architecture is that the Pub/Sub paradigm of the MQTT protocol remains unchanged. Therefore, clients can continue to produce messages on a given topic and subscribe to receive messages from a given topic. However, MQTT commands now have extensions that allow nodes to transmit properties of the messages assigned to a certain topic, such as the maximum size of a particular message and how often it is sent. These extensions leverage the available optional fields in the MQTT protocol, namely user properties, and thus no modification is made to the protocol specification.

In this proposal, each publisher can independently specify their requirements in any topic. This means that if two publishing nodes were to publish on the same topic, that the resources necessary to ensure the quality of service for that topic would be the cumulative value of both. And so, the OpenFlow controller will allocate the necessary resources to guarantee the requirements requested by all the intervening nodes, including the publishers and subscribers.

### 3.1.2 Handling of network resources

Figure 3.5 presents a flowchart describing the general procedure performed by the OpenFlow controller when admitting MQTT packets.

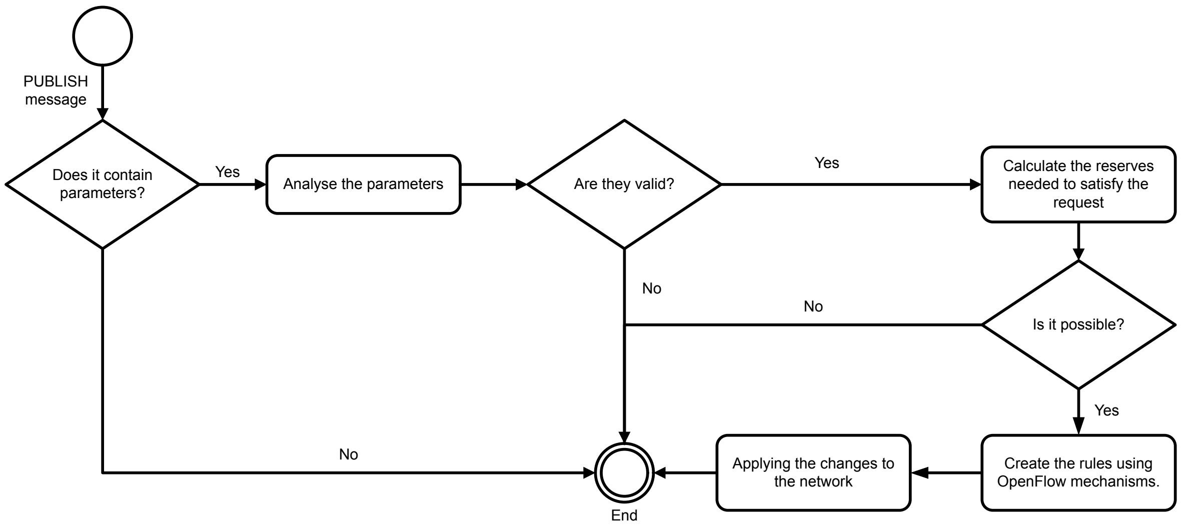
PUBLISH message
Does it contain parameters?
Yes
Analyse the parameters
Are they valid?
Yes
Calculate the reserves needed to satisfy the request
No
No
Is it possible?
Yes
No
Applying the changes to the network
Create the rules using OpenFlow mechanisms.
End

Figure 3.2: Overview of the procedure for accepting and processing parameters in MQTT messages by the OpenFlow controller.

When a client sends a PUBLISH message with a property containing bandwidth requirements, this message is sent simultaneously to the OpenFlow controller and to the MQTT broker. When the controller receives the PUBLISH message, it checks if it contains any User Properties. If it does, the controller validates those parameters and if valid, it computes the necessary reservation amount to fulfil the request. To prevent overloads, the controller keeps track of the bandwidth allocated in each link, rejecting reservations that exceed the corresponding link capacity. To this end, several factors are considered, such as:

- Past requests from the current publisher for that particular topic;
- Past requests from other publishers for that particular topic;
- The number of subscribers that are associated with that particular topic
- All other previous calculations of active reserves from other topics;

Taking all these factors into account, the controller calculates the amount of bandwidth required to satisfy all requests. After reaching a value, the controller proceeds to generate the appropriate OpenFlow commands to reconfigure all relevant QoS mechanisms, such as meters, and other configuration rules. Finally, it applies the changes to the network by sending these commands to the relevant switches, confirming the request(s). If it has no parameters or if it has invalid parameters, the reserve request is ignored.
By default, the following QoS mechanisms are available:

- Priority queues;

- Bandwidth limiting through the use of configurable meters;

The priority queues are used to prioritize MQTT over other types of traffic. Additionally, meters are used to make bandwidth limitations over non-MQTT traffic. However, the use of these rules needs prior identification of MQTT-related flows. For this, it is necessary to previously configure the OpenFlow controller in order to identify the MQTT messages. For this purpose, the OpenFlow protocol has several filters that can be used to make a match. Thus, the following matching fields were used:

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

### 3.1.3 Extension added to MQTT

The extensions developed used the “User Property” which is available in some messages in version 5.0 of the MQTT protocol. Using this header, it is possible to extend the protocol without making any changes to the protocol header itself. Figure 3.3 contains the representation of the existing headers of a PUBLISH message of the MQTT protocol.

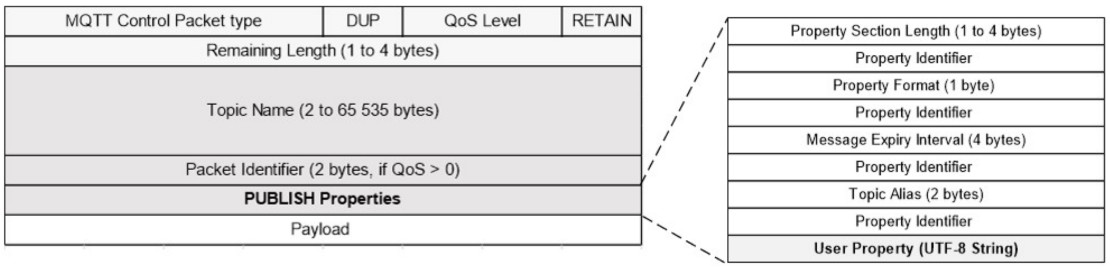
MQTT Control Packet type
DUP
QoS Level
RETAIN
Remaining Length (1 to 4 bytes)
Topic Name (2 to 65 535 bytes)
Packet Identifier (2 bytes, if QoS > 0)
PUBLISH Properties
Payload
Property Section Length (1 to 4 bytes)
Property Identifier
Property Format (1 byte)
Property Identifier
Message Expiry Interval (4 bytes)
Property Identifier
Topic Alias (2 bytes)
Property Identifier
User Property (UTF-8 String)

Figure 3.3: Headers present on a PUBLISH MQTT message.

It was added to the protocol due to the need to transfer metadata whose meaning and interpretation are known only to the entities responsible for sending and receiving it. Table 3.1 summarizes the available extensions.

| Name | Value | Description |
|---|---|---|
| "MSG-SZ" | "DDDDD" | Maximum message size, in bytes. |
| "MSG-PR" | "DDDDD" | Minimum time between consecutive transmissions, in milliseconds. |

Table 3.1: Available user property extensions for PUBLISH messages

The User Property consists of a key-value pair in UTF-8 string format. It is expected to send two pairs per message: “MSG-SZ” and “MSG-PR”. “MSG-SZ” is the maximum expected size of the message being sent, in bytes, and “MSG-PR” is the minimum time between each message, in milliseconds. Their corresponding value consists of an unsigned integer formatted as an UTF-8 string.

### 3.1.4 General message sequence

As previously mentioned, the switch's ability to duplicate the packet to be analysed by the controller is taken into play. In this manner, the publisher client sends a message to the broker and the switch duplicates it by sending it to both the broker and the controller. It should be noted that, from the perspective of the MQTT communications, nothing is changed, except for the fact that the client inserts the two parameters in the User Property header. These parameters target the controller and not the broker. However, since the parameters are part of the MQTT protocol they will propagate to the other clients. Nevertheless, the parameters reach the controller through the mentioned duplication and forwarding done by the switch. Figure 3.4 presents the general sequence of the proposed architecture.

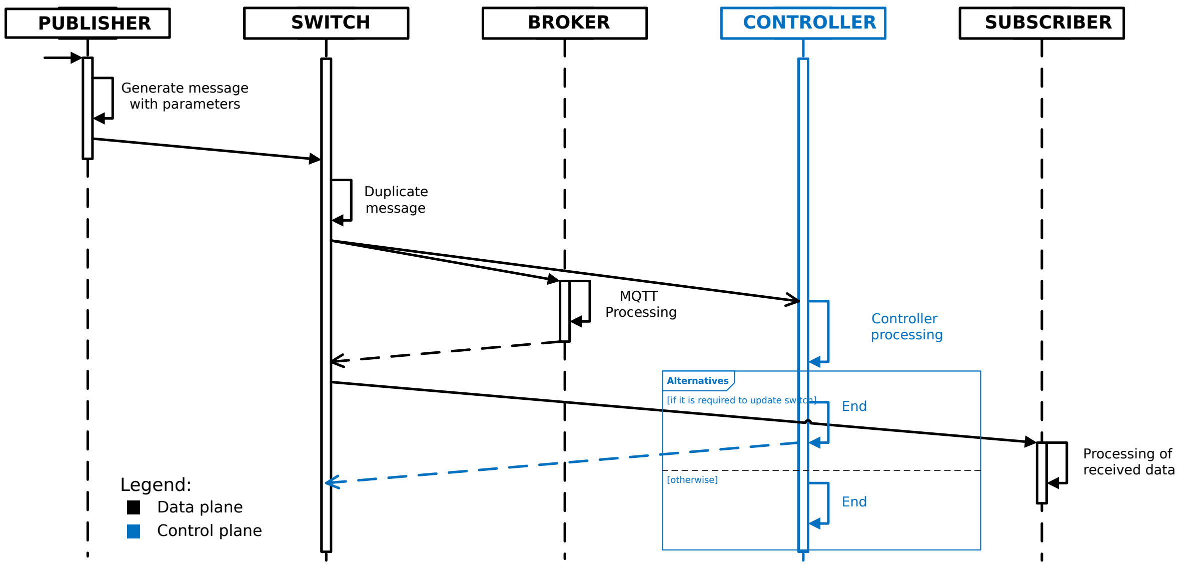

Figure 3.4: General message sequence.

The sequence is initiated by the MQTT client publisher. This client sends a PUBLISH message whose header "User Property" contains the two implemented parameters. To reiterate, these parameters contain the value of the message size as well as its periodicity. That message is destined for the MQTT broker, but first has to be forwarded by the OpenFlow switch. That switch detects, through OpenFlow mechanisms, that the received packet is an MQTT packet and as such duplicates that packet and sends it to the broker (data plane) and to the controller (control plane). From the point of view of the data plane the path taken by the MQTT packet is sent in the usual way, with the differences imposed by this architecture being present only in the control plane. The duplicated packet is sent to the controller. In the controller the packet is parsed and analysed, and the PUBLISH message is extracted, namely the User Properties header which contains the parameters whose values are used by the controller. Through these parameters, the controller will make the necessary reservations to ensure the transmission of MQTT packets. If there is a need to activate or adapt the mechanisms, the controller updates the switch table with the new rules.

## 3.2 Limitations and possible workarounds

In the architecture described beforehand, all MQTT flows are sent to the OpenFlow controller. This is because MQTT packets contain metadata relevant for quality of service control of the network, and so it is necessary to read it.

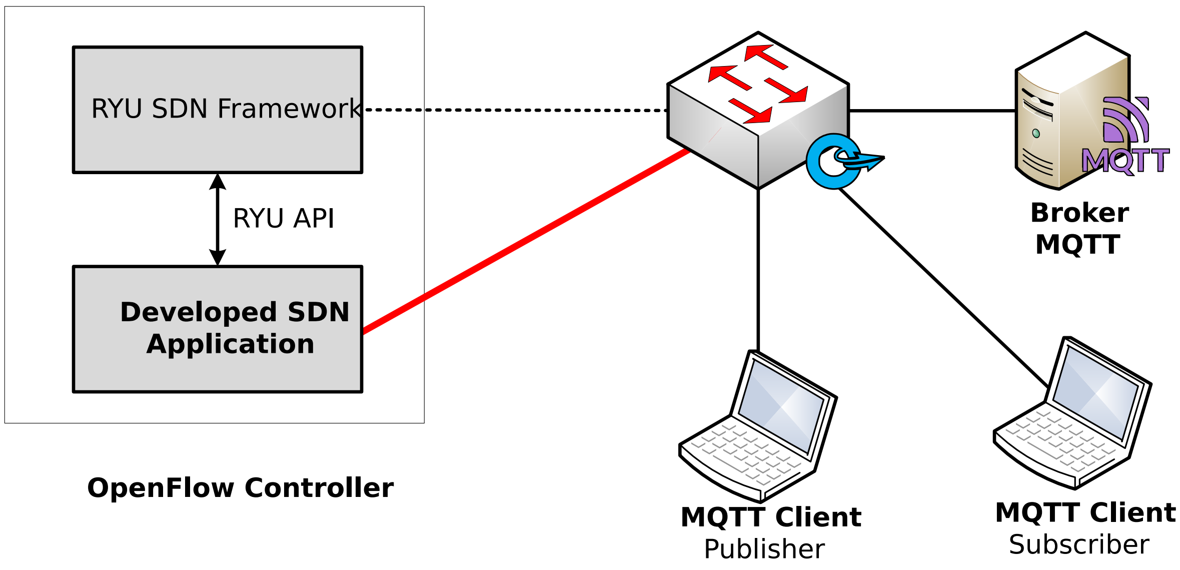

Figure 3.5: Implemented workaround.

However, continuously sending packets to the controller is not ideal, as it adds unnecessary delay to the processing of those packets. As a possible solution, we can take advantage of OpenFlow's duplication capabilities and duplicate the packet forwarding to the broker and the controller. This way, the extra delay is nullified, since it is not necessary to wait for its analysis and processing.
Nevertheless, the duplication of packets in OpenFlow can be done through the use of Group Tables. Group Tables are defined for in the specification, but implementation may vary depending on the vendor. Thus, it is possible that some hardware may not fully support this functionality, which was the case with the Edgecore AS4610-54T [22] switch used in this work. The ASIC present in it does not support replicating the packet and then operating by the actions in each bucket [23]. Thus, this would mean that all buckets would have to be the same, but, in OpenFlow, the forwarding of packets to a port and to the controller are different actions.
So to overcome all the limitations presented, the end solution was a sampling-based system where only a few MQTT packets are sent to the controller at the same time. By doing this, the controller does not have to continuously process MQTT packets. As the clients should not send the metadata in every packet, an acknowledgment mechanism was used where the controller confirms the execution of the clients' requests. The acknowledgement mechanism is described in the following chapter.

CHAPTER 4

# Implementing the system

**Contents**

*T*his chapter addresses the development of several components of the proposed system. It starts by listing all the tools used during development, followed by an overview of the network used in this implementation. The chapter closes with a description of the various components of the system.

## 4.1 Overview of the test network

The network developed for testing is based on the one presented in Chapter 2 and is therefore an OpenFlow-based SDN network. Figure 5.1 shows the aforementioned network and all its components. The network consists of six fundamental entities: the switch, the controller, the master, the MQTT clients, the MQTT broker and traffic generating entities.

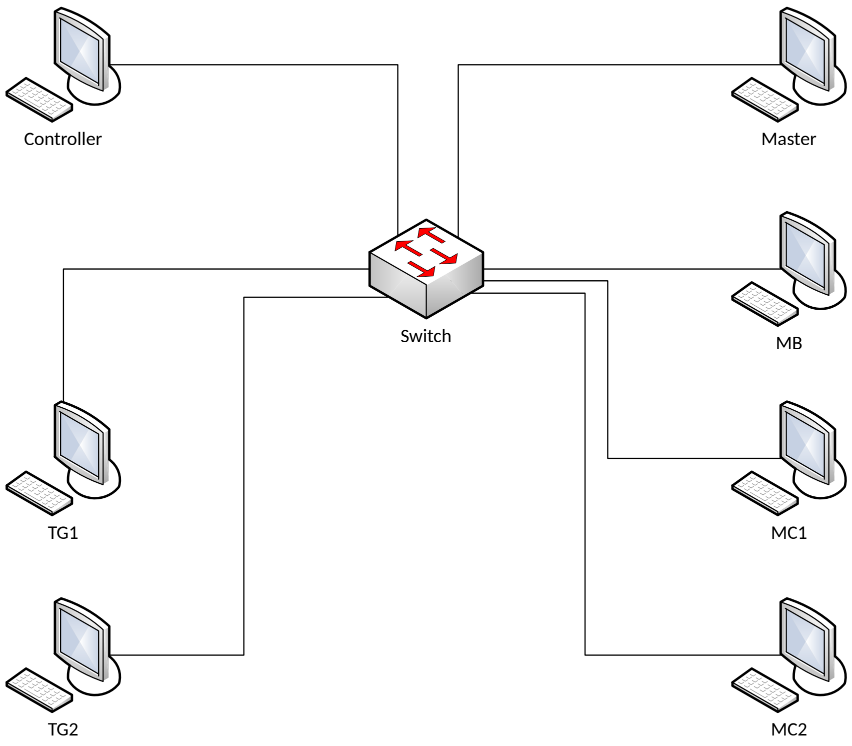

Figure 4.1: Overview of the network used for testing.

The role of each entity is as follows:

- **Switch**: an OpenFlow-based SDN switch responsible for ensuring communication between all other components. It also has to guarantee enough bandwidth for MQTT packets.
- **Controller**: an OpenFlow-based SDN controller responsible for managing the switch
- **Master**: A computer that monitors the network and the devices.
- **MQTT Broker** (MB): Receives messages published by clients and distributes them to subscribed clients.
- **MQTT Clients** (MC1, MC2): Devices that can subscribe and/or publish to MQTT topics.
- **Traffic Generators** (TG1, TG2): Computers that are responsible for generating traffic and send it to a specific port load the network.

## 4.2 Software tools and system components

The system described in the previous section used a variety software and hardware.
A list of the main software tools can be found in the Table 4.1.

| Tool | Description |
|---|---|
| Ryu | A SDN framework that supports OpenFlow. |
| Wireshark | A network protocol analyser. |
| iPerf3 | A network bandwidth testing tool. |
| ethtool | A Linux tool used to control the network device driver and the hardware settings. |
| Paho MQTT | A MQTT client library. |
| Mosquitto | A MQTT broker client. |
| screen | A terminal multiplexer that supports serial and SSH connections. |
| Python 3 | Main programming language. |

Table 4.1: Summary of the tools used for this work.

As discussed in chapter 2, Ryu is a framework for the SDN paradigm. Among many of the features it has available out-of-the-box, Ryu provides tools for developing custom OpenFlow controllers. Thus, this framework was used for this purpose. Another important software was the Paho library. This library, written in Python 3, was used to develop the logic for the MQTT clients (publishers and subscribers). However, unlike the clients, the MQTT broker used a standard implementation of the broker, Mosquitto. The remaining pieces of software were used for debugging and testing processes of the system.
Likewise, Table 4.2 lists all the hardware.

| Component/Label | Hardware (Operating System) |
|---|---|
| Switch | Edgecore AS4610-54T (Pica8 4.1.3) |
| Controller and Master | Personal computer (Ubuntu 20.04 LTS) |
| MC1 and MC2 | SIMATIC IOT2040 (Yocto Linux 3.1) |
| MB and TG2 | Raspberry Pi 4 (Raspberry Pi OS 5.10) |
| TG1 | Personal computer (Ubuntu 20.04 LTS) |

Table 4.2: Summary of the hardware and respective operating system used for this work.

## 4.3 Setting up the network

To facilitate network testing and avoid throttling of the devices, they were all configured for Fast Ethernet speed in Full-Duplex mode. That is, these devices were configured to limit the speed to 100 Mb/s in Full Duplex mode. For this purpose, the command following command was used: `ethtool –s eth0 speed 100 duplex full autoneg on`.

This command can be used by all devices except the switch, which needs a separate configuration that will be covered in the next section. This command configures the eth0 interface to Fast Ethernet and Full-Duplex using ethtool software.

### 4.3.1 Switch

As can be seen in Table 4.2, the switch used for this work was an Edgecore AS4610-54T[22]. This is a commercial switch with 48 Gigabit Ethernet (GE) ports and runs the PicOS operating system [24]. This operating system supports the OpenFlow protocol and as such one or more controllers can be connected. This switch can be configured in two ways: Command Line Interface (CLI) or Graphical User Interface (GUI). CLI over the serial console connection or GUI over the management Ethernet port, but before using it for the first time, an IP address must be assigned to that port.

Nevertheless, the switch needs to be configured first. The first step is to create a *bridge*. A bridge is essentially a virtual switch that can have ports assigned to it. Therefore, the next step is to assign the desired ports to this bridge. In the case of this setup ports GE–1/1/1 to GE–1/1/5 were assigned. Finally, we associate the OpenFlow controller to that bridge. Listing 4.1 shows the commands needed to do the configuration via the CLI. These commands configure a br0 bridge, ports GE–1/1/1 to GE–1/1/5, and the OpenFlow controller.

```
# add bridge br0
$ ovs-vsctl add-br br0 -- set bridge br0 datapath_type=pica8

# add ports
$ ovs-vsctl add-port br0 ge-1/1/1 vlan_mode=trunk tag=1 -- set Interface ge-1/1/1 type=pica8
$ ovs-vsctl add-port br0 ge-1/1/2 vlan_mode=trunk tag=1 -- set Interface ge-1/1/2 type=pica8
$ ovs-vsctl add-port br0 ge-1/1/3 vlan_mode=trunk tag=1 -- set Interface ge-1/1/3 type=pica8
$ ovs-vsctl add-port br0 ge-1/1/4 vlan_mode=trunk tag=1 -- set Interface ge-1/1/4 type=pica8
$ ovs-vsctl add-port br0 ge-1/1/5 vlan_mode=trunk tag=1 -- set Interface ge-1/1/5 type=pica8

# add OF controller with an IP address of 192.168.1.50, and port number of 6633.
$ ovs-vsctl set-controller br0 tcp:192.168.1.50:6633
```

Listing 4.1: Configuring the switch.

### 4.3.2 Other devices

Other than the switch, no other component requires a special network configuration. However, it is important to make sure that they all are on the same network, for example subnet 192.168.1.1/24 The command found in Listing 4.2 can be used to configure the interface.

```
# set 192.168.1.10 to eth0
$ ifconfig eth0 192.168.1.10 netmask 255.255.255.0 up
```

Listing 4.2: Configuring the remaning devices.

## 4.4 OpenFlow controller

### 4.4.1 Flow table layout

Figure 4.2 represents the tables that the controller creates on the switch.

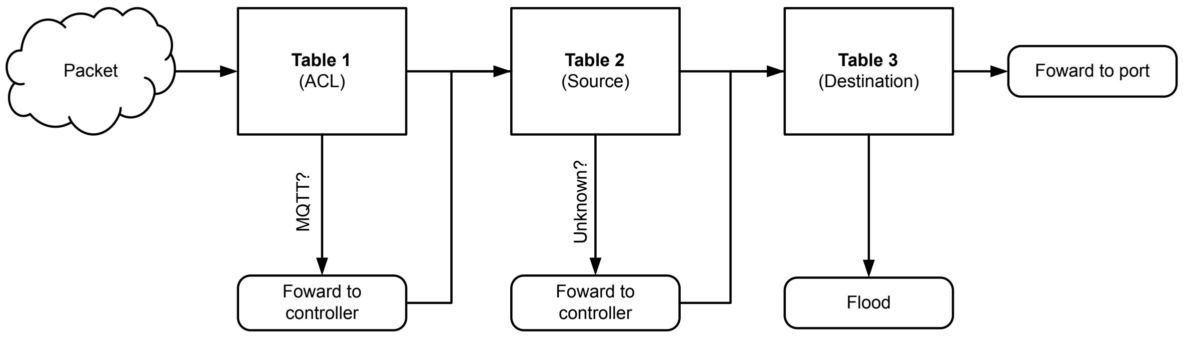

Figure 4.2: High-level overview of the flow tables.

#### 4.4.1.1 Table 1 - ACL

This table contains the flow entries that should be dropped, such as LLDP packets and broadcast sources. Additionally, it contains high-priority flows to identify MQTT packets and forward them to the controller, assigning them to a high-priority queue, and then to the next table. All other packets are forwarded to the next table on a low-priority queue and designated meters.

#### 4.4.1.2 Table 2 - Source

This table is used to keep the state of known devices and their physical ports. If the packet fails to match any of these entries, the table-miss entry instructs the switch to forward it to the controller. If a matching of a known device occurs, it is forwarded to the next table.

#### 4.4.1.3 Table 3 - Destination

This table deals with the final forwarding of a packet. If the destination of the packet is unknown, the table-miss instructs the switch to flood the packet to all ports.

### 4.4.2 Packet Flow

When a new switch is added to the controller, the three aforementioned tables are created. New packets will follow a similar flow to the one presented in Figure 4.4

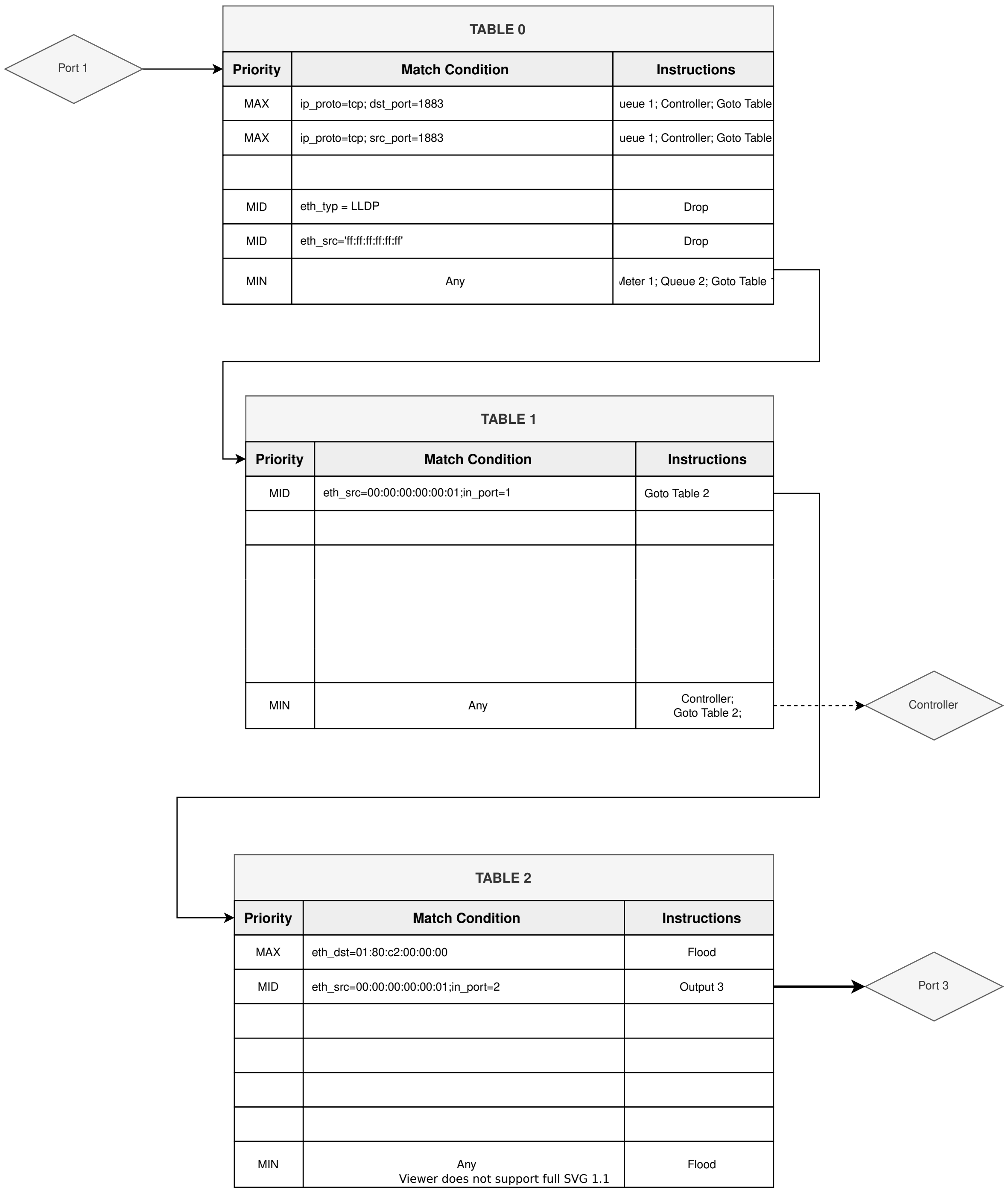

TABLE 0

| Priority | Match Condition | Instructions |
|---|---|---|
| MAX | ip_proto=tcp; dst_port=1883 | ueue 1; Controller; Goto Table |
| MAX | ip_proto=tcp; src_port=1883 | ueue 1; Controller; Goto Table |
| | | |
| MID | eth_typ = LLDP | Drop |
| MID | eth_src='ff:ff:ff:ff:ff:ff' | Drop |
| MIN | Any | leter 1; Queue 2; Goto Table |

TABLE 1

| Priority | Match Condition | Instructions |
|---|---|---|
| MID | eth_src=00:00:00:00:00:01;in_port=1 | Goto Table 2 |
| | | |
| | | |
| MIN | Any | Controller; Goto Table 2; |

TABLE 2

| Priority | Match Condition | Instructions |
|---|---|---|
| MAX | eth_dst=01:80:c2:00:00:00 | Flood |
| MID | eth_src=00:00:00:00:00:01;in_port=2 | Output 3 |
| | | |
| | | |
| | | |
| | | |
| MIN | Any<br>Viewer does not support full SVG 1.1 | Flood |

Figure 4.3: Example of a packet flow through switch.

1. A new packet arrives at port 1. The packet is processed at Table 0. Here, the following cases can happen:

    - If it is a MQTT packet, it will be sent to the controller and then forwarded to Table 1. The flow is assigned a queue of minimum bandwidth of 5 Mb/s. Only the first 255 bytes are sent.
    - If a LLDP or broadcast packet, it will be dropped

- Anything else will trigger a table-miss, that forwards the packet to Table 1, assigning a meter and a maximum bandwidth queue of 95 Mb/s. The meter adjusts of the requested reserves imposed by the MQTT publishers.

2. If packet already exists on the Table, it will be forwarded to Table 3. Otherwise a table-miss will be triggered and forward to the controller to be added to the table. Only the first 255 bytes are sent.

3. The final Table. This is where the final routing is done. If the flow does not exist, the switch performs a flood

### 4.4.3 Bandwidth reservation mecanisms

Through OpenFlow it is possible to use queues or meters to implement QoS mechanisms. OpenFlow queues can make minimum and maximum bandwidth guarantees for egress traffic, which are applied per port. However, they are not managed directly in OpenFlow, being necessary to use another protocol, specific to the switch, to create and modify them. Unlike OpenFlow queues, OpenFlow meters can only guarantee maximum bandwidth for ingress traffic, and these are applied on a per-flow basis. Furthermore, the meters are fully managed in OpenFlow. Therefore, these two mechanisms are not mutually exclusive, but methods that can complement each other.
Figure 4.4 represents in an abstract way what are the mechanisms used by the controller to make bandwidth reservations.

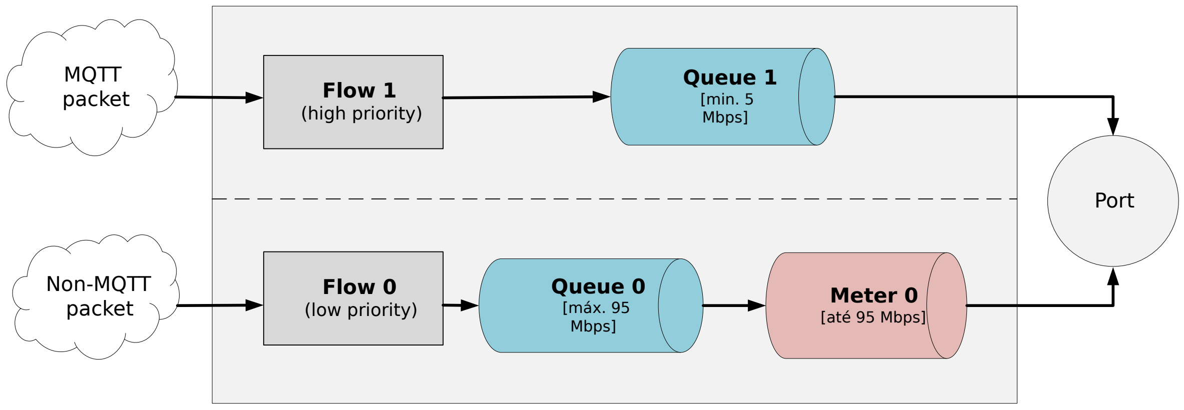

Figure 4.4: Reserve mechanisms implemented.

In one hand, MQTT packets are matched with high priority flows that are then forwarded to queues that guarantee a minimum bandwidth values of 5 Mbps. This value is used to ensure that MQTT communications can be established even in scenarios where congestion might occur. On the other hand, MQTT packets are matched with low priority flows and pass through a queue that guarantees a maximum bandwidth of 95 Mbps. In addition, they also pass through a meter that makes variable bandwidth guarantees up to 95 Mbps. As mentioned before queues cannot be changed natively through OpenFlow and as such meters are used to make dynamic bandwidth adjustments through.

## 4.5 MQTT Clients

The MQTT client is written in Python using the Paho library. The client can function as a publisher as well as a subscriber. The principles of operation of the client are:

- Establish communications with the broker and subscribe or publish for a given topic
- Regarding publishing, it sends messages of a certain size with a fixed predefined period between messages. The size of the messages is increased every minute and ends after 5 minutes. The maximum size of the message and period are added to the user proprieties
- The starting command for both subscribing and publishing is given by the Master

Whenever an MQTT packet is received or sent, one of the device's IO pins is turned on. This is used to accurately measure the timing of sending and receiving without the overhead created by the software stack.

CHAPTER 5

# Validation of the implemented controller

**Contents**

*T*his chapter presents the experiments done to validate the correctness of the system described in Chapter 4. The chapter starts by comparing the data throughput of the proposed SDN system with a conventional L2 switch. The chapter concludes with a comparison of the latency between the developed SDN system and a conventional L2 switch.

## 5.1 Comparing the performance of the SDN switch to a conventional L2 switch

The focus of this chapter is to verify the correct implementation of the system described in chapter 4. For this purpose, two experiments were conducted for two different scenarios. Both experiments are based on the comparison between a conventional switch and an SDN switch, namely the one developed in this work. The goal of these experiments is fundamentally to compare the values of latency and throughput from node to node. Figure 5.1 shows the general network topology used in both experiments, which is very similar to the one presented in previous chapters.

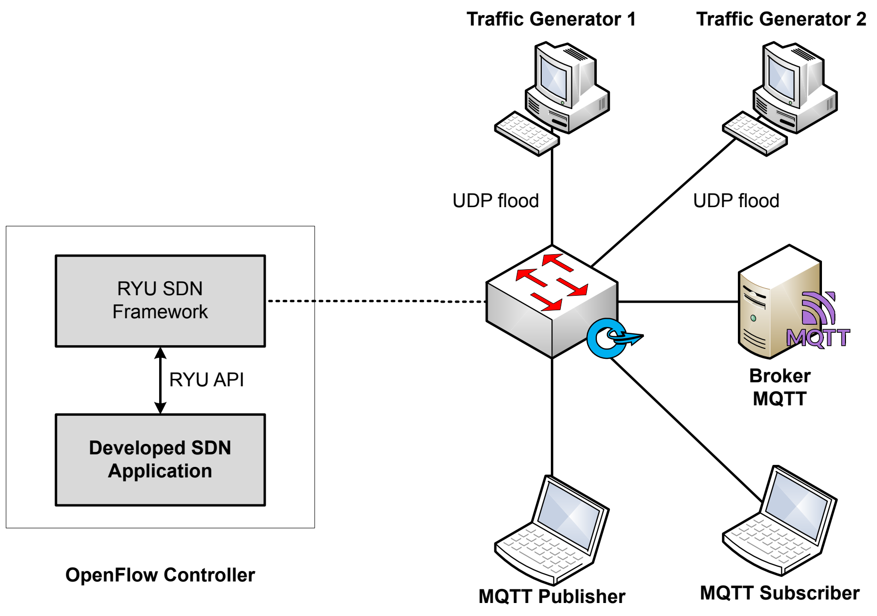

Figure 5.1: Overview of the network used for testing.

In both experiments presented below, the network consists of two MQTT clients (a publisher and a subscriber) and a switch. The publisher client is responsible for publishing to a given topic. He does so periodically during a defined time interval. The aforementioned topic is subscribed to by another MQTT client. Briefly, publishing messages to topics is done as follows:

- The message is sent every 50 ms.
- The publisher sends messages for 5 min.
- Every minute that passes, the size of the message to be transmitted is doubled, having initially a payload of 128 bytes and at the end of 1028 bytes. This is done to simulate variations in the information transmitted.

As said in Chapter 4, MQTT messages contain a header dedicated to user properties. In this work, this header has information regarding the message sent, that is, information such as its size and the periodicity at which it is sent. Represented in Figure 5.1 are two more entities not yet spoken about, traffic generators. These traffic generators are used in only one of the scenarios, and their goal is to try to congest the network through packet flooding.

### 5.1.1 Experiment 1 — Latency and throughput measurements on a balanced network

This experiment intends to compare the network behaviour in a balanced network scenario, that is, in a network that is not overloaded, between several switches, namely the developed SDN switch and a conventional switch. For that, a similar network Figure 5.1 was used, but with the traffic generators inactive. Therefore, the measurements are made in two distinct phases, each one with a different switch. In both cases, the measurements were performed using the IO pins available in the MQTT clients to mitigate the possible delay additions imposed by the equipment network stack. Table and Figure 5.2 and 5.3 show the results for both measurement phases.

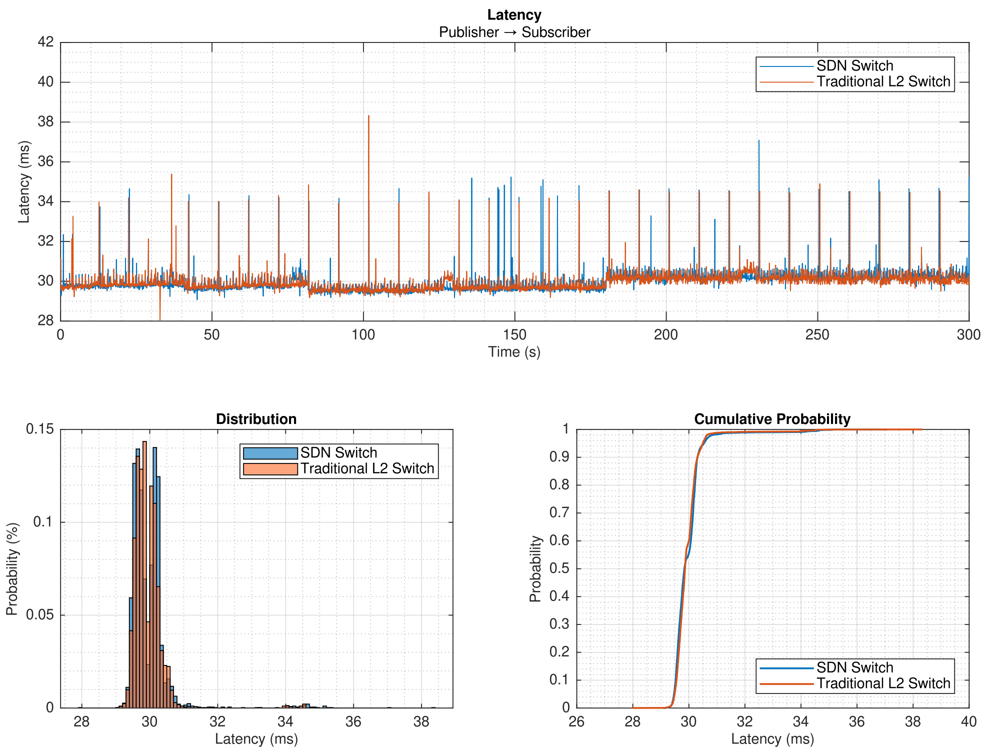

Figure 5.2: Measured latency between the publisher and the subscriber using a conventional L2 switch and a SDN switch on a balanced network.

| | conventional L2 Switch | SDN Switch |
|---|---|---|
| Average latency (ms) | 29.949 | 29.963 |
| Maximum latency (ms) | 38.333 | 37.086 |
| Minimum latency (ms) | 27.990 | 29.071 |
| Standard deviation | 0.518 | 0.585 |

Table 5.1: A summary of the measured latency values on a balanced network.

As can be seen in Table and Figures 5.2 and 5.3, both phases of this scenario show similar results, which would be expected given the nature of these measurements. In this scenario, the network contains only MQTT traffic and is therefore not overloaded, posing no difficulty on either switch. Considering this, we can verify that the developed system does not present any disadvantage with respect to the conventional switch. Note that the peaks shown in the figures are due to the internal behaviour of the switch.

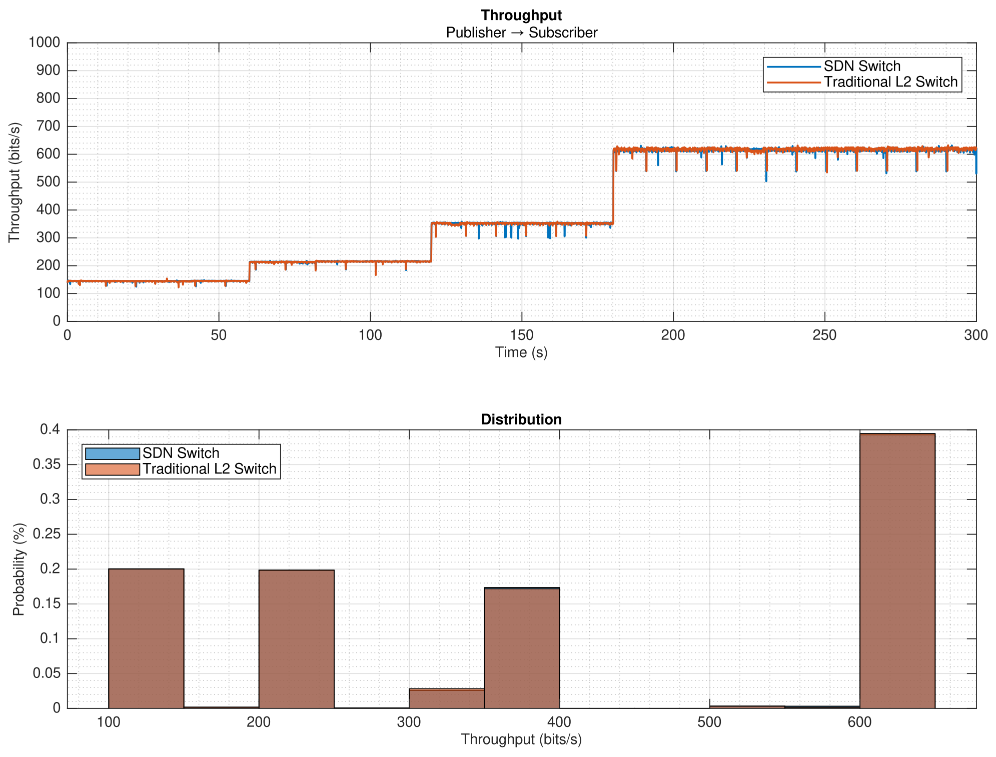

Figure 5.3: Measured data throughput using a conventional L2 switch and a SDN switch on a balanced network.

### 5.1.2 Experiment 2 — Latency and throughput measurements on a loaded network

This experiment intends to compare the network behaviour in a loaded network scenario between the developed SDN switch and a conventional switch. To load the network, the traffic generator devices shown in Figure 5.1 will attempt to saturate the link of the MQTT broker using UDP flooding. Table and Figure 5.4 and 5.5 show the results for both measurement phases.

| | **conventional L2 Switch** | **SDN Switch** |
|---|---|---|
| Average latency (ms) | 37.086 | 30.116 |
| Maximum latency (ms) | 44.264 | 44.599 |
| Minimum latency (ms) | 28.673 | 27.314 |
| Standard deviation | 2.233 | 0.649 |

Table 5.2: A summary of the measured latency values on a loaded network.

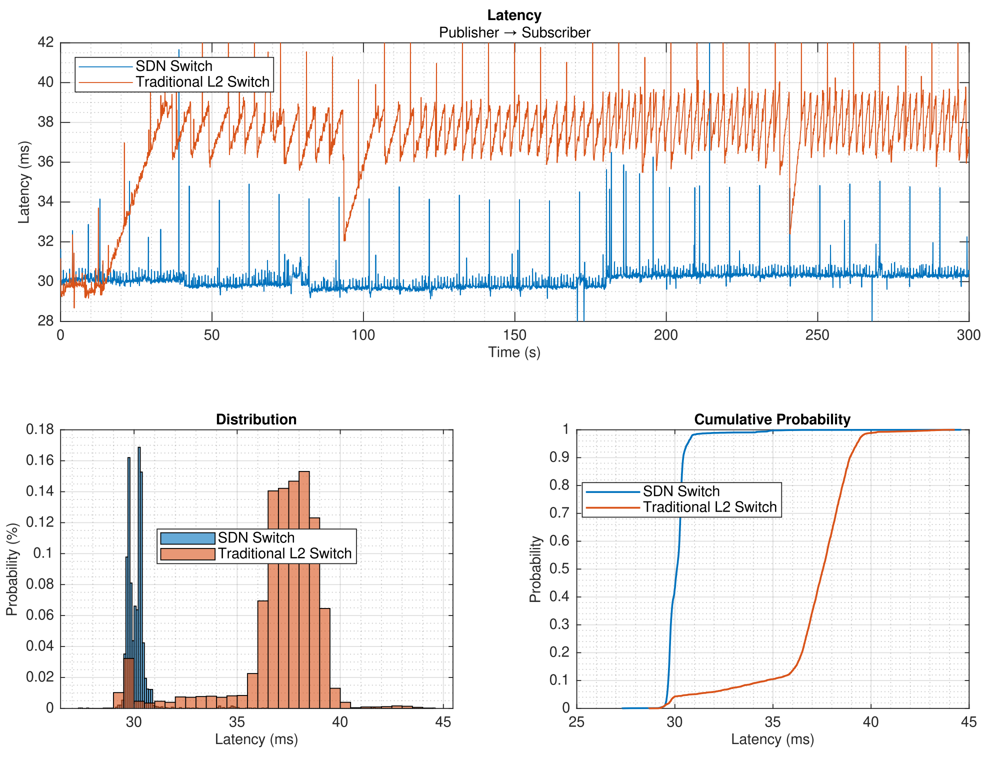

Figure 5.4: Measured latency between the publisher and the subscriber using a conventional L2 switch and a SDN switch on a loaded network.

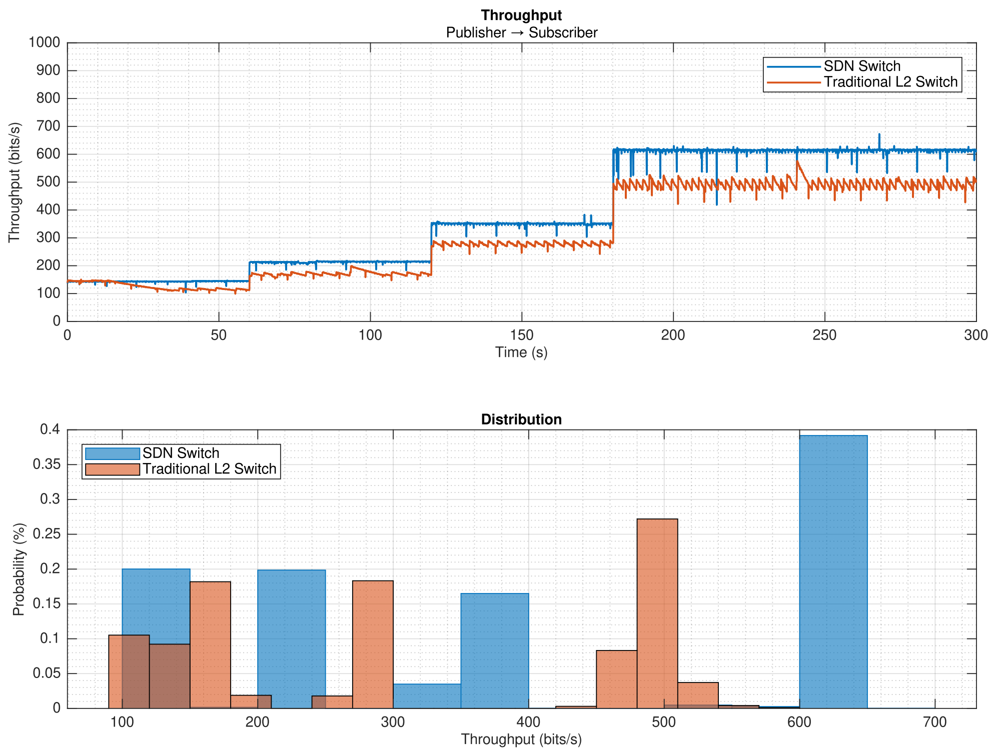

Figure 5.5: Measured data throughput using a conventional L2 switch and a SDN switch on a loaded network.

Through the obtained results it is possible to verify that the SDN switch has a much more stable behaviour than the conventional switch. Besides, the conventional switch has higher latency values as well as lower throughput. This is to be expected since, unlike the SDN switch, the conventional switch does not have the same quality of service mechanisms available. Thus, in the case of the conventional switch, which has no traffic segregation mechanisms, all packets are treated the same way. Thus, MQTT packets have the same priority as other packets in the network. Being congested, the MQTT packets suffer delays as shown in Figure 5.4, reducing the overall throughput as seen in Figure 5.5.

CHAPTER 6

# Closure

**Contents**

## 6.1 Conclusions

The use of IoT and M2M devices is increasing in the industrial context. These devices have enormous flexibility and as such are widely used in industrial applications. However, they use protocols that were not designed to meet such demanding timing and reliability requirements as those found in industry.

Thus, many of the protocols used in this context, such as the MQTT protocol, lack the mechanisms needed to ensure the requirements imposed by communications in an industrial environment. Additionally, this lack of quality of service mechanisms is exacerbated by the fact that networks are typically heterogeneous, that is, they have several protocols simultaneously. The coexistence with other protocols can cause many problems, due for instance to unexpected network congestions.

To overcome this limitation, this work proposed a system that allows these IoT devices to make quality of service changes at the network level. To accomplish this, an extension to the MQTT protocol was developed that allows establishing communications with an OpenFlow controller. Through this extension, an MQTT client can inform the controller of properties related to the transmitted messages, such as the maximum expected size and expected minimum period between messages. With those properties, the controller takes the necessary measures, such as traffic segregation and network level reservation mechanisms, to ensure the quality of service of these transmissions.

The results obtained in chapter 5 demonstrated that the system that was proposed is working as expected. In this chapter comparisons of throughput and latency were made between a network using a traditional L2 switch and an SDN switch using the application developed in this work. These comparisons were made under two scenarios: balanced network and loaded network. In both situations, the developed system demonstrated better performance than the traditional system, being noticeably better than the traditional system in the loaded network scenario, showing higher throughput and less latency.

## 6.2 Future Work

Future works may include:

- Complement the developed system with a MQTT broker supporting real-time applications.
- Mitigate hardware limitations through different methods
- Use of more powerful SDN technologies, such a P4

# References

[1] M. Wollschlaeger, T. Sauter, and J. Jasperneite, "The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0," *IEEE Industrial Electronics Magazine*, vol. 11, no. 1, pp. 17–27, Mar. 2017, ISSN: 19324529. DOI: 10.1109/MIE.2017.2649104. [Online]. Available: http://ieeexplore.ieee.org/document/7883994/.

[2] N. Q. Uy and V. H. Nam, "A comparison of AMQP and MQTT protocols for Internet of Things," in *2019 6th NAFOSTED Conference on Information and Computer Science (NICS)*, IEEE, Dec. 2019, pp. 292–297, ISBN: 978-1-7281-5163-2. DOI: 10.1109/NICS48868.2019.9023812. [Online]. Available: https://ieeexplore.ieee.org/document/9023812/.

[3] B. Kada, A. Alzubairi, and A. Tameem, "Industrial Communication Networks and the Future of Industrial Automation," in *2019 Industrial & Systems Engineering Conference (ISEC)*, IEEE, Jan. 2019, pp. 1–5, ISBN: 978-1-7281-0145-3. DOI: 10.1109/IASEC.2019.8686664. [Online]. Available: https://ieeexplore.ieee.org/document/8686664/.

[4] J.-h. Park, S. Yun, H.-s. Kim, and W.-T. Kim, "Emergent-MQTT over SDN," in *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, vol. 11, IEEE, Oct. 2017, pp. 882–884, ISBN: 978-1-5090-4032-2. DOI: 10.1109/ICTC.2017.8190805. [Online]. Available: http://ieeexplore.ieee.org/document/8190805/.

[5] Francisco Pinto, "Framework for Centralized Technical Management Systems," Ph.D. dissertation, Universidade de Aveiro, Aveiro, 2021. [Online]. Available: http://hdl.handle.net/10773/32282.

[6] E. Shahri, P. Pedreiras, and L. Almeida, "Enhancing MQTT with Real-Time and Reliable Communication Services," in *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*, IEEE, Jul. 2021, pp. 1–6, ISBN: 978-1-7281-4395-8. DOI: 10.1109/INDIN45523.2021.9557514.

[7] K. S. Umadevi, M. S. S. Pranay, and K. Rachana, "Multilevel queue scheduling in software defined networks," in *2017 Innovations in Power and Advanced Computing Technologies, i-PACT 2017*, vol. 2017-Janua, IEEE, Apr. 2017, pp. 1–4, ISBN: 9781509056828. DOI: 10.1109/IPACT.2017.8245144. [Online]. Available: http://ieeexplore.ieee.org/document/8245144/.

[8] R. Sahba, "A Brief Study of Software Defined Networking for Cloud Computing," in *2018 World Automation Congress (WAC)*, vol. 2018-June, IEEE, Jun. 2018, pp. 1–5, ISBN: 978-1-5323-7791-4. DOI: 10.23919/WAC.2018.8430419. [Online]. Available: https://ieeexplore.ieee.org/document/8430419/.

[9] Open Networking Foundation, *OpenFlow Switch Specification Version 1.5.1*, 2015. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[10] ONF, *Open Networking Foundation*, 2018. [Online]. Available: https://www.opennetworking.org/.

[11] Open Networking Foundation, *OpenFlow Switch Specification Version 1.0.0*, 2009. [Online]. Available: https://opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf.

[12] Ryu SDN Framework Community, *Ryu SDN Framework*, 2017. [Online]. Available: https://ryu-sdn.org/.

[13] Nippon Telegraph and Telephone Corporation, *What's Ryu.* [Online]. Available: https://ryu.readthedocs.io/en/latest/getting_started.html#what-s-ryu.

[14] Kei Ohmura, *OpenStack/Quantum SDNbased network virtulization with Ryu*, 2013. [Online]. Available: https://ryu-sdn.org/slides/LinuxConJapan2013.pdf.

[15] Nippon Telegraph and Telephone Corporation, *Packet library.* [Online]. Available: https://ryu.readthedocs.io/en/latest/library_packet.html.

[16] OASIS, "MQTT Version 5.0," Tech. Rep., 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf%20https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

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

[18] B. Mishra and A. Kertesz, "The Use of MQTT in M2M and IoT Systems: A Survey," *IEEE Access*, vol. 8, pp. 201 071–201 086, 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3035849. [Online]. Available: https://ieeexplore.ieee.org/document/9247996/.

[19] S. Quincozes, T. Emilio, and J. Kazienko, "MQTT Protocol: Fundamentals, Tools and Future Directions," *IEEE Latin America Transactions*, vol. 17, no. 09, pp. 1439–1448, Sep. 2019, ISSN: 1548-0992. DOI: 10.1109/TLA.2019.8931137. [Online]. Available: https://ieeexplore.ieee.org/document/8931137/.

[20] HiveMQ, "MQTT & MQTT 5 Essentials," 2020. [Online]. Available: https://www.hivemq.com/mqtt-5/.

[21] P. Torres, R. Dionisio, S. Malhao, L. Neto, R. Ferreira, H. Gouveia, and H. Castro, "Cyber-Physical Production Systems supported by Intelligent Devices (SmartBoxes) for Industrial Processes Digitalization," in *2019 IEEE 5th International forum on Research and Technology for Society and Industry (RTSI)*, IEEE, Sep. 2019, pp. 73–78, ISBN: 978-1-7281-3815-2. DOI: 10.1109/RTSI.2019.8895553. [Online]. Available: https://ieeexplore.ieee.org/document/8895553/.

[22] Edgecore Networks Corporation, *AS4610-54T Datasheet*, 2021.

[23] Pica8 Inc, *Creating a Group Table.* [Online]. Available: https://docs.pica8.com/display/picos292cg/Creating+a+Group+Table.

[24] ——, "PicOS Overview," Tech. Rep. [Online]. Available: https://www.pica8.com/wp-content/uploads/pica8-whitepaper-picos-overview.pdf.