Implementation of MQTT-SN in OMNeT++

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Tayfun Gumus**

Student ID: 945752
Advisor: Prof. Alessandro Redondi
Academic Year: 2022-23

Abstract

In recent years, interest in the Internet of Things has significantly grown, highlighting the need for standards to manage connections within a Wireless Sensor Network. Due to its resource-limited characterization, evaluating effective methods to manage and to optimize communication has been a challenge.
The objective of this thesis is to delve into the MQTT-SN protocol, proposed as a key solution, and to focus on its Quality of Service (QoS) feature assessing its behaviour within a wireless network. By implementing the protocol using the OMNeT++ discrete event simulator with its INET framework, an adequate environment is established, modeling in an efficient way the communication between the devices, supported by an extensive utilization of parameters. The evaluation is conducted using specific performance metrics and introducing intentional errors, based on a model, in order to simulate a near-real-world scenario. The collected results were analyzed to make comparisons between QoS levels. Furthermore, a secondary investigation into the number of packet retries within the protocol is conducted. The reported outcomes show that, QoS choice is directly linked with the application that make use of it, and its requirements. Indeed, a trade-off between responsiveness and reliability can serve as a solution, where one may be preferred over the other despite its disadvantages.

Keywords: MQTT-SN, QoS levels, OMNeT++, INET

Abstract in lingua italiana

Negli ultimi anni l'interesse nella branca Internet of Things è cresciuto in modo significativo, evidenziando la necessità di standard per gestire le connessioni all'interno di una rete di sensori wireless. A causa della sua caratterizzazione di risorse limitate, la valutazione di metodi efficaci per gestire e ottimizzare la comunicazione è stata una sfida.
L'obiettivo di questa tesi è quello di approfondire il protocollo MQTT-SN, proposto come soluzione chiave, e di concentrarsi sulle sue caratteristiche di Qualità del Servizio (QoS) valutando il suo comportamento all'interno di una rete wireless. Implementando il protocollo utilizzando il simulatore di eventi discreti OMNeT++ con il suo framework INET, viene costruito un ambiente adatto, modellando in modo efficiente la comunicazione tra dispositivi, supportato da un ampio uso di parametri. La valutazione viene condotta utilizzando delle metriche di performance specifici e introducendo errori intenzionali, sulla base di un modello, al fine di simulare uno scenario quasi reale. I dati raccolti sono stati analizzati per effettuare confronti tra i livelli di QoS. Inoltre, viene condotto un ulteriore approfondimento sul numero di tentativi di trasmissione per pacchetto all'interno del protocollo. I risultati riportati mostrano che la scelta della QoS è direttamente collegata all'applicazione che la utilizza e ai suoi requisiti. In effetti, un compromesso tra reattività e affidabilità può servire come soluzione, laddove l'uno può essere preferito rispetto all'altro nonostante i suoi svantaggi.

Parole chiave: MQTT-SN, Livelli di QoS, OMNeT++, INET

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**

2 Related Work **3**

3 Methodology **5**

3.1 MQTT-SN 5

3.1.1 Overview 5

3.1.2 Architecture 6

3.1.3 Message Formats 8

3.1.4 Connection and Subscription 10

3.1.5 Quality of Service and Publication 11

3.1.6 MQTT Comparison 16

3.2 OMNeT++ Discrete Event Simulator 17

3.2.1 Introduction 17

3.2.2 Modeling Concept 17

3.2.3 Workflow 19

3.2.4 INET Framework 20

3.3 Implementation 22

3.3.1 Project Overview 22

3.3.2 Network Topology 24

3.3.3 C++ Classes 25

3.3.4 Parameters 26

3.3.5 Validation Approach 27

4 Experiments **29**
4.1 Performance Metrics 29
4.1.1 Average End-to-End Delay 30
4.1.2 Hit Rate 30
4.1.3 Auxiliary Indicators 31
4.2 Error Model 32
4.3 Quality of Service Analysis 34
4.3.1 Parameter Setup 34
4.3.2 Simulation Execution 36
4.3.3 QoS Comparison Results 39
4.3.4 QoS 1 Retries 41

5 Conclusion and Future Work **45**

Bibliography **47**

List of Figures **49**

List of Tables **51**

Acknowledgements **53**

1 | Introduction

In recent years, the field of the Internet of things (IoT) has grown considerably, driven by market demands. More and more electronic devices require integration into a network in order to be better used and controlled easily and quickly. To cope with this trend, the development of new protocols for Wireless Sensor Networks (WSNs) is crucial with the aim of improving the transmission of messages between devices.
Among the various protocols used in this context, two in particular stand out: the Message Queuing Telemetry Transport for Sensor Networks (MQTT-SN) and the Constrained Application Protocol (CoAP). The use of these dedicated protocols, in addition to enabling efficient communication between the various devices in the network, has broadened the IoT application domain. In fact, the advancement of this tendency continues from home automation to industrial processes. The need to communicate effectively between various nodes that have limited resources such as power, memory and bandwidth has led to the development of specialised approaches for an optimal operation. The ability to adapt to changes and the goal of having a stable message flow over a weak and unstable network are key characteristics to be taken into account.
The focus of this document will be on the MQTT-SN protocol, which is proposed as one of the solutions with a key role. It derives from MQTT and shares many of its features, including being a lightweight publish/subscribe messaging protocol. However, the primary aspect is that MQTT-SN is specifically designed to better accommodate the limitations of WSNs, such as computing power, bandwidth and energy. In this scenario, the ability of the protocol to meet high-level requirements, such as those at the application layer, is crucial. To fulfil this, MQTT-SN provides various levels of Quality of Service (QoS) configurations, ensuring reliable message delivery.
This thesis aims to take a closer look at this protocol with a particular attention to QoS and its performance under certain network conditions.
By modelling MQTT-SN accurately using the OMNeT++ network simulator with its INET framework, an adequate simulation environment is established to assess the reliability and the efficiency of the protocol. To achieve this goal, a simplified wireless network is constructed, onto which the implementation of the MQTT-SN protocol is integrated.

Following this integration, specific packet-related metrics are added in the development in order to evaluate the performance of the chosen QoS level under different network conditions. All of this provides the required basis for an adequate analysis of the objectives. As a last but not least step for the continuation of the study, errors are intentionally introduced into the application packets flowing in the network. This strategy is specifically chosen to be able to accurately simulate the different challenges that WSNs generally have in real-world applications. It allows for the replication of realistic error scenarios, evaluating the reliability and efficiency of the protocol and the communication, even under error-prone conditions.

The rest of this document is organized as follows:

Chapter 2 reviews concisely a related work on the two protocols used in the IoT context: MQTT-SN and CoAP.

Chapter 3 provides an overview of the MQTT-SN protocol, focusing specifically on QoS levels, and describes the overall system and the approaches implemented.

Chapter 4 illustrates a comparison of the protocol's QoS levels, supported by experimental data, using metrics operating under different network conditions.

Chapter 5 concludes with an evaluation summary of MQTT-SN from the simulations performed and suggests a direction for practical applications and future work.

2 | Related Work

The importance of developing effective, lightweight and reliable communication protocols for the Internet of Things is increasingly evident. MQTT-SN and CoAP are among these solutions, both supporting a publish/subscribe communication model essential for managing IoT sensor networks and devices with limited resources such as bandwidth, computing power and available energy. A significant study in this field has been conducted [1]. This analysis reports a detailed comparison of the efficiency between MQTT-SN and CoAP, in the Pub/Sub version, particularly aimed at the sensor network and IoT. The work provides critical insights into the potential and limitations of the two protocols across variable network scenarios, conducting an in-depth analysis of their theoretical characteristics and practical performance.

At conceptual level, the analysis highlighted many differences between the two protocols. Connection management, QoS levels, topic structure and fragmentation support are among the aspects that underscore these differences. In particular, CoAP is more efficient in highly dynamic networks and this is due to the connectionless communication model. On the other hand, MQTT-SN requires more structured connection management as it is a protocol designed for scenarios with limited resources. From a practical point of view, the authors proposed an open-source implementation of CoAP Pub/Sub and compared it with MQTT-SN in a simulated environment. The performance of the two protocols under various network conditions were all well-documented by this experiment.

The results indicated that under ideal network conditions, both protocols have similar performance. However, in unstable and highly dynamic networks, CoAP tends to be more advantageous. This feature, which is essential for IoT solutions operating in unreliable networks, results from its quick network adaptability without maintaining the connection with the broker.

Finally, particular attention concerns the application layer, especially the management of QoS levels. The ability to handle them are closely linked to the connection model adopted by the two protocols. With its simplified approach to QoS regulation, using either Confirmable or Non-Confirmable messages, CoAP offers more flexibility in adapting to dynamic network conditions without requiring a constant connection to the broker.

On the contrary, MQTT-SN provides a sophisticated control with the four QoS levels, enabling more accurate message delivery guarantees. However, this granularity comes at the cost of greater complexity in handling connections. The exploration of these differences leads to the conclusion, as mentioned previously, that CoAP is effective in networks characterized by frequent changes, whereas MQTT-SN has potential uses in applications that require maintaining a stable connection.
This conducted study has served as a source of inspiration to investigate Quality of Service in an IoT protocol. In the context of this thesis, the implementation and the analysis of this feature are critical points, highlighting the idea that an efficient QoS management is essential for the development of reliable and resilient IoT systems, taking into account the various challenges presented by the real-world applications.

3 | Methodology

The chapter will discuss the methodology used to build the system in order to carry out simulations and achieve the objectives of the analyses.
Starting from a brief description of MQTT-SN and a quick comparison with MQTT, it will focus on the Quality of Services that the protocol offers.
The second section will describe the tools used by analyzing their characteristics and thus enabling the actual implementation of MQTT-SN.
Finally, the last part will discuss about the structure of the project and the implementation of the protocol by integrating it into the simulation environment. Furthermore, the validation of the implementation and some simulation design considerations will be described.

3.1. MQTT-SN

In this section, the Message Queuing Telemetry Transport for Sensor Network protocol (MQTT-SN) will be described. The specifications used in the thesis are cited from [2].

3.1.1. Overview

In recent times, general interest in Wireless Sensor Networks has grown considerably, offering opportunities to create new systems capable of operating in conditions that challenge their reliability. In particular, it was necessary to implement protocols to enable efficient communication in environments where network and device resources are limited. MQTT-SN was specially designed for this goal within WSNs.
The protocol can be considered a version of MQTT for which it has been adapted and optimized for the implementation on low-cost, battery-operated devices with limited computing power and memory, such as sensors.
Due to its streamlined design compared to MQTT, MQTT-SN offers an effective mechanism for publish/subscribe communications allowing a successful distribution of messages among the various nodes while supporting the scalability of the network topology.
MQTT-SN is highly adaptable and compatible in a variety of settings since it does not

depend on the underlying network and can also be carried over any non-TCP/IP network. Furthermore, being a connection-oriented protocol between devices and available brokers, MQTT-SN is ideal for scenarios with limited resources enabling a reliable communication even in the presence of interruptions or variations in the network.
The protocol also focuses on saving energy by reducing duty cycles and extending the life of the batteries. This characteristic is supported through the use of different device modes that maximize energy efficiency as much as possible.
Each MQTT-SN message is designed to be retransmitted in case of non-delivery due to any type of error between sender and recipient. The retransmission mechanism provides substantial advantages to enable reliable and resilient communication. Moreover, it has a fundamental role in applying different Quality of Service levels that the protocol offers. Through a negotiation process it is possible to configure the desired QoS level for each message, offering flexibility and priority in transmissions.

3.1.2. Architecture

In its general structure, the protocol was built by the Oasis foundation to be flexible and adaptable even to the original version of MQTT, thus requiring an interface. In particular, in order to keep the MQTT broker in the new model, it was necessary to implement an element that acts as a translator of MQTT-SN requests for the MQTT broker. This element is the MQTT-SN gateway (GW).
Depending on how it performs the protocol translation, we can distinguish between two types, namely transparent and aggregating GWs, (see Figure 3.1):

- Transparent gateway: it creates a connection for each client to the broker, serving as a translator of protocols between the two parties
- Aggregate gateway: it establishes only one connection for all the clients, in order to minimize the quantity of linked devices to the broker

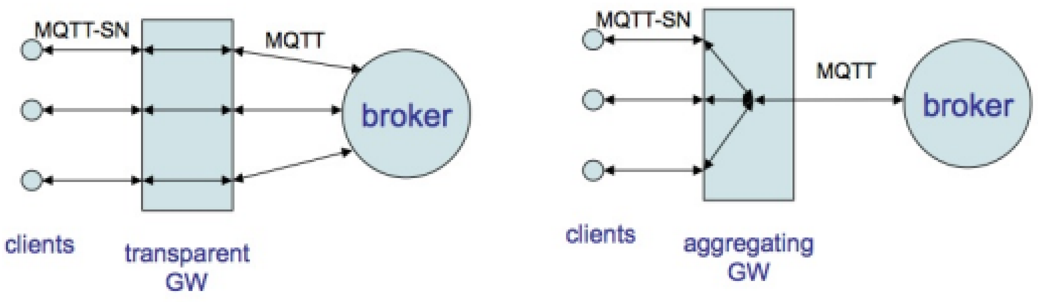


Figure 3.1: Transparent and Aggregating Gateways

Clients can also connect via a MQTT-SN forwarder in case the gateway is not directly attached to their network. This device is able to receive the packets, encapsulate correctly and forward them to the GW. Obviously, it can also deliver the packet from the gateway to the client with a similar procedure. The figure 3.2 shows the complete architecture of the protocol.

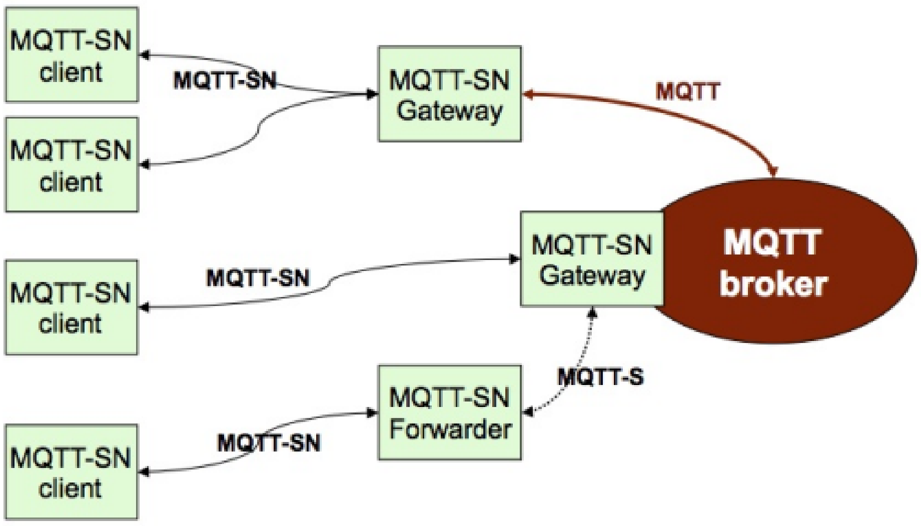


Figure 3.2: MQTT-SN Architecture

In the context of this paper, the architecture is simplified. The approach allows a focus on the aspects of the protocol to evaluate and to integrate easily with the simulation environment.
In particular, it is planned to implement MQTT-SN on top of the TCP/IP stack with UDP at the transport level. The choice of UDP is motivated by its light and fast nature,

with no need to establish connections before transmissions. MQTT-SN fits this choice better, but others are still valid as it is a protocol agnostic to the underlying network. Ultimately, the protocol architecture boils down to these two elements: MQTT-SN Client and MQTT-SN Broker. The latter element is interchangeably called MQTT-SN Server or MQTT-SN Gateway, reflecting the versatility of the role it plays in this context.

3.1.3. Message Formats

The general format of a MQTT-SN message is shown in Table 3.1. A MQTT-SN message consists of two parts: a header and an optional variable part. The header contains essential information about the message in its entirety such as its length and type. The variable part, on the other hand, varies according to the type of message in question.

Message Header	Message Variable Part

Table 3.1: General Message Format

The table 3.2 shows all the message types used in MQTT-SN. Among all these, three in particular are of fundamental importance for the objectives of the protocol: CONNECT, PUBLISH and SUBSCRIBE messages.

ADVERTISE	SEARCHGW
GWINFO	CONNECT
CONNACK	WILLTOPICREQ
WILLTOPIC	WILLMSGREQ
WILLMSG	REGISTER
REGACK	PUBLISH
PUBACK	PUBCOMP
PUBREC	PUBREL
SUBSCRIBE	SUBACK
UNSUBSCRIBE	UNSUBACK
PINGREQ	PINGRESP
DISCONNECT	WILLTOPICUPD
WILLTOPICRESP	WILLMSGUPD
WILLMSGRESP	

Table 3.2: MQTT-SN Message Types

One of the relevant messages is the CONNECT one, which is sent by the client to establish a connection with the server. Its format is shown in the table 3.3.

Length	MsgType	Flags	ProtocolId	Duration	ClientId

Table 3.3: CONNECT Message Format

An overview of the primary fields is provided:

- Flags: it includes some control indicators, the main one being CleanSession. If set, the flag indicates to the server to reset the client's session state upon connection
- Duration: it indicates the maximum time interval (in seconds) within which the client should send a message to the server. If no meaningful message has been sent in this period, then the client should send a PINGREQ so that the server recognizes the client's active state and responds with a PINGRESP
- ClientId: it uniquely identifies the client to the server in the connection session

Another message that has a crucial role in the protocol is PUBLISH. This message is used both by clients, specifically those acting as publishers, and by GWs in order to publish data related to a certain topic. Its structure is shown in the table 3.4.

Length	MsgType	Flags	TopicId	MsgId	Data

Table 3.4: PUBLISH Message Format

The main fields are:

- Flags: it contains several indicators that provide information about the message. A fundamental one is the Quality of Service level for this PUBLISH message
- TopicId: it identifies the topic on which this publication will take place. An important aspect is that the field takes up little space, thus reducing the overhead of the message
- MsgId: it allows the sender to match the message with its corresponding acknowledgment. It is also used in various message types and has a crucial role in QoS management, in particular levels one and two, enabling reliable message delivery
- Data: it carries the payload

Finally, the last message to be explored in detail is SUBSCRIBE used by clients, specifically those acting as subscribers, to subscribe to a specific topic. Its format is illustrated in table 3.5:

Length	**MsgType**	**Flags**	**MsgId**	**Topic**

Table 3.5: SUBSCRIBE Message Format

with fields:

- Flags: it includes indicators that influence the subscriber's subscription behaviour. The main one is the QoS level required for the topic indicated in the field of this message
- MsgId: it identifies the corresponding SUBACK message
- Topic: it specifies the name of the topic to which the subscriber wants to subscribe

3.1.4. Connection and Subscription

One of the key aspects of the protocol is the ability to connect clients to the servers and to maintain it over time. Generally, without a connection, an MQTT-SN client cannot exchange information with the GW.
Each client uses a CONNECT message, in the format discussed earlier, to establish a session, identifying itself with a unique identifier which will be stored on the server.
After previous connection sessions, the client may reset its state using one of the flags in the message field. In particular, the subscriber may request to delete its subscriptions from the broker by using the clean session flag. On the other hand, the publisher may request the deletion of its WILL data by using the will flag.
The response of the server to a CONNECT request is a CONNACK message whose purpose is to inform the client if the connection attempt was successful. The return code field within the response from the gateway contains this information indicating a correct connection or a failure specifying the rejection reason.
Established the connection, the two parties check each other during the session in order to supervise the liveliness of the connection itself, maintaining it over time. The duration field in the CONNECT request is crucial for the purpose setting up a Keep Alive period. As mentioned before, if no meaningful message has been sent in this interval, the client should send a PINGREQ so that the server recognizes the client's active state and responds with a PINGRESP. If the server does not receive any message from the client for a period longer than the Keep Alive duration, it will consider that client as lost and

will take the necessary actions. From the client perspective instead, if it does not receive a PINGRESP from the server even after multiple retransmissions of the PINGREQ message, it should try to reconnect or change gateway.

One of the main operations performed by a client after the connection phase is the subscription. By including the topic in the SUBSCRIBE message, a subscriber sends this request to the broker. If the server is able to accept the subscription, it returns a SUBACK response with the success code. If the request cannot be accepted, a SUBACK message is still returned with rejection cause encoded in the return code field.
As in the previous connection case, but also in general for each unicasted message of the protocol, if the acknowledgement is not received by the client, the original message is retransmitted. Thus, in this case, if the subscriber does not have a SUBACK response, it will retransmit the SUBSCRIBE request again.

3.1.5. Quality of Service and Publication

In this part, the most important concept of the protocol will be discussed.
The Quality of Service is crucial in MQTT-SN. It provides the ability to select a service level that aligns the network reliability with the application's requirements.
As mentioned before, PUBLISH messages are delivered according to the QoS level specified in the corresponding field. Depending on this value, sender and receiver take different actions. The exploration of the QoS levels is done by considering two general parties and observing how the mechanism works. After this simplification, the concept will be applied at overall MQTT-SN system.

QoS level 0 (At most once): At the lowest level, QoS 0 offers a best-effort delivery mechanism where the sender does not expect an acknowledgment or guarantee of message delivery. In particular, after receiving the message, the recipient only processes it, without replying with an ack. On the other hand, the sender does not retry with further retransmissions. QoS 0 is commonly called "fire and forget". The simple interaction is illustrated in the figure 3.3.

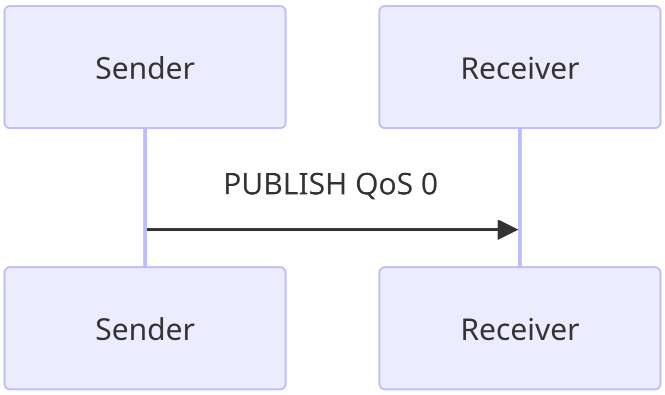


Figure 3.3: Quality of Service 0

QoS level 1 (At least once): The reception of a PUBLISH message with QoS 1 by the receiver is acknowledged by a PUBACK message (PUBlish ACKnowledgment).
In particular, the sender inserts a message identifier within the PUBLISH request before sending it. The recipient replies with a PUBACK including the same message identifier received. This operation allows the sender to correctly identify the response to its request. QoS 1 message flow is shown in figure 3.4.
When the acknowledgment is not received after a specified period of time, the sender retransmits the message within a reasonable time period. The PUBLISH duplicate flag is also set.
The message arrives at the receiver at least once. Thus, the recipient can acknowledge and process both original and duplicate messages.
Overall, the QoS 1 approach guarantees more reliability and efficiency but at cost of having also duplicates at receiver-side.

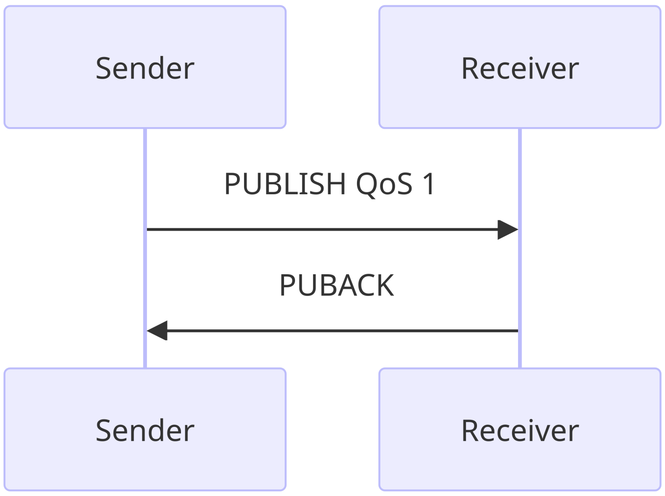


Figure 3.4: Quality of Service 1

QoS level 2 (Exactly once): Upon reception of a PUBLISH message with QoS 2, the message is retained by the receiver and acknowledged with a PUBREC message (PUBlish RECeived). Once it receives a PUBREC message, the sender sends a PUBREL message (PUBlish RELease) to the recipient. Finally, upon reception of the PUBREL message, the receiver sends back a PUBCOMP (PUBlish COMPlete) to the sender.
PUBREC, PUBREL and PUBCOMP messages all contain the same original PUBLISH message identifier.
The receiver processes the original PUBLISH message only upon receiving the first PUBREL message. This is the “exactly once” mechanism.
The overall QoS 2 flow is illustrated in figure 3.5.

In the first handshake (PUBLISH - PUBREC), a crucial point is that, after receiving a PUBLISH message, the recipient stores the message state until it receives subsequently the first PUBREL message from the sender. Upon reception, the receiver can delete the state related to the PUBLISH message. In this way, all the eventually next PUBREL retransmissions will be discarded avoiding multiple processing.
The other important aspect is that, still in the first handshake, also the sender stores the state of the PUBLISH message to be sent. Upon reception of the first PUBREC message, the sender deletes its PUBLISH message state and stores the PUBREC one. This operation avoids multiple processing of received PUBREC messages in case of PUBLISH message retransmissions.

QoS 2 offers the highest level of service, ensuring that each message is delivered exactly once to the recipients without any duplicates. There is an increase in network traffic, but it is usually acceptable because of the importance of the message content.

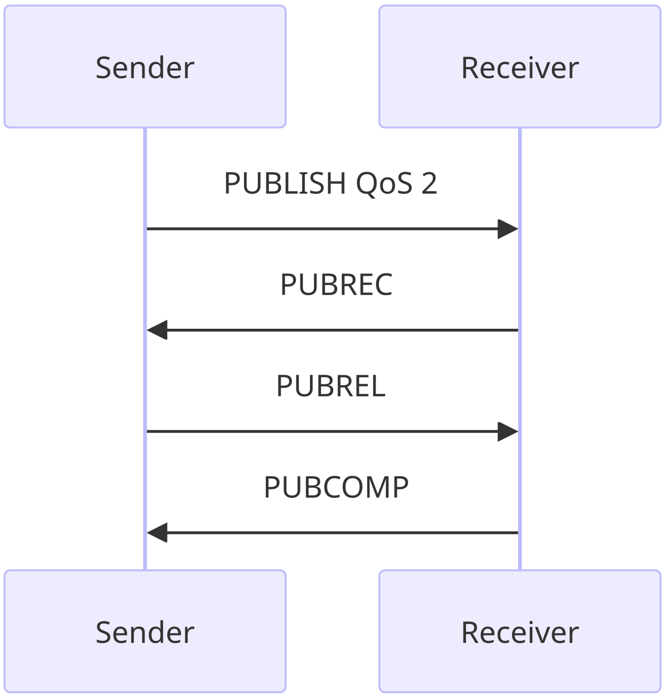


Figure 3.5: Quality of Service 2

The simplified Quality of Service presentation is over. As next step, the concept will be applied at MQTT-SN system level.

Overall, the protocol architecture applies two stages of QoS levels.
The assumption is that all the clients are connected to the same gateway. Publishers publish to a certain topic and subscribers are subscribed to a certain topic. Clearly, both stages refer to the same topic.
In the first stage, between a publisher and a broker, the QoS level to be applied is simply indicated by the publisher in the PUBLISH message field.
The second stage, instead, occurs between a broker and an interested subscriber. The QoS level to be applied is the **minimum** between the QoS specified in the PUBLISH message received by the broker in the first stage and the QoS desired by the subscriber in the subscription phase. In this stage, the broker decides the final QoS level, specifying it in a new PUBLISH message, and acts as a “publisher”.
Among various scenarios, a couple of examples will be presented. In figure 3.6, it is shown that the resulting QoS level between broker and subscriber is 2.

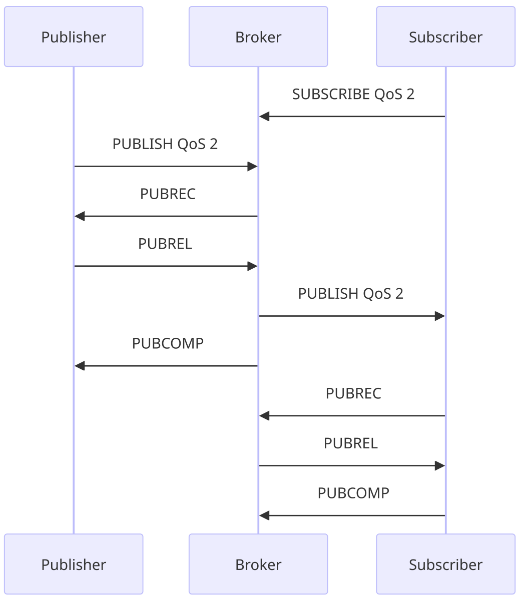


Figure 3.6: QoS Level 2 from Broker to Subscriber

In figure 3.7, instead, it is shown that the resulting QoS level between broker and subscriber is 0, indicating a downgrade.

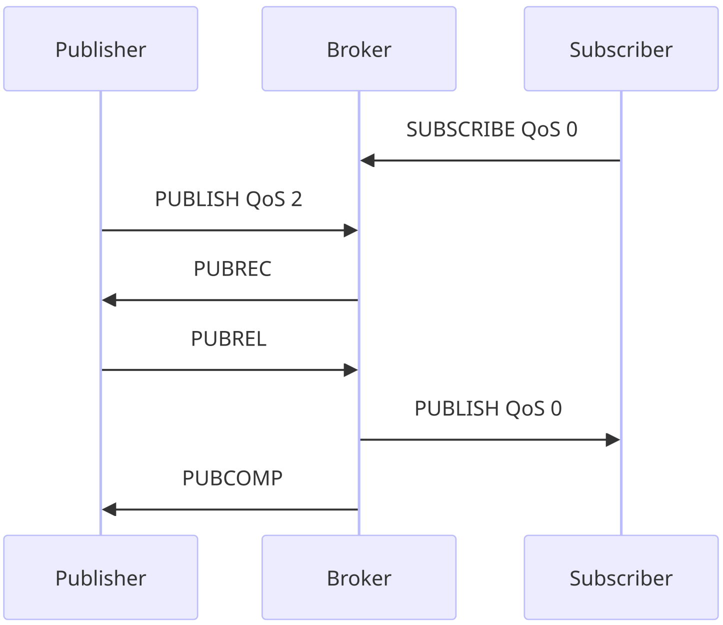


Figure 3.7: QoS Level 0 from Broker to Subscriber

Finally, the protocol also offers the **QoS level -1** that is defined for very simple client implementations which do not support any other features except this one.

Similar to **QoS level 0**, the publisher does not expect an acknowledgment or guarantee of message delivery. However, QoS -1 goes further by completely eliminating the notion of connection session between the publisher and the broker.

3.1.6. MQTT Comparison

MQTT-SN is designed to be as close as possible to MQTT, but is adapted to the peculiarities of a wireless communication environment. It is also optimized for the implementation on low-cost, battery-operated devices with limited processing and storage resources.

MQTT-SN offers a lightweight connection phase splitting the CONNECT message into three messages, making more modular the process with respect to MQTT.

In MQTT-SN, topics, are managed in a different way, avoiding the use of the entire topic name within PUBLISH messages but replacing it with a topic identifier, previously registered with the gateway. Also, predefined and short topics are introduced, for which no registration is required.

The discovery mechanism supported by MQTT-SN enables clients to automatically find a gateway that is available in the network, making the initial configuration of devices in complex or dynamic networks easier.

MQTT-SN supports an offline keep-alive procedure defined for sleeping clients. With this approach, battery-operated devices can go to a sleeping state during which all messages destined to them are buffered at the server and delivered later when they wake up.

MQTT-SN, in addition to the three MQTT's QoS levels, also introduces the QoS level -1.

3.2. OMNeT++ Discrete Event Simulator

The section will provide a short presentation of the main features of OMNeT++, which is used as a simulator in this project.
The OMNeT++ documentation consulted is the official one [3].

3.2.1. Introduction

OMNeT++ is a scalable, highly modular and well-organized discrete event simulator. Based on C++, it offers a fundamental framework through which modules communicate with one another. OMNeT++ is an abbreviation for Objective Modular Network Testbed in C++. Academic research institutions can use it for free, and its distribution policy is open-source. It has a robust graphical user interface in addition to a command line interface, and it runs on Linux and Windows Unix platforms. In addition to validating hardware architectures, the simulator can be used to simulate multiprocessor systems, communication and queuing networks, and other distributed hardware systems.

Due to its several advantages that align with the thesis objectives, OMNet++ has been chosen as simulation environment.
Its support for NED language and C++ for module programming offers flexibility and customization. Furthermore, once learned the basics, its simplicity is also one of the characteristics that support its usage. Finally, the extensibility and the capability to integrate additional frameworks are enormously advantageous.

3.2.2. Modeling Concept

An OMNeT++ model consists of hierarchically nested modules, which communicate by passing messages to each other. The whole model, called network in OMNeT++, is itself a compound module.
An overview of the model is illustrated in figure 3.8.

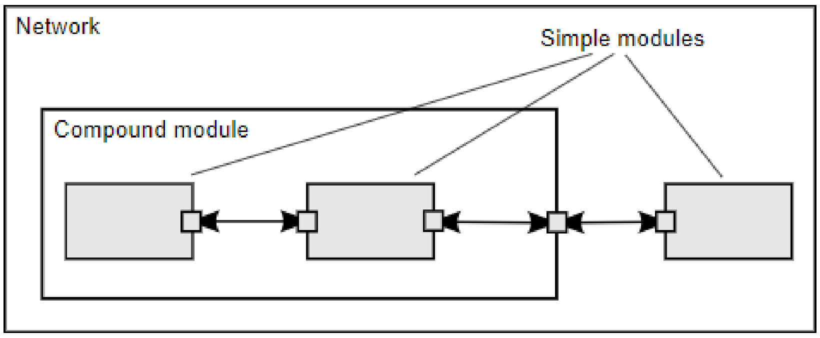


Figure 3.8: OMNeT++ Model

The top level module is the system module. Sub-modules that are contained within the system module may also contain other sub-modules. Because there are no restrictions on the depth of module nesting, the user can represent with the model the logical structure of the real system.
The model composition is described with OMNeT's NED language. Modules that contain sub-modules are termed compound modules, as opposed to simple modules which are at the lowest level of the module hierarchy. The model's algorithms are contained within simple modules. Using the OMNeT++ simulation class library, the user implements the basic modules in C++.
Modules communicate with messages that may contain arbitrary data. A typical message use case scenario in a simulation could be represented by packets in a computer network. The local simulation time of a module advances when the module receives a message. The message can arrive from another module or from the same module. Self-messages are used to implement timers.
Gates are the input and output interfaces of modules: messages are sent through output gates and arrive through input gates.
An input gate and output gate can be linked by a connection. Each connection, also called link, is created within a single level of the module hierarchy: within a compound module, one can connect the corresponding gates of two sub-modules, or a gate of one sub-module and a gate of the compound module. Because of the hierarchical structure of the model, messages typically travel through a chain of connections, starting and arriving in simple modules. Compound modules act like "cardboard boxes" in the model, transparently relaying messages between their inside and the outside world.
Modules can have parameters. These parameters are crucial for configuring module behaviours and defining their topology. Parameters can be assigned in either the NED files or the configuration file `omnetpp.ini`.

3.2.3. Workflow

A brief overview of the workflow for using OMNeT++ will be discussed. The objective is also to present the files that compose a basic model.

- Building blocks (modules) of an OMNeT++ model exchange messages with one another. It is possible for modules to be nested, meaning that multiple modules can be combined to create a compound module. The process involves mapping the system into a hierarchy of interconnected modules in order to create the model.
- The model structure is defined in the NED (NEtwork Description) language. It is organized in one or more files, facilitating modular design and organization. The main one describes the simulation network topology with its modules and connections. Figure 3.9 below shows an example of a simple network topology.

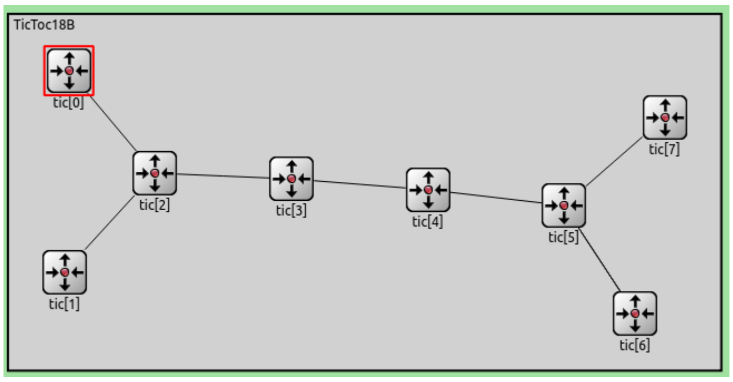
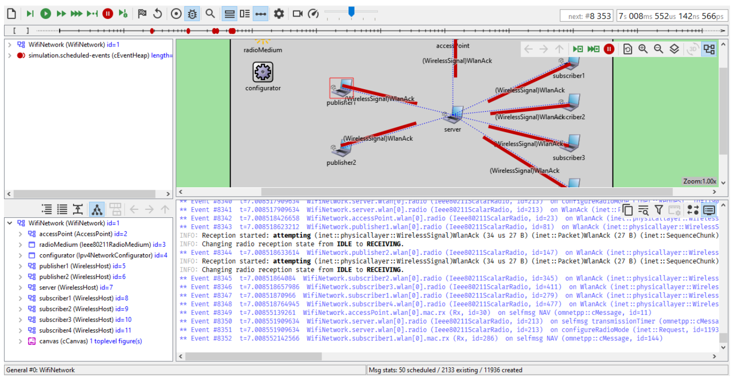


Figure 3.9: NED Network Topology

- Active components of the model (simple modules) are programmed in C++, using the simulation kernel and class library. C++ classes that represent protocol headers are described in message definition files which are then translated into C++ code.
- A suitable `omnetpp.ini` file to hold OMNeT++ configuration and parameters of the model, which is crucial for customizing the network and modules behaviour.
- Building the simulation program and running it as next step. OMNeT++ will link it to a simulation environment that includes a graphical user interface, as shown in figure 3.10



```
WifiNetwork (WifiNetwork) id=1
simulation.scheduled-events (cEventHeap) length=

WifiNetwork (WifiNetwork) id=1
  accessPoint (AccessPoint) id=2
  radioMedium (Ieee80211RadioMedium) id=3
  configurator (Ipv4NetworkConfigurator) id=4
  publisher1 (WirelessHost) id=5
  publisher2 (WirelessHost) id=6
  server (WirelessHost) id=7
  subscriber1 (WirelessHost) id=8
  subscriber2 (WirelessHost) id=9
  subscriber3 (WirelessHost) id=10
  subscriber4 (WirelessHost) id=11
  canvas (cCanvas) 1 toplevel figure(s)
```

```
** Event #8340  t=7.008517909634  WifiNetwork.server.wlan[0].radio (Ieee80211ScalarRadio, id=213)  on configureRadioMode (inet::Request, id=1190
** Event #8341  t=7.008517909634  WifiNetwork.server.wlan[0].radio (Ieee80211ScalarRadio, id=213)  on WlanAck (inet::P
** Event #8342  t=7.008518426658  WifiNetwork.accessPoint.wlan[0].radio (Ieee80211ScalarRadio, id=23)  on WlanAck (inet::physicallayer::Wireless
** Event #8343  t=7.008518623212  WifiNetwork.publisher1.wlan[0].radio (Ieee80211ScalarRadio, id=81)  on WlanAck (inet::physicallayer::WirelessS
INFO: Reception started: attempting (inet::physicallayer::WirelessSignal)WlanAck (34 us 27 B) (inet::Packet)WlanAck (27 B) (inet::SequenceChunk)
INFO: Changing radio reception state from IDLE to RECEIVING.
** Event #8344  t=7.008518633614  WifiNetwork.publisher2.wlan[0].radio (Ieee80211ScalarRadio, id=147)  on WlanAck (inet::physicallayer::Wireless
INFO: Reception started: attempting (inet::physicallayer::WirelessSignal)WlanAck (34 us 27 B) (inet::Packet)WlanAck (27 B) (inet::SequenceChunk)
INFO: Changing radio reception state from IDLE to RECEIVING.
** Event #8345  t=7.00851864084  WifiNetwork.subscriber2.wlan[0].radio (Ieee80211ScalarRadio, id=345)  on WlanAck (inet::physicallayer::Wireless
** Event #8346  t=7.008518657986  WifiNetwork.subscriber3.wlan[0].radio (Ieee80211ScalarRadio, id=411)  on WlanAck (inet::physicallayer::Wireles
** Event #8347  t=7.00851870966  WifiNetwork.subscriber1.wlan[0].radio (Ieee80211ScalarRadio, id=279)  on WlanAck (inet::physicallayer::Wireless
** Event #8348  t=7.008518764945  WifiNetwork.subscriber4.wlan[0].radio (Ieee80211ScalarRadio, id=477)  on WlanAck (inet::physicallayer::Wireles
** Event #8349  t=7.00855139261  WifiNetwork.accessPoint.wlan[0].mac.rx (Rx, id=30)  on selfmsg NAV (omnetpp::cMessage, id=11)
** Event #8350  t=7.008551909634  WifiNetwork.server.wlan[0].radio (Ieee80211ScalarRadio, id=213)  on selfmsg transmissionTimer (omnetpp::cMessa
** Event #8351  t=7.008551909634  WifiNetwork.server.wlan[0].radio (Ieee80211ScalarRadio, id=213)  on configureRadioMode (inet::Request, id=1193
** Event #8352  t=7.008552142566  WifiNetwork.subscriber1.wlan[0].mac.rx (Rx, id=286)  on selfmsg NAV (omnetpp::cMessage, id=144)
```

```
General #0: WifiNetwork
Msg stats: 50 scheduled / 2133 existing / 11936 created
```

Figure 3.10: OMNeT++ Simulator

- Simulation results are written into output vector and output scalar files. Analysis and plotting tools are available.

3.2.4. INET Framework

In order to accomplish the study objectives, it was necessary to find a solution that allows the emulation of real-case scenarios without building all the components from scratch. The INET framework was the ideal solution [4].

INET is an open-source model library for the OMNeT++ simulation environment. It provides protocols, agents, and models that facilitate case studies in communication networks. Wired, wireless, and mobile simulations are optimally supported, and it is especially useful when designing and validating new protocols.
In particular, it offers models for the Internet stack (TCP, UDP, IPv4, etc.), wired and wireless link layer protocols (Ethernet, IEEE 802.11, etc.), support for mobility, and many other protocols and components.

INET benefits from the infrastructure provided by OMNeT++. It is built around the concept of modules that communicate by message passing. Agents and network protocols are represented by components, which can be freely combined to form hosts, routers, switches, and other networking devices. New components can be programmed by the user, and existing components can be easily modified.
Figure 3.11 shows a basic network with INET modules.

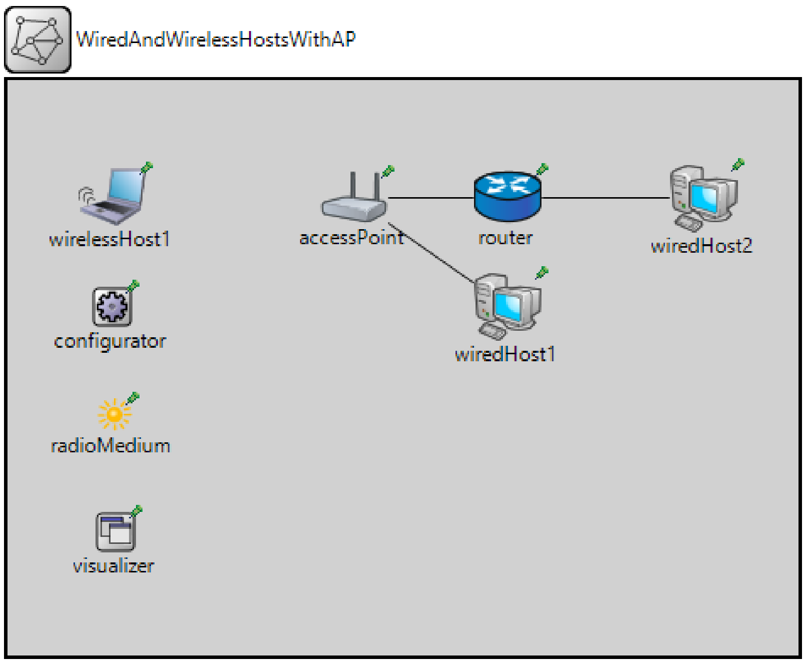


Figure 3.11: Network Scenario with INET

A brief overview on the main modules used to simulate and to accomplish the analysis is provided:

- `WirelessHost`: it models a host with one wireless (802.11) card in infrastructure mode by default. It should be used in conjunction with an Access Point, or any other AP model which contains IEEE 802.11 Interface. A Wireless Host also integrates the complete TCP/IP stack, which is crucial for simulating a wide range of applications that require TCP and UDP protocols at the transport layer.
- `AccessPoint`: it supports multiple wireless radios and multiple ethernet ports serving as a central point for communication within the network.
- `RadioMedium`: it describes the shared physical medium where communication takes place. The module computes and manages all about transmissions and noises within the radio environment. Additionally, it efficiently simulates interfering communications for the receivers.
- `Ipv4NetworkConfigurator`: it manages configurations of IPv4 networks in the simulation, assigning IP addresses, setting up routing tables and many other features. The module supports both static and dynamic settings.

3.3. Implementation

The final part of this chapter will be dedicated to describing the overall implementation of MQTT-SN. After an overview of the project's directories, the discussion will proceed with the workflow presented in the previous section, thus maintaining the coherence in the presentation.

3.3.1. Project Overview

The main directories and files are shown in figure 3.12.

```
/
├── simulations/
│   ├── results/
│   ├── omnetpp.ini
│   └── WifiNetwork.ned
└── src/
    ├── externals/
    ├── helpers/
    ├── messages/
    ├── modules/
    ├── neds/
    ├── tags/
    └── types/
```

Figure 3.12: Project Structure

Top-level project directory is organized into two principal sub-directories: `simulations` and `src`.

The `simulations` directory contains essential files for the description and the configuration of a simulation. In particular:

- `WifiNetwork.ned`: NED file used for building and modifying the network topology. INET modules are utilized
- `omnetpp.ini`: configuration file used for customizing the network and modules behaviour with parameters
- `results`: directory containing the emulation results written into output vector and output scalar files. Also custom formats should be placed in this folder.

All source files are structured in sub-directories and held within `src`. Most of them are in C++, with the exception of the files in the `neds` directory. A quick description of the main sub-directories is provided:

- `neds`: it contains NED files, each providing descriptions for MQTT-SN modules
- `modules`: it holds C++ class files, each providing implementations for MQTT-SN modules
- `messages`: MQTT-SN messages are defined within each C++ class file in this folder
- other remaining folders provide C++ application definitions and utilities

The core behaviour of MQTT-SN application is implemented within the `modules` directory. In order to represent the protocol architecture, C++ classes are structured in a hierarchical way. Also NED files in `neds` folder follow the same approach. Thus, modules and C++ classes are aligned for coherence. Figure 3.13 below shows the MQTT-SN application structure.

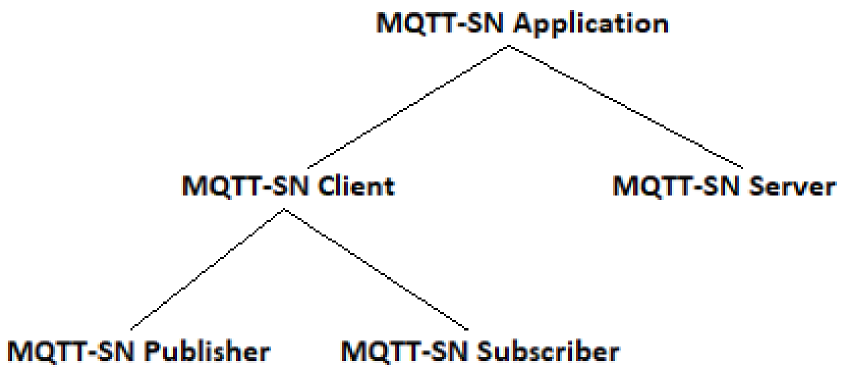


Figure 3.13: MQTT-SN Application Structure

A NED module extends its parent, inheriting parameters and other features.
Similarly, a C++ class also uses the same mechanism with Object-Oriented Programming (OOP), inheriting parent's attributes and methods.
The two concepts are linked together using, within each NED module, the `@class` directive, binding the module with its corresponding C++ class.

Inheritance is a crucial aspect used for describing the overall application behaviour and structure, allowing the components to share common features.

Parameters usage is another fundamental point for the application configuration. A parameter is defined within a NED module and its value can be assigned either in the NED module itself or in the `omnetpp.ini` file. These values are utilized by C++ classes in a variety of scenarios.

3.3.2. Network Topology

The aim of this project is to simulate the MQTT-SN protocol over a wireless network and to analyze its QoS level performances under certain conditions.

As first step, a basic wireless network is built, with all devices using the IEEE 802.11 standard (Wi-Fi). INET components meet this requirement and adopt by default the 802.11g configuration. The modules used are (see previous section 3.2.4):

- `WirelessHost`
- `AccessPoint`
- `Ieee80211RadioMedium`
- `Ipv4NetworkConfigurator`

The overall wireless network topology is shown below in figure 3.14. As mentioned previously, its description is within the `simulations/WifiNetwork.ned` file. Instead, application modules are included under the `src/neds` folder.

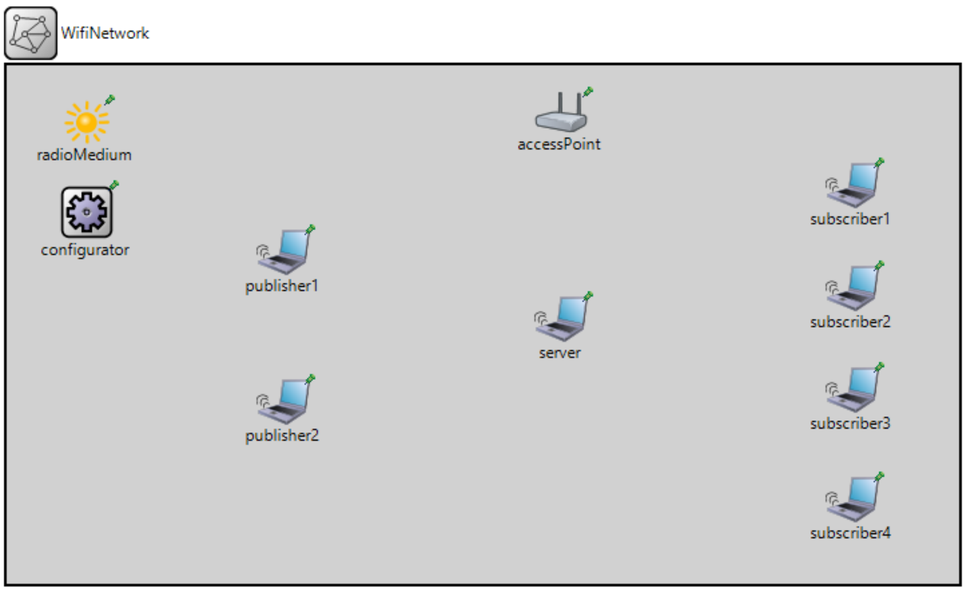


Figure 3.14: Wireless Network Topology

As illustrated in the figure, MQTT-SN application module is incorporated into each `WirelessHost` to simulate the architecture of the protocol throughout the network nodes. This action is defined within the configuration file `simulations/omnetpp.ini`.

More specifically, `WirelessHost` provides an application interface (also referred to as `IApp`) that facilitates the binding of modules that make use of this interface. The parent

MQTT-SN application module, within the `src/neds` directory, uses `IApp`, connecting its communication gates to the appropriate sockets.

At higher level, the MQTT-SN architecture is composed of two publishers, four subscribers and one broker. The aim is to simulate a real-case scenario in which publishers send data to subscribers through a broker.

3.3.3. C++ Classes

The application C++ classes, as already introduced, are located within `src/modules` directory. Files are organized in a manner that follow nearly the hierarchical structure of the protocol. Also NED modules in `src/neds` adhere to the same structure. The figure 3.15 below shows the disposition of the files within the folder.

```
/
└── modules/
    ├── client/
    │   ├── MqttSNClient.cc
    │   ├── MqttSNClient.h
    │   ├── MqttSNPublisher.cc
    │   ├── MqttSNPublisher.h
    │   ├── MqttSNSubscriber.cc
    │   └── MqttSNSubscriber.h
    ├── server/
    │   ├── MqttSNServer.cc
    │   └── MqttSNServer.h
    ├── MqttSNApp.cc
    └── MqttSNApp.h
```

Figure 3.15: C++ Classes for MQTT-SN Application

At the top level, `MqttSNApp` class contains general application attributes and methods that are inheritable by its child classes. In particular, the class defines and uses `ApplicationBase` and `UdpSocket` features provided by the INET framework. The latter is widely utilized in every subclass in order to send and receive application packets through the network using a socket with UDP as the transport layer. The packets themselves are INET-based offering a feature-rich and versatile data structure. Henceforth, these type of packets at application layer will simply be referred as "packets".

`MqttSNClient` and `MqttSNServer` classes derive from the `MqttSNApp` class. The former implements all the functionalities required for a correct MQTT-SN client operation, including gateway discovery, connection phase, retransmission mechanism for unicasted messages and client's state transition management. The latter has a central role in the overall protocol handling, applying the functionalities described within the specifications. Finally, `MqttSNPublisher` and `MqttSNSubscriber` classes extend `MqttSNClient` class, adding specialization for their respective roles. In particular, `MqttSNPublisher` includes topic registration and message publication procedures. `MqttSNSubscriber`, instead, enables for topic subscription, unsubscription and manages publish messages received from the broker.

In addition to the application classes, another essential element of the MQTT-SN implementation is the definition and the handling of the message format.
In order to represent the different types of MQTT-SN messages, it was necessary to use specialized C++ classes that extend `FieldsChunk`, a base class made available by the INET environment. Inheritance facilitates the various message format compositions described in the protocol specifications. Furthermore, the INET packet data structure builds on top of chunks simplifying the overall process of message management. Thus, each MQTT-SN message is encapsulated within an INET packet.
The `src/messages` directory contains all MQTT-SN message definitions. In particular, `MqttSNBase` class, which inherits from `FieldsChunk` and represents the message header, serves as the base class for the child classes with specialized messages fields.

3.3.4. Parameters

One of the important aspects of the overall system is the usage of parameters. The file `simulations/omnetpp.ini` plays a central role for customizing and configuring every feature of the simulation environment.
As first step, the connection setup between devices was necessary. Thus, by modifying `Ipv4NetworkConfigurator` module parameter, a `WirelessHost` is associated with a static IP address. Although the configurator module can manage automatically this operation it was chosen to assign the addresses to the devices manually for better debugging purposes. At application layer, every node must bind its UDP socket with its own IP address and local port. The latter also is specified in the configuration file.
The client operations, such as gateway discovery, connection management, message retransmission settings and state transition intervals are all configurable via counters and values indicated as parameters. Most of these parameters are expressed in the client NED file as default values, following the "best practice" assignments described in the specifications. This practice is not limited only to the client configuration but extends in the overall system component.
The most used and variable parameters are those relating to publishers and subscribers.

For the former, `simulations/omnetpp.ini` includes settings about topic definition, registration and publication tasks. In particular, JSON-formatted values indicate the topics and their data to be published during the simulation. The number of registrations and publications and their associated intervals are among the parameters defined.
Similarly, JSON-formatted values are used for subscribers, specifying the topics to be subscribed. Subscription and unsubscription operations are also parameterized.
Furthermore, Quality of Service levels, crucial for the overall analysis, are specified as fields within the JSON descriptions.
Finally, regarding the JSON format, it is parsed using an external open-source implementation located in `src/externals` project directory.

3.3.5. Validation Approach

The implementation of the protocol as closely as possible to its specifications is one of the project's main goals. Indeed, the main and crucial MQTT-SN functionalities are covered with an accurate development and validation approach during all the phases. In particular, the strategy adopted is an incremental approach, firstly implementing a specification and then validating it with one or more simulation runs.
The pivotal OMNeT++ command utilized is `EV` that represents the "environment" or user interface of the simulation. This is a key command for printing debug messages in the emulation environment, not only to check the state variable values but also to verify the correct execution flow. By positioning `EV` in strategical points in the code, it helps avoiding any kind of error. Each specific function increment is considered "validated" if results are the expected one, even after some repeated simulation with distinct data and different parameter configurations. Additionally, at every step, an overall simulation execution is done in order to avoid unpredictable system errors or even some incorrect behaviour or crashes after some emulation time.
A fundamental feature provided by the simulation environment is the possibility to choose the running mode, facilitating the validation objective. In particular, there are four simulation running speeds, from the slowest (Step Run) to the fastest one (Express Run). Slower emulations allow the console log events exploration and the output printing, being able to verify in detail many crucial simulation aspects supported also by a graphical visualization. Faster modes, on the other hand, enable to verify the system behaviour at later simulation stages, speeding up the process.

The overall approach described aims to correctly implement the MQTT-SN protocol over the OMNeT++ simulator, to scrupulously validate every functionality and to maintain the system behavioural integrity with environmental changes modeled by the configuration parameters.

4 | Experiments

The chapter describes the experiments conducted after the completion of the protocol implementation on the simulator. Specifically, it aims to analyze Quality of Service behaviour under certain simulation conditions, using selected performance metrics applied to publication packets from publishers to subscribers. Meanwhile, simulation errors are introduced for all the protocol packets in order to evaluate the effect on the overall network and in particular on the QoS levels.

The first section provides a description of the used performance metrics and their integration into the implementation. Next, the second part discusses the error model adopted and its application within the simulation environment. Finally, the third section outlines the simulation setup and examines the obtained results focusing on two distinct analysis scenarios.

4.1. Performance Metrics

In this section, the key performance metrics used to assess the Quality of Service will be described. The focus will be on the End-to-End Delay of packets flowing from publishers to subscribers and the Hit Rate indicating the overall delivery success rate.
The introduced metrics are crucial to evaluate the QoS performance and consequently the system behaviour in terms of responsiveness and reliability within a wireless network.
Additionally, the number of duplicates will be considered in order to analyze and to confirm certain aspects regarding the QoS.
These introduced metrics, as already mentioned, are applied specifically to publication packets transmitted from publishers to interested subscribers, as the QoS is implemented with this type of message.
Furthermore, an important development element is the use of INET-based tags that allows the exchange of information within packets in a transparent manner, without influencing the actual protocol message size. This is useful for the application of the metrics and the simulation objectives, enabling the recipients (subscribers) to accurately trace and manage QoS-related metrics. In particular, INET provides commonly used tags but customization

is also possible. The latter goal is achieved in the `src/tags` directory.

4.1.1. Average End-to-End Delay

The objective is to calculate the average End-to-End delay of publication packets received by subscribers. A single End-to-End delay is obtained by subtracting the packet's sending timestamp from its arrival timestamp. The average delay is obtained by dividing the sum of these individual delays by the total number of received packets.
Equation (4.1) summarizes as said before:

$$\text{Average End-to-End Delay} = \frac{\sum_{i=1}^{N}(T_{\text{arrival},i} - T_{\text{send},i})}{N} \tag{4.1}$$

where N indicates the total number of publication packets received by subscribers, and T represents the timestamp value in seconds.

A publisher, before the publication, embeds within the packet a tag that specifies the current timestamp, thereby indicating the sending time. Upon receiving the packet, a subscriber records this timestamp by adding it to a cumulative variable, which is shared among all subscribers. The same approach is also utilized with the total number of received packets, tracking them incrementally at every reception. At the end of the simulation, the average End-to-End delay is calculated using these two variables. From C++ point of view, these attributes, shared among all the subscribers, are implemented with `static` members.

Overall, the responsiveness of a system is closely related to the End-to-End delay and it describes the ability to respond to changes within the network. A lower End-to-End delay means a more responsive system with quick delivery of packets from the sender to the receiver.

4.1.2. Hit Rate

The aim is to compute the total number of unique publication packets received by the subscribers with respect to the total number of unique publication packets sent by the publishers. This calculation defines the Hit Rate metric.
An important aspect to note is that packets sent are considered across all the publishers and, similarly, the packets received are among all the subscribers. Thus, it can be seen as a *Global Hit Rate* within the overall system.
The uniqueness of packets is another critical point. Specifically, a publication packet

may be received by one or more subscribers. The approach is to consider each packet received by any subscriber just once, ensuring an accurate method for Hit Rate calculation. Equation (4.2) defines the Hit Rate as:

$$\text{Hit Rate} = \frac{\text{Number of Unique Packets Received by Subscribers}}{\text{Number of Unique Packets Sent by Publishers}} \tag{4.2}$$

The integration of the Hit Rate in the development process follows a similar approach to the previous metric. A publisher, before publication, generates a unique identifier for that packet, keeps track of it and embeds it within the packet itself using a tag. Upon receipt, a subscriber extracts the identifier, thereby uniquely identifying the packet. In C++, these operations are facilitated by sets: at the publisher's side, a set tracks the unique identifiers of packets sent by all publishers; similarly, at the subscriber's side, another set tracks the unique identifiers of packets received by all subscribers.
Sets, as containers that store unique elements, are particularly suitable for achieving the discussed objective. Their visibility among the same instances of publishers and subscribers is also ensured in this case with `static` variables. Upon the conclusion of the simulation, the number of elements within each collection will be counted in order to finally calculate the metric.

The Hit Rate measures the reliability of the overall system, providing insights on packet losses under certain network conditions. Lower values of Hit Rate indicate an inefficient transmission with many packet losses from the sender to the receiver.

4.1.3. Auxiliary Indicators

Upon receiving the unique identifier within the publication packet, a subscriber also accounts for duplicates. As in the previous descriptions, it utilizes a set, which, this time, is not shared among themselves. Thus, with an instance attribute each subscriber counts its own duplicate. At the simulation's end, the total number of duplicates will be summed among all the subscribers. The information about duplicates is a key aspect in the QoS level 1, where at the receiver's side they can be present. In many systems, receiving duplicates is undesirable, as it increases processing overhead and reduces network efficiency.

The number of packet retransmissions is another important indicator to be taken into account. Retries can occur from publishers to brokers and from subscribers to brokers for every type of packet. Instead, also from broker to subscribers but only for publication and registration packets. Publishers count their retries with a shared counter among themselves. The same implementation approach is valid also for brokers and subscribers.

Retransmissions provide insights about the overall system communication and also about network conditions. High number of retries indicate that the acknowledgment for each packet is not received, suggesting that the environment presents errors and the system performance is degrading.

4.2. Error Model

The objective is to evaluate the Quality of Service performance with the metrics introduced in the previous section. QoS levels operate within a system represented by the protocol and its functionalities. Thus, the introduction of error conditions will be done taking into account the overall complexity and not only the QoS feature as an isolate case. A holistic approach provides a general view of the MQTT-SN protocol, enabling to simulate real-case scenarios. Based on this concept, the error model should be applied to all the MQTT-SN packets transmitted within the network.
Any corruption in the communication will impact the transmission of higher-layer packets, resulting in losses or bit errors. The notion of Bit Error Rate (BER) is introduced.
BER (also known as Bit Error Ratio) is the number of bit errors divided by the total number of transferred bits during a studied time interval.
The packet length is also another important parameter to consider for an error model. Longer packets have an higher probability of corruption to ensure their integrity. A probabilistic error model is presented below in Equation (4.3):

$$\text{Packet Error Probability} = 1 - (1 - \text{BER})^{\text{Length}} \tag{4.3}$$

A basic assumption in this model is that every bit has the same probability error independently from its position within the packet. Considering that the BER represents the probability of a bit error in a packet. Thus, $(1 - \text{BER})$ indicates the success probability of a single bit. Raising this probability to the power of the packet's Length, the result is the probability of sending an entire packet without errors. Subtracting one from the outcome, the overall probability of incurring at least one error in the packet during transmission is obtained. The model described is largely used in networking community and it also named as Packet Error Rate (PER). [5]

The described approach is implemented in C++ with a function that accepts the BER and the packet length as parameters and returns a boolean indicating whether the packet should be considered corrupted or not. If the result value is true, the INET packet flag will be set, indicating a corrupted packet. This method will be used throughout the code before sending any MQTT-SN packet. The BER is set as a parameter in the configuration file `simulations/omnetpp.ini`.

Algorithm 4.1 illustrates the process of determining whether a packet is erroneous, applying the packet error probability as discussed.

Algorithm 4.1 Determining Probabilistic Error Presence

```
Require: Packet bit length ($length$), Bit Error Rate ($ber$)
Ensure: Returns true if the packet is considered corrupted
  Calculate packet error probability: $packetErrorProbability = 1 - (1 - ber)^{length}$
  Generate a random number $rand$ between 0 and 1
  if $rand < packetErrorProbability$ then
    Return $true$ (packet is corrupted)
  else
    Return $false$ (packet is not corrupted)
  end if
```

The comparison of the calculated probability with a randomly generated number is a crucial aspect. The process of adding randomness enables the characterization of packet behaviour within networks as in real-world scenarios. OMNeT++ provides the `uniform` method that returns a random variate with uniform distribution in the range [a,b).
A corrupted packet indicates whether it contains at least one error bit. Therefore, upon this determination, the packet's bit error should be set. INET packet provides the method `setBitError` exactly for the described purpose. Algorithm 4.2 below implements the procedure discussed, utilizing `hasProbabilisticError` method as outlined in the previous step 4.1.

Algorithm 4.2 Determining Packet Corruption

```
Require: Packet ($packet$), Bit Error Rate ($ber$)
  $isCorrupted \leftarrow$ hasProbabilisticError($packet$->$getLength()$, $ber$)
  if $isCorrupted$ = true then
    $packet$->$setBitError(true)$
  end if
```

In conclusion, the probabilistic error model provides an approach to simulate the impact of the error transmissions in real-case network scenarios. This facilitates the evaluation of the overall system and in particular the QoS levels under different BER values. Furthermore, introduction of stochastic elements enables the validation of various simulation scenarios, aligning with the objectives of the thesis.

4.3. Quality of Service Analysis

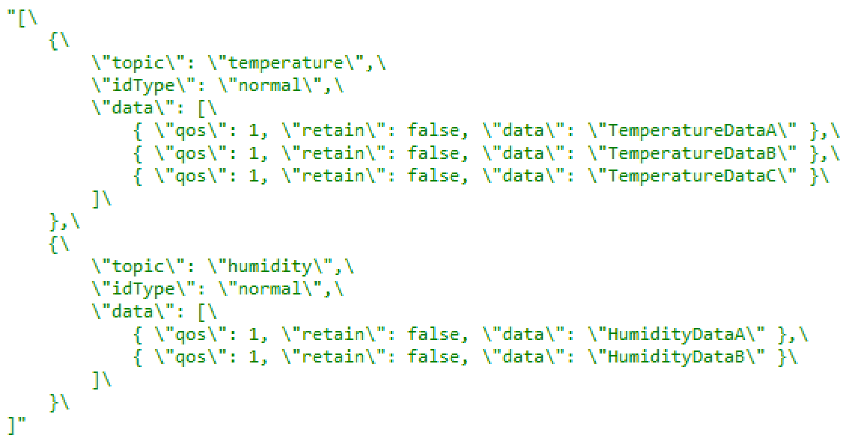
The section will provide an analysis of the protocol's Quality of Service behaviour using the performance metrics and introducing the probabilistic error model within the simulation. As the first steps, an overview of the parameter setup and simulation execution will be presented. Then, a comparison between different QoS levels in terms of End-to-End Delay and Hit Rate will be discussed. Finally, a second analysis will be on the QoS 1 focusing on how performance metrics change in relation to the number of retransmissions per packet illustrating also the variation of number of duplicates and the total retransmissions among publishers, broker and subscribers.

4.3.1. Parameter Setup

The simulation is configured within the `simulations/omnetpp.ini` file. As mentioned previously, the network topology is composed by two publishers, one broker and four subscribers.

Each publisher is configured with these main parameters and values:

- `registrationLimit`: the total number of topic registrations is set to two
- `publishLimit`: the maximum number of publications is equal to 50
- `publishInterval`: the time interval between two consecutive publications is 15 seconds
- `itemsJson`: JSON parameter containing the topic names and their associated data to be published. Each data field includes the QoS level.
  Figure 4.1 below shows an example of the parameter value



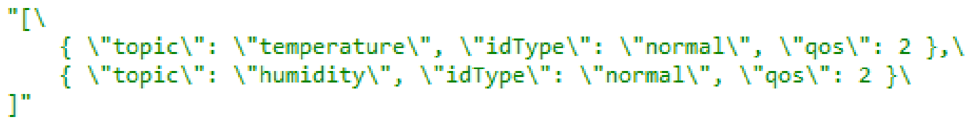
```
"[\
    {\
        \"topic\": \"temperature\",\
        \"idType\": \"normal\",\
        \"data\": [\
            { \"qos\": 1, \"retain\": false, \"data\": \"TemperatureDataA\" },\
            { \"qos\": 1, \"retain\": false, \"data\": \"TemperatureDataB\" },\
            { \"qos\": 1, \"retain\": false, \"data\": \"TemperatureDataC\" }\
        ]\
    },\
    {\
        \"topic\": \"humidity\",\
        \"idType\": \"normal\",\
        \"data\": [\
            { \"qos\": 1, \"retain\": false, \"data\": \"HumidityDataA\" },\
            { \"qos\": 1, \"retain\": false, \"data\": \"HumidityDataB\" }\
        ]\
    }\
]"
```

Figure 4.1: Publisher Items in JSON Format

The total number of publications in the simulation is 100. This value was chosen to conduct the analyses with a satisfactory number of packets to send. Moreover, there are four unique topics, which simplifies the tracking of subscriptions.

Each subscriber is set with these main parameters and values:

- `subscriptionLimit`: the maximum number of subscriptions to topics is equal to two
- `subscriptionInterval`: the time interval between two consecutive subscriptions is two seconds
- `itemsJson`: JSON parameter containing the topics and their QoS level to be subscribed, as shown in an example figure 4.2



```
"[\
    { \"topic\": \"temperature\", \"idType\": \"normal\", \"qos\": 2 },\
    { \"topic\": \"humidity\", \"idType\": \"normal\", \"qos\": 2 }\
]"
```

Figure 4.2: Subscriber Items in JSON Format

Two subscribers will subscribe to the topics of the first publisher, while the remaining ones will subscribe to the topics of the second publisher. Consequently, each subscriber will receive 50 publication packets. Therefore, the total number of publications to be received by all subscribers is 200, evenly distributed.
The subscription QoS levels are set to the maximum to ensure that the resulting QoS level from broker to subscriber is directly determined by the publication's QoS level from publisher to broker. In the examples shown above, the result QoS level of a publish packet from broker to subscriber will be one.

One key parameter in the overall simulation is the Bit Error Rate (BER). Its value will be applied to introduce errors to packets, as discussed previously. The range of values, from `1e-3` to `1e-2` with an interval of `5e-4`, is relatively high in order to simulate an error-prone environment for transmissions.

The number of retransmissions per packet (`Nretry`) is another crucial parameter. In the simulation, it will take on one these three values: three, five and eight. Additionally, retries occur at intervals of 10 seconds, as indicated by the parameter `Tretry`. Both values follow the best practices within the specifications.

Last but not least, the `seed-set` will be utilized to observe varying outcomes for random variables. It allows the replication of a result by using the same seed for the random number generator. The values are: 2, 13, 31, 53 and 73. This choice is made by considering a certain gap between each seed in order to minimize influences between simulations.

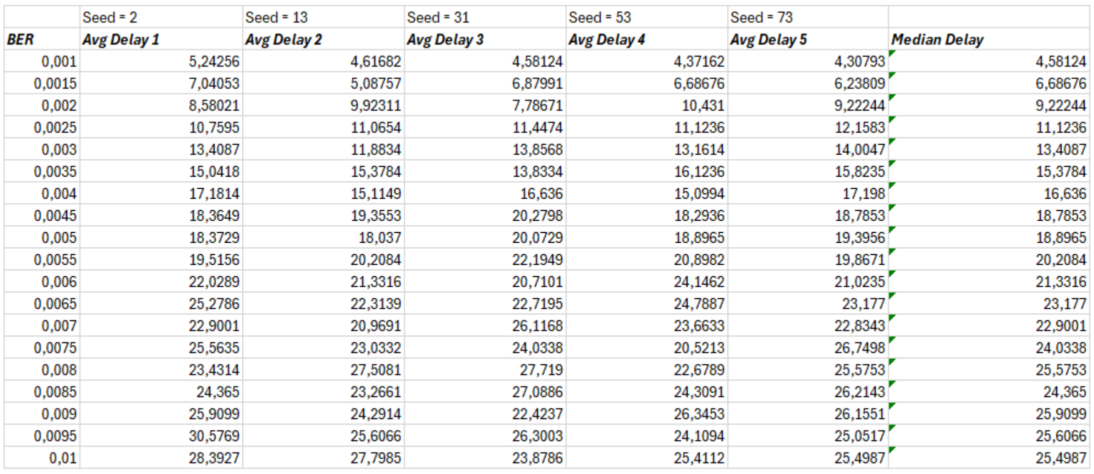
4.3.2. Simulation Execution

Upon configuring the parameters, the simulation can start. There are two focuses, as previously mentioned. The first scenario has the aim to compare the four QoS levels each other using the two performance metrics: End-to-End Delay and Hit Rate. The second approach utilizes the same indicators to evaluate the QoS 1 with different number of retransmissions per packet (`Nretry`). Furthermore, the inclusion of additional elements, like the number of duplicates and the total number of retries, enables to better explore some characteristic.

Starting the description from the first type of simulation, for each QoS level is performed an execution with the five previously specified seeds. Moreover, simulations have been conducted with each seed for every BER value within the specified range. Thus, the total number of simulations can be described as in Equation (4.4) below:

$$\text{Total} = (\text{QoS Levels}) \times (\text{Seeds}) \times (\text{BER Values}) \tag{4.4}$$

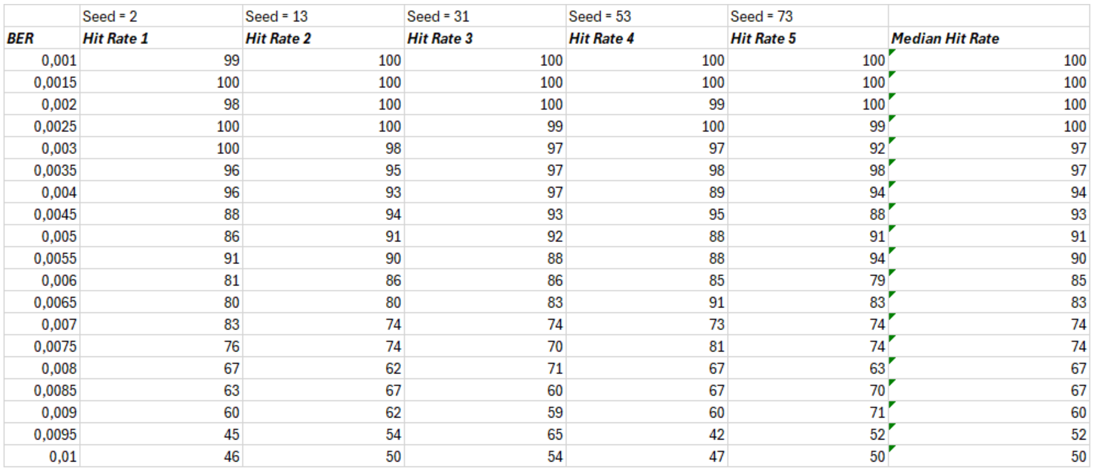
At the conclusion of each simulation, the outcomes are recorded in a CSV file, organized accordingly. Then, a final Excel file is compiled, incorporating these results.
For each QoS level, seed, and BER value, the recorded outcomes include values of End-to-End Delay and Hit Rate. The result metric will have the median values of the five realizations for each QoS level. Figure 4.3 below shows an example of the final table for QoS 1 Median Delays.



	Seed = 2	Seed = 13	Seed = 31	Seed = 53	Seed = 73	
BER	***Avg Delay 1***	***Avg Delay 2***	***Avg Delay 3***	***Avg Delay 4***	***Avg Delay 5***	***Median Delay***
0,001	5,24256	4,61682	4,58124	4,37162	4,30793	4,58124
0,0015	7,04053	5,08757	6,87991	6,68676	6,23809	6,68676
0,002	8,58021	9,92311	7,78671	10,431	9,22244	9,22244
0,0025	10,7595	11,0654	11,4474	11,1236	12,1583	11,1236
0,003	13,4087	11,8834	13,8568	13,1614	14,0047	13,4087
0,0035	15,0418	15,3784	13,8334	16,1236	15,8235	15,3784
0,004	17,1814	15,1149	16,636	15,0994	17,198	16,636
0,0045	18,3649	19,3553	20,2798	18,2936	18,7853	18,7853
0,005	18,3729	18,037	20,0729	18,8965	19,3956	18,8965
0,0055	19,5156	20,2084	22,1949	20,8982	19,8671	20,2084
0,006	22,0289	21,3316	20,7101	24,1462	21,0235	21,3316
0,0065	25,2786	22,3139	22,7195	24,7887	23,177	23,177
0,007	22,9001	20,9691	26,1168	23,6633	22,8343	22,9001
0,0075	25,5635	23,0332	24,0338	20,5213	26,7498	24,0338
0,008	23,4314	27,5081	27,719	22,6789	25,5753	25,5753
0,0085	24,365	23,2661	27,0886	24,3091	26,2143	24,365
0,009	25,9099	24,2914	22,4237	26,3453	26,1551	25,9099
0,0095	30,5769	25,6066	26,3003	24,1094	25,0517	25,6066
0,01	28,3927	27,7985	23,8786	25,4112	25,4987	25,4987

Figure 4.3: QoS 1 Median Delays Table

Similarly, Figure 4.4 illustrates the example table for QoS 1 Median Hit Rates.



	Seed = 2	Seed = 13	Seed = 31	Seed = 53	Seed = 73	
BER	***Hit Rate 1***	***Hit Rate 2***	***Hit Rate 3***	***Hit Rate 4***	***Hit Rate 5***	***Median Hit Rate***
0,001	99	100	100	100	100	100
0,0015	100	100	100	100	100	100
0,002	98	100	100	99	100	100
0,0025	100	100	99	100	99	100
0,003	100	98	97	97	92	97
0,0035	96	95	97	98	98	97
0,004	96	93	97	89	94	94
0,0045	88	94	93	95	88	93
0,005	86	91	92	88	91	91
0,0055	91	90	88	88	94	90
0,006	81	86	86	85	79	85
0,0065	80	80	83	91	83	83
0,007	83	74	74	73	74	74
0,0075	76	74	70	81	74	74
0,008	67	62	71	67	63	67
0,0085	63	67	60	67	70	67
0,009	60	62	59	60	71	60
0,0095	45	54	65	42	52	52
0,01	46	50	54	47	50	50

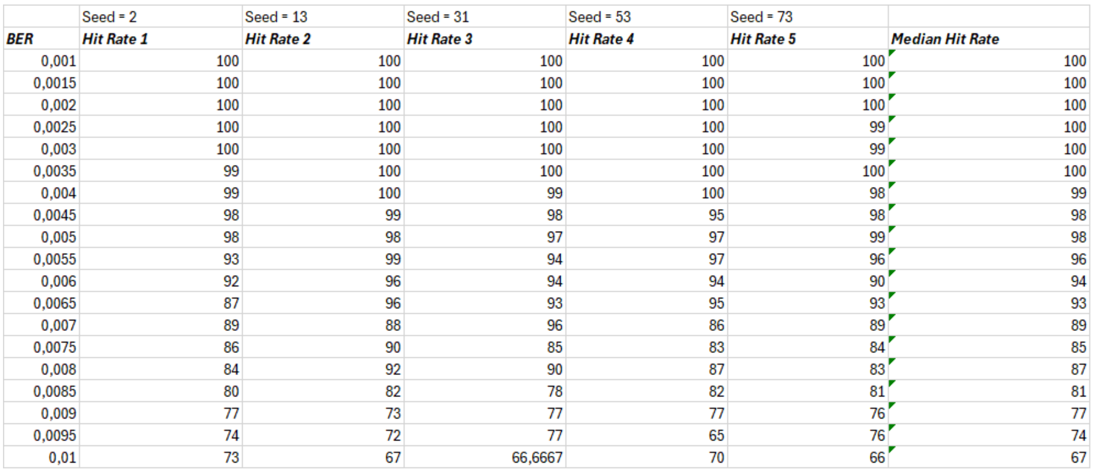
Figure 4.4: QoS 1 Median Hit Rates Table

The described approach in the figures above is executed for each QoS level and subsequently plotted on graphs. The median indicates the central value among others and its use is due to the robustness it provides against outliers.
From a configuration perspective in `simulations/omnetpp.ini` file, the involved parameters are: `seed-set`, `BER` and the QoS levels specified within publisher's `itemsJson`. Number of retries per packet (`Nretry`) is set to three.

The second type of simulation uses the same method as discussed, but it focuses only on QoS 1. The variable part of the simulation is the number of retries per packet within the system. It can assume three values as previously specified. Hence, the total number of simulations can be described as in Equation (4.5) below:

$$\text{Total} = (\text{Nretry Values}) \times (\text{Seeds}) \times (\text{BER Values}) \qquad (4.5)$$

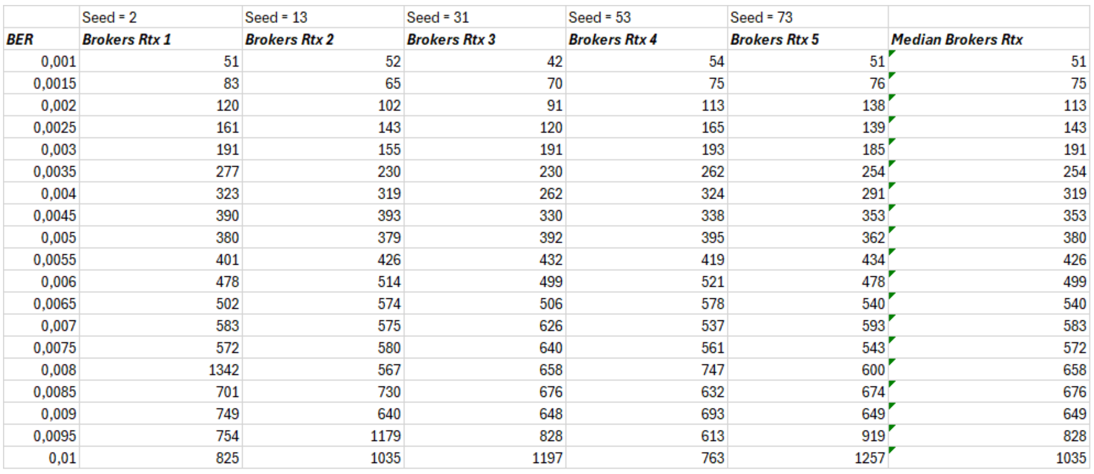
For each retry value, seed, and BER value, the recorded outcomes include values of End-to-End Delay, Hit Rate, Duplicates and Total Retransmissions from publishers, broker and subscribers. Figure 4.5 below illustrates an example of the table displaying the Median Hit Rates using `Nretry` set to five.



	Seed = 2	Seed = 13	Seed = 31	Seed = 53	Seed = 73	
BER	***Hit Rate 1***	***Hit Rate 2***	***Hit Rate 3***	***Hit Rate 4***	***Hit Rate 5***	***Median Hit Rate***
0,001	100	100	100	100	100	100
0,0015	100	100	100	100	100	100
0,002	100	100	100	100	100	100
0,0025	100	100	100	100	99	100
0,003	100	100	100	100	99	100
0,0035	99	100	100	100	100	100
0,004	99	100	99	100	98	99
0,0045	98	99	98	95	98	98
0,005	98	98	97	97	99	98
0,0055	93	99	94	97	96	96
0,006	92	96	94	94	90	94
0,0065	87	96	93	95	93	93
0,007	89	88	96	86	89	89
0,0075	86	90	85	83	84	85
0,008	84	92	90	87	83	87
0,0085	80	82	78	82	81	81
0,009	77	73	77	77	76	77
0,0095	74	72	77	65	76	74
0,01	73	67	66,6667	70	66	67

Figure 4.5: Median Hit Rates Table for Nretry=5

Similarly, Figure 4.6 illustrates another example table for `Nretry` equal to five.



	Seed = 2	Seed = 13	Seed = 31	Seed = 53	Seed = 73	
BER	***Brokers Rtx 1***	***Brokers Rtx 2***	***Brokers Rtx 3***	***Brokers Rtx 4***	***Brokers Rtx 5***	***Median Brokers Rtx***
0,001	51	52	42	54	51	51
0,0015	83	65	70	75	76	75
0,002	120	102	91	113	138	113
0,0025	161	143	120	165	139	143
0,003	191	155	191	193	185	191
0,0035	277	230	230	262	254	254
0,004	323	319	262	324	291	319
0,0045	390	393	330	338	353	353
0,005	380	379	392	395	362	380
0,0055	401	426	432	419	434	426
0,006	478	514	499	521	478	499
0,0065	502	574	506	578	540	540
0,007	583	575	626	537	593	583
0,0075	572	580	640	561	543	572
0,008	1342	567	658	747	600	658
0,0085	701	730	676	632	674	676
0,009	749	640	648	693	649	649
0,0095	754	1179	828	613	919	828
0,01	825	1035	1197	763	1257	1035

Figure 4.6: Median Broker Retransmissions Table for Nretry=5

4.3.3. QoS Comparison Results

The final objective is to evaluate the protocol's QoS behaviour using performance metrics and conducting comparisons.

As an initial result, the focus is on the Median End-to-End Delay, consolidating the QoS levels in one place and observing their differences. Figure 4.7 shows the Median End-to-End Delays among the various QoS levels.

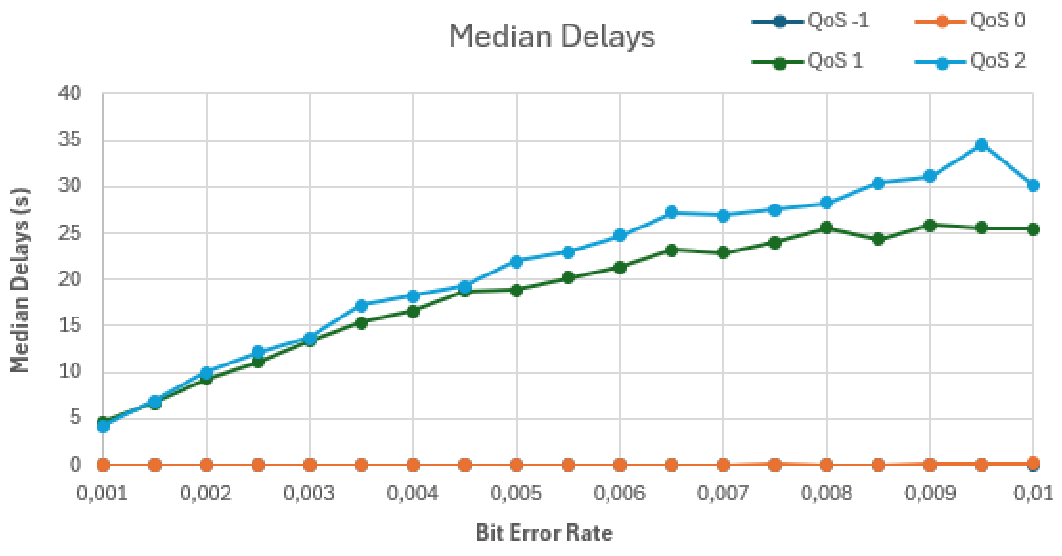


Figure 4.7: QoS Levels: Median End-to-End Delay Comparison

The first behaviour observable from the plot is that QoS 1 and QoS 2 exhibit the poorest performance in terms of delay compared to the other QoS levels. Indeed, median delays for QoS 1 and QoS 2 are significantly higher, reaching several seconds. On the other hand, QoS -1 and QoS 0 show delays on the order of milliseconds. This evidence is justified by the fact that QoS -1 and QoS 0 do not have any acknowledgment upon receipt; their approach is "fire and forget". Conversely, QoS 1 and QoS 2 wait for handshakes and if there is no response, the packets are retransmitted. As specified previously, the number of retries per packet is three at intervals of every 10 seconds.
To summarize, the impact of increasing BER is notably lower for QoS -1 and QoS 0 in terms of end-to-end delay.
From the plot is also possible to understand that QoS 1 median delays are lower than those of QoS 2 at almost every BER value. The gap becomes evident for higher values of BER. This behaviour is attributed to the fact that QoS 2 involves more packet exchange phases compared to QoS 1.

The second analysis concerns the Median Hit Rates among the various QoS levels, as depicted in figure 4.8 below.

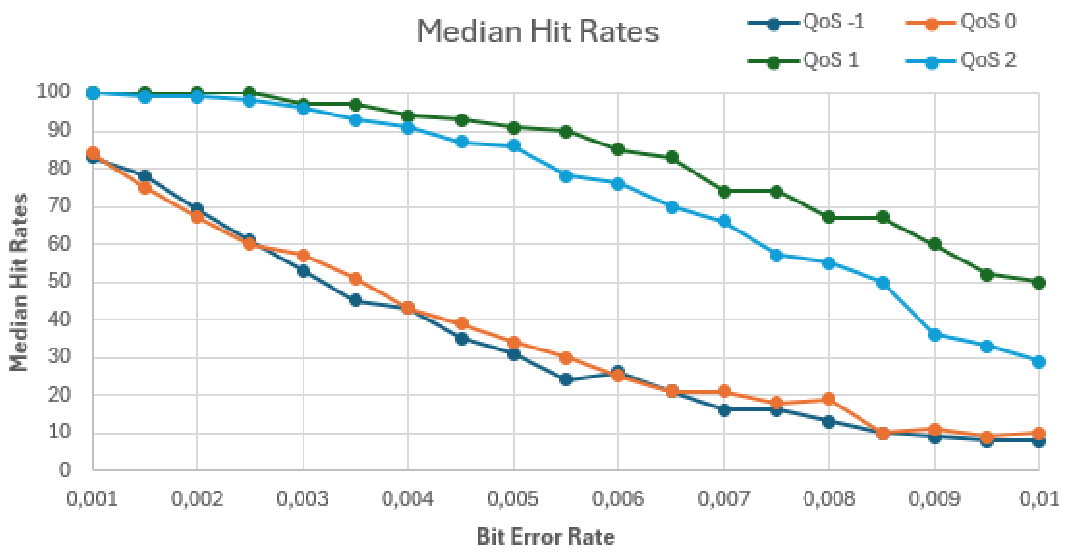


Figure 4.8: QoS Levels: Median Hit Rate Comparison

The plot clearly illustrates that QoS 1 and QoS 2 exhibit a notably high hit rates compared to the other two levels. This behaviour is attributed to the acknowledgment procedure of QoS 1 and QoS 2, which is also supported by the retransmission mechanism.
QoS -1 and QoS 0 demonstrate similar performances, with QoS 0 slightly better due to the establishment of a connection setup.
Analysing the plot, it is evident that QoS 1 performs better than QoS 2 in terms of hit rates. The gap is evident for increasing BER values. An explanation for this characterization is that QoS 2, for a single acknowledgment, requires more message exchanges. Therefore, with the same number of retransmissions, it is subject to more errors compared to QoS 1. On the other hand, QoS 1 can have duplicates while in QoS 2 the uniqueness of packets is guaranteed. Duplicates and retries will be discussed next.
To conclude, the increment of BER presents drastic reductions in QoS -1 and QoS 0, while hit rates for QoS 1 and QoS 2 are more resilient, with curves decreasing more smoothly.

The analysis of the two metrics demonstrates that there is a trade-off between each feature. QoS -1 and QoS 0 are suitable for applications that require responsiveness as a priority, but in contrast, packet losses will be high in error-prone environments. On the other hand, QoS 1 and QoS 2 are appropriate to manage applications that have reliability as key requirement, despite the reactivity loss. Moreover, QoS 2 is crucial for sensitive applications in which delivery only once is a key need without duplicates upon receipt.

4.3.4. QoS 1 Retries

A second objective is to explore the QoS behaviour using another parameter that concerns the system as a whole. In addition to the use of BER, the number of retransmissions per packet is introduced as a variable.
To cover every QoS aspect, including also the duplicates, QoS 1 was chosen, analyzing it with the metrics already mentioned. Moreover, for further insights, other indicators such as the total number of retransmissions by publishers, broker and subscribers are introduced.

First, QoS 1 Median Delays among different retry values is shown below in figure 4.9.

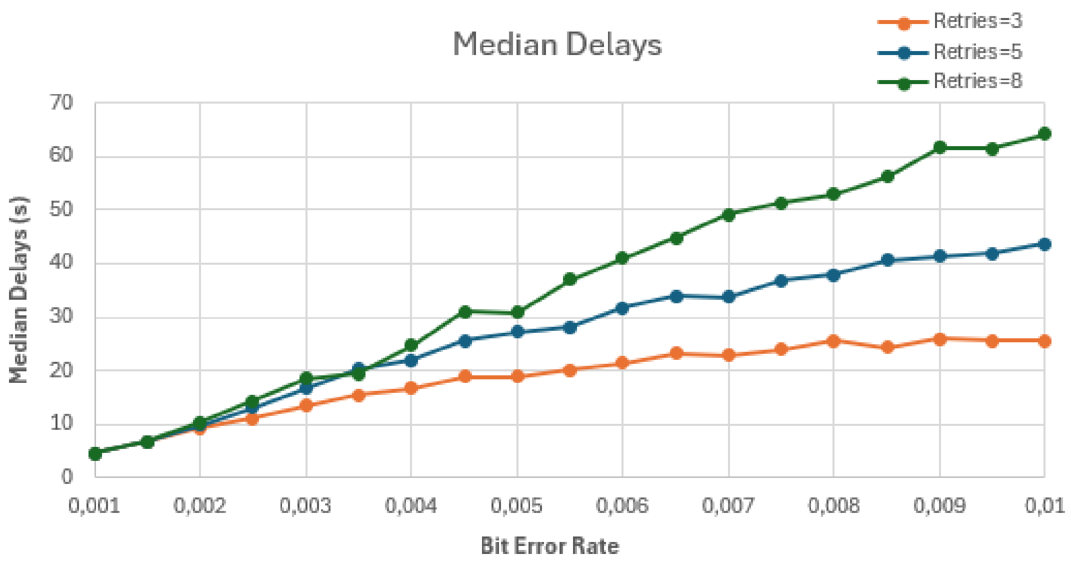


Figure 4.9: Retries: QoS 1 Median End-to-End Delay Comparison

As it is possible to notice, for each BER value, the increase of retransmissions per packet implies higher delays. This is due to the fact that the availability of retries is simply greater. Therefore, the system makes multiple attempts before reaching its capacity. The described behaviour is confirmed by the figure above, observing that after a certain increase the curves tend to stabilize.

A second observation can be made for the QoS 1 Median Hit Rates shown in figure 4.10.

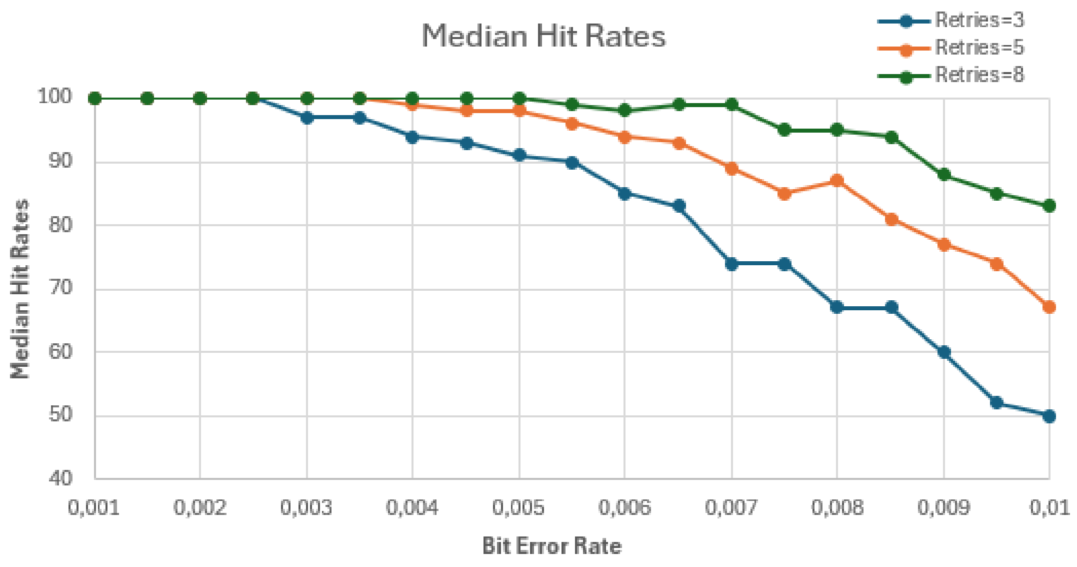


Figure 4.10: Retries: QoS 1 Median Hit Rates Comparison

The plot illustrates that, for a given BER, QoS 1 hit rates are better for larger values of packet attempts. Indeed, the slopes of the curves are smaller for a higher number of retransmissions, indicating greater resilience to errors.

A particular attention will be on the total number of received duplicates among all subscribers. Figure 4.11 below shows the QoS 1 Median Received Duplicates and a comparison with different packet retries is made.

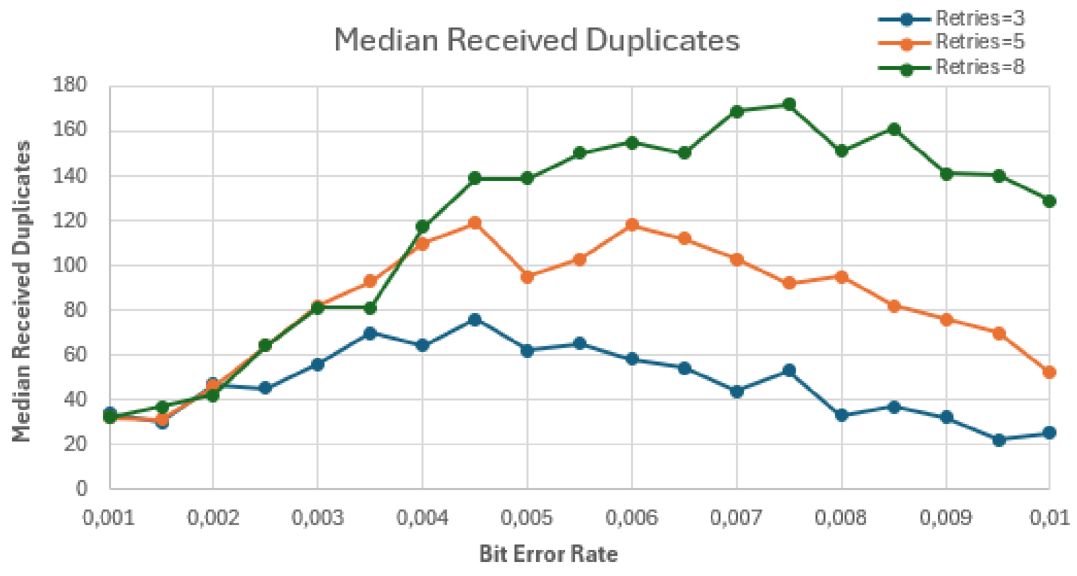


Figure 4.11: Retries: QoS 1 Median Duplicates Comparison

As expected, by increasing the retransmissions per packet, the total received duplicates

increases. Indeed, the number of duplicates in general reaches a peak and once a certain BER has been reached the curves begin to show a slight stabilization and finally a gradual decrease. This behaviour indicates a saturation of retransmissions at very high BERs and the system begins to lose not only original packets but also duplicates.
Finally, to further explore this observation, the total number of attempts made by publishers, broker, and subscribers is calculated. Figures 4.12, 4.13 and 4.14 show the cumulative retries for the three types of entities of the protocol.

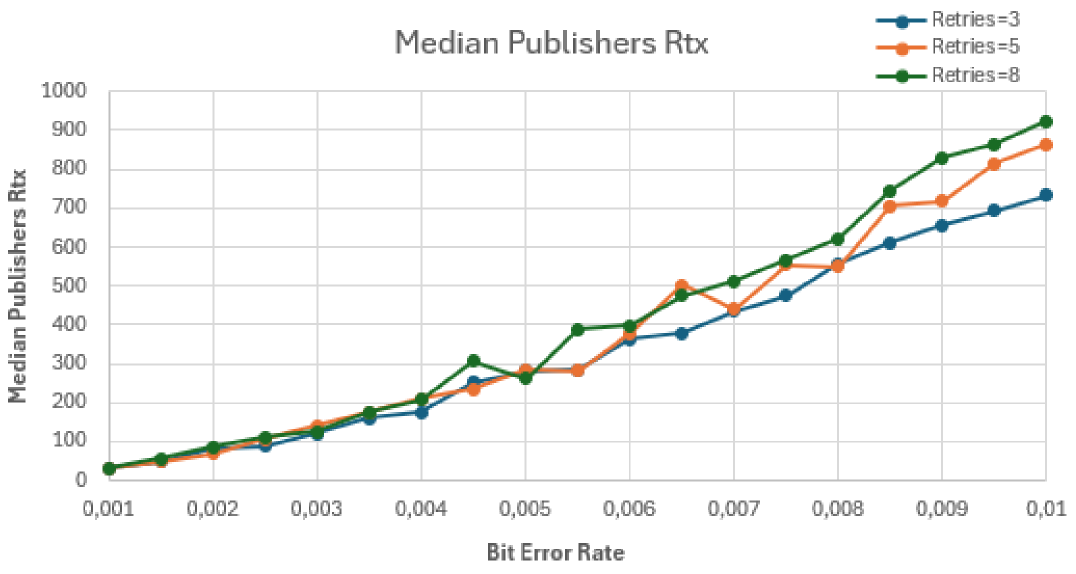


Figure 4.12: Retries: Median Publishers Retransmissions Comparison

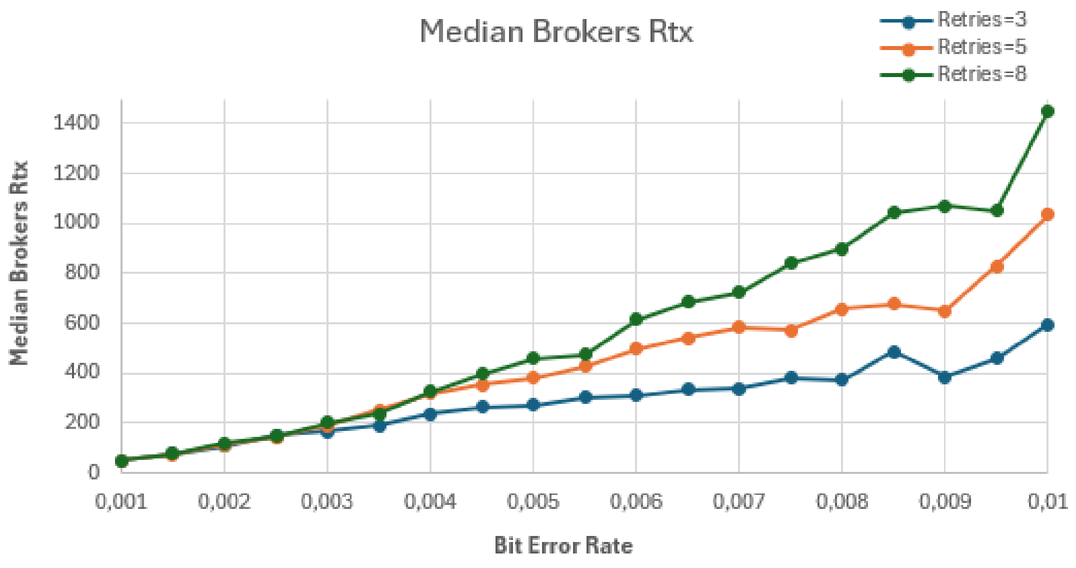


Figure 4.13: Retries: Median Broker Retransmissions Comparison

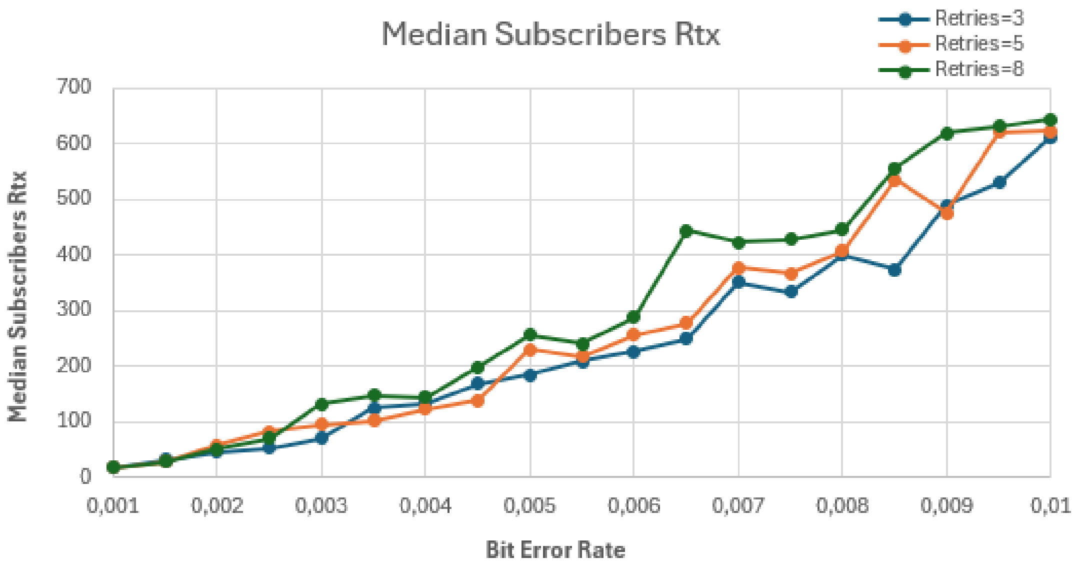


Figure 4.14: Retries: Median Subscribers Retransmissions Comparison

As can be seen from the figures shown above, the curves representing the number of retransmissions show a significant increase, particularly for high BER values, thus confirming the previous considerations.

5 | Conclusion and Future Work

The main objective of this thesis was to correctly implement MQTT-SN according to the specification requirements, utilizing the simulation environment provided by OMNeT++ and to study the Quality of Service behaviour under various network conditions.

The first part of the project is dedicated to an overview of related work in the IoT context with the aim of describing the comparison of two protocols and ultimately providing insights into QoS. In the second part, an in-depth analysis of the methodology adopted is discussed. Starting from the description of the fundamentals of MQTT-SN, the chapter provides basic information on the discrete event simulator utilized, also exploring the integrated INET framework. Finally, the implementation section describes the approaches used to build the overall system. The last part delves into the experiments conducted after the development of the protocol was concluded. In particular, two main performance metrics are introduced with the aim of describing QoS behaviours. Then, an error model is used to represent overall network errors within specific scenarios. In conclusion, the section provides a description of setting parameters and running the simulation and then concludes with focused comparisons. The first comparison between different QoS levels and their respective suitable applications. The second explores QoS behaviour by modifying another system parameter, namely retries per packet.

From the results of the simulations it is possible to see that, in a wireless network scenario, the choice of QoS is strictly linked to the application requirements. In fact, lower QoS levels are suitable for reactive implementations, despite the packet losses that can occur in error-prone environments. On the other hand, higher QoS levels are better for applications that need reliability as a key requirement, despite the loss of responsiveness. Analysis with key metrics demonstrates this trade-off between these features.
Per-packet retransmission is another key aspect to consider for dealing with faulty networks. From the observed results, in fact, it is clear that the protocol requires a balance of this parameter also considering the application requirements in this case.

The potential future opportunities for studies that can be related to this implementation are the following:

- Adaptive QoS: to produce a reliable and efficient system capable of adapting the MQTT-SN Quality of Service and to improve packet transmission
- Improved retransmission mechanism: to adjust retries per packet parameter based on network conditions and target performance

Bibliography

[1] Fabio Palmese, Edoardo Longo, Alessandro E. C. Redondi, and Matteo Cesana. CoAP vs. MQTT-SN: Comparison and Performance Evaluation in Publish-Subscribe Environments. In *Proceedings of the 2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, New Orleans, LA, USA, 2021. IEEE.

[2] Andy Stanford-Clark and Hong Linh Truong. MQTT-SN Version 1.2. Protocol Specification, OASIS, November 2013.

[3] András Varga. OMNeT++ Discrete Event Simulator. `https://omnetpp.org`, 2022.

[4] INET Framework Community. INET Framework for OMNeT++ Simulations. `https://inet.omnetpp.org`, 2022.

[5] R. Khalili and K. Salamatian. A new analytic approach to evaluation of packet error rate in wireless networks. In *3rd Annual Communication Networks and Services Research Conference (CNSR'05)*, 2005.

List of Figures

3.1 Transparent and Aggregating Gateways . . . 7
3.2 MQTT-SN Architecture . . . 7
3.3 Quality of Service 0 . . . 12
3.4 Quality of Service 1 . . . 13
3.5 Quality of Service 2 . . . 14
3.6 QoS Level 2 from Broker to Subscriber . . . 15
3.7 QoS Level 0 from Broker to Subscriber . . . 15
3.8 OMNeT++ Model . . . 18
3.9 NED Network Topology . . . 19
3.10 OMNeT++ Simulator . . . 20
3.11 Network Scenario with INET . . . 21
3.12 Project Structure . . . 22
3.13 MQTT-SN Application Structure . . . 23
3.14 Wireless Network Topology . . . 24
3.15 C++ Classes for MQTT-SN Application . . . 25

4.1 Publisher Items in JSON Format . . . 34
4.2 Subscriber Items in JSON Format . . . 35
4.3 QoS 1 Median Delays Table . . . 37
4.4 QoS 1 Median Hit Rates Table . . . 37
4.5 Median Hit Rates Table for Nretry=5 . . . 38
4.6 Median Broker Retransmissions Table for Nretry=5 . . . 38
4.7 QoS Levels: Median End-to-End Delay Comparison . . . 39
4.8 QoS Levels: Median Hit Rate Comparison . . . 40
4.9 Retries: QoS 1 Median End-to-End Delay Comparison . . . 41
4.10 Retries: QoS 1 Median Hit Rates Comparison . . . 42
4.11 Retries: QoS 1 Median Duplicates Comparison . . . 42
4.12 Retries: Median Publishers Retransmissions Comparison . . . 43
4.13 Retries: Median Broker Retransmissions Comparison . . . 43
4.14 Retries: Median Subscribers Retransmissions Comparison . . . 44

List of Tables

3.1 General Message Format 8
3.2 MQTT-SN Message Types 8
3.3 CONNECT Message Format 9
3.4 PUBLISH Message Format 9
3.5 SUBSCRIBE Message Format 10

Acknowledgements

Vorrei ringraziare il Prof. Alessandro Redondi per la sua disponibilità nel chiarire i vari dubbi e per la flessibilità nelle tempistiche.
Ringrazio la mia famiglia per avermi sempre sostenuto in questo percorso intrapreso.
Infine ringrazio tutti gli amici che mi sono stati vicini e su cui ho potuto contare.

Grazie mille.