Machine-to-machine (M2M) communication using MQTT-SN over an heterogeneous WSN infrastructure

KAMARATAKIS GEORGIOS

B.A. Computers Engineering in Technological Educational Institute of Piraeus, 2014

A Thesis

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN INFORMATICS ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SCHOOL OF ENGINEERING

HELLENIC MEDITERRANEAN UNIVERSITY

CRETE

MARCH 2022

Approved by:

Associate Professor

PANAGIOTAKIS SPYROS

Abstract

The major protocols used for M2M communication are MQTT, CoAP, OMA LWM2M. In short, these protocols target devices that have to conserve power so that they can operate for a long time on batteries. In M2M/IoT protocols, the payload is very small compared to the internet protocols where the payload is heavy and the headers accordingly large. For a predefined, pre-planned WSN, the static allocation of sensor nodes and gateways is appropriate. The more sensor nodes we have, the more we need gateways to accomplish the whole scenario based on IoT.

With the exception of very few TCP/IP based WSNs (e.g. WLAN), where IP addresses are used for addressing the nodes, most WSNs adopt their own addressing scheme to accomplish inter-nodal communication. This feature makes the sensor networks complicated to expand and to interact with other WSN of different technology or the rest of the TCP/IP world. To this end, we can make use of MQTT protocol. MQ Telemetry Transport (MQTT) is an open source publish/subscribe messaging protocol designed for power constrained devices and low-bandwidth, high-latency networks. MQTT creates multicast groups of subscribers based on the so-called “topics” they are registered to, and by that way the issue of communication between heterogeneous networking infrastructures can be solved at Application Layer.

However, MQTT is TCP/IP based. For a successful MQTT implementation there is the need of an intermediate broker keeping TCP sockets alive for continuous submission & receipt of data with the nodes participating in each Topic. This constrains its applications to TCP/IP-based WSNs. In this scope, the MQTT-SN (MQTT for Sensor Networks**)** protocol has been proposed. MQTT-SN is an optimized version of MQTT, designed specifically for efficient operation in large low-power IoT Sensor Networks (SN). It uses the same publish/subscribe model and hence it can be considered as a version of MQTT. However, it uses UDP as the transport protocol and introduces MQTT-SN Gateways as intermediaries between different WSNs. Currently, very few deployments of MQTT-SN exist.

Aim of this thesis is to: a) Investigate how we can interconnect various WSN and achieve a common communication infrastructure by implementing MQTT-SN, b) study the evaluation tests of this implementation, c) develop a system that can easily fit any heterogeneous IoT deployment. In general, we are interested in confirming the results by other researchers concerning MQTT-SN for better performance.

Keywords: MQTT-SN , MQTT , heterogeneous wsn , communication , Publish/subscribe system , forwarding , RSMB

Table of contents

List of Figures

List of Tables

Chapter One - Introduction

1. Introduction

Machine-to-machine, or M2M, is a general term that refers to any technology that allows networked devices to communicate data and conduct operations without the need for human intervention. Machine-to-machine technology's main goal is to collect sensor data and communicate it to a network. M2M communications were originally built on the "telemetry" idea, which involved remote devices and sensors gathering and sending data to a central location for analysis. Instead of using radio signals to send data, M2M communication now uses public networks to lower total costs such as cellular or Ethernet. One of the most important features of M2M communications is the use of wireless sensors to send telemetry data. Sensors, a wireless network, and a linked computer are the core M2M tools that make data centralization and analysis possible. The data is then translated by the system, which triggers preprogrammed, automated responses to the scenario.

The Internet of Things (IoT) is formed from devices and smart objects who have been assigned a distinctive identifier and are connected to the internet over wireless networks. These devices typically have limited resources, such as battery capacity, memory, and processor power, etc. This section provides an overview of the present state of the field of communication on heterogeneous wireless networks.

First let’s start with the most common wireless communication protocols that IoT devices are using at the moment. These protocols are Wi-Fi, Bluetooth, ZigBee, 6LowPan, LoRa and nRF. Each protocol has its pros and cons, they support different topologies and of course they have different transport medium so they are incompatible with each other in so many ways.

As described, different types of machines might use different ways to communicate, making it hard for them to understand each other. To solve this problem, there is something called a "bridge" that translates these different ways of communicating into a common format, so that all the machines can understand each other.

This common format is called User Datagram Protocol (UDP) packets, and the "bridge" sends these packets using a protocol called MQTT-SN (MQTT for Sensor Networks). This protocol is designed especially for M2M communication and works well even in networks that have slow internet speeds or are unreliable.

In short, the "bridge" and MQTT-SN make it possible for machines to talk to each other easily and efficiently, even if they use different types of wireless technologies. This is because the packet size and in general the whole protocol it is designed to be fit in any wireless technology.

MQTT-SN is especially helpful because it is a lightweight protocol, meaning it doesn't require a lot of resources to use. This makes it a good choice for devices with limited resources, such as sensors and IoT devices. So, in conclusion, the "bridge" and MQTT-SN help make sure that different types of machines can talk to each other easily, making the IoT ecosystem more effective and efficient.

Our approach would be to make use of some wireless network protocols in the way of receive - transmit asynchronously. There will be a central node (having all the physical mediums attached) and the end-nodes (different wireless networks) that they will communicate to the central node. Each technology will make use of their protocols in a way of receive - transmit async to the central node. The central node will have scripts running sending/receiving messages from the end-nodes and forwarding them to the MQTT-SN Gateway and vice versa.

2. Problem Statement

The problem that we are trying to solve is the lack of interoperability between different wireless communication technologies. Each wireless technology uses its own protocols and packet structures, which makes it difficult for machines using different technologies to communicate with each other directly.

While searching on the Internet for an implementation of a common protocol unifying multiple wireless infrastructures, we came up to an article about MQTT-SN implementation of Benjamin Cabe [12] using MQTT-SN protocol to communicate through BLE on MicroBit device to the broker on the Internet. This article made us curious about how this pattern of implementation is really feasible and what about its performances.

The aim of this thesis is to: a) Investigate how we can interconnect various WSN and achieve a common communication infrastructure by implementing MQTT-SN, b) study the performance of this implementation, c) develop a system that can easily fit any heterogeneous IoT deployment. In general, we are interested in confirming the results by other researchers concerning MQTT-SN for better performance [7].

Chapter Two - Related Technologies

1. Wireless Sensor Networks - WSNs

The Wireless Sensor Network (WSN) is an infrastructure-free wireless network that uses an ad-hoc deployment of a large number of wireless sensors to monitor system, physical, and environmental factors. WSN uses sensor nodes in conjunction with an integrated CPU to manage and monitor the environment in a specific area. They are linked to the Base Station, which serves as the WSN System's processing unit.

Applications of WSN:

1. Internet of Things (IoT)
2. Surveillance and Monitoring for security, threat detection
3. Environmental temperature, humidity, and air pressure
4. Noise Level of the surrounding
5. Medical applications like patient monitoring
6. Agriculture

1.1. Bluetooth Low Energy - BLE

The low-power wireless technology called Bluetooth Low Energy (BLE) enables connections between devices. BLE is designed for low-power applications that may operate for a long period running on batteries and uses the 2.4 GHz ISM band. BLE is mostly used for low-bandwidth transfers of small quantities of data across close distances. Unless a connection is made, BLE is always in sleep mode in contrast to Bluetooth, which is constantly active. It thus consumes comparatively little electric power. Depending on the use case, BLE uses around a hundred times fewer energy than Bluetooth. In addition to point-to-point communication, BLE supports broadcast mode and mesh networks. [8,9]

Due to its advantages, BLE is perfect for apps that run on coin cells and often exchange small quantities of data. BLE is widely utilized in the home automation, security, tracking, fitness, and healthcare sectors. [8]

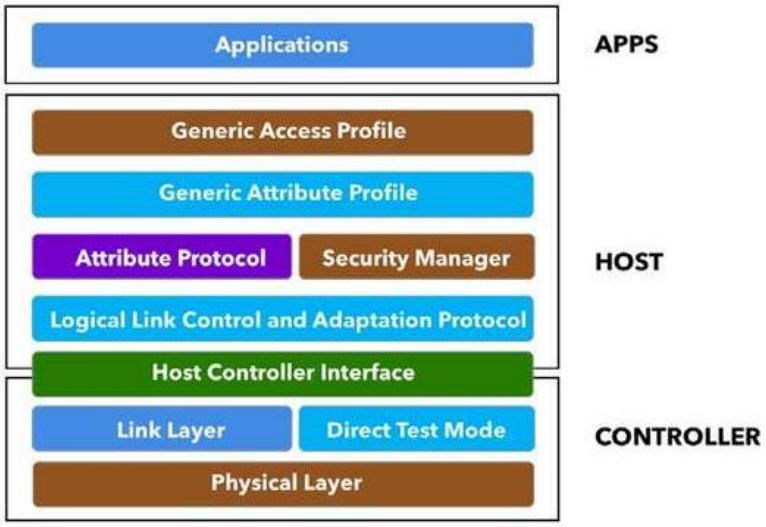


Figure 1. BLE Architecture layers

Image acquired from : https://www.novelbits.io/

The Generic Access Profile (GAP)

The Generic Access Profile (GAP) [9], initializes a foundation for how BLE devices communicate with each other. This consists the following:

- BLE devices functions
- Advertisements (Broadcasting, Discovery, Advertisement parameters, Advertisement data)
- Creating a connections
- Security

The following are the various roles of a BLE device:

- Broadcaster: a device that broadcasts advertisements but does not accept packets or allow other devices to connect to it.
- Observer: a device that listens for other devices putting out Advertising Packets but does not initiate a connection with them.
- Central: a device that searches for and listens to other Advertising devices. A Central can also link to a device that displays advertisements.
- Peripheral: A device that advertises and accepts connections from central equipment.

The Generic ATTribute Profile (GATT)

The primary purpose of connecting two BLE devices is to communicate data between them. A bidirectional data transfer between two BLE devices is impossible without a connection. Which brings us to the GATT notion [9].

The GATT specifies the format in which data from a BLE device is accessible. It also specifies the steps required to gain access to the data disclosed by a device. Within GATT, there are two roles: server and client. The Server is the device that exposes the data it manages or possesses, as well as maybe some other characteristics of its behavior that other devices can influence.

A Client, on the other hand, is a device that communicates with the Server in order to read the Server's disclosed data and/or influence its behavior.

Keep in mind that a BLE device can serve as both a server and a client. Simply expressed, it functions as a Server when presenting its own data and as a Client when accessing the data of another device.

Characteristics and Services

The terms "services" and "characteristics" are arguably the most commonly used in BLE! That's why it's critical to understand them, especially for BLE devices that communicate with one another. For explaining how GATT works, we need to explain a few important notions (including Services and Characteristics):

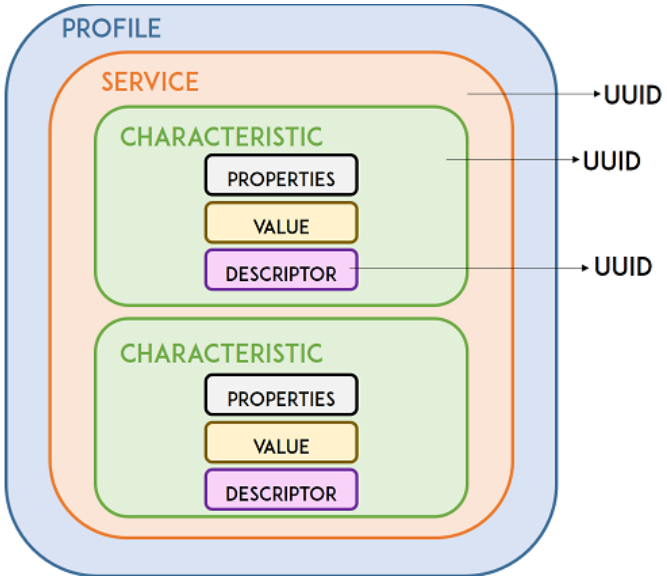


Figure 2. Services and Characteristics in BLE

Image acquired from : https://randomnerdtutorials.com/

Attributes: Any type of information exposed by the server that establishes its structure is referred to as an attribute. Services and Characteristics.

Services: A Service is a collection of one or more Attributes, some of which are Characteristics. Its goal is to group pertinent Attributes that support a certain Server operation. For instance, the Battery Level is a Characteristic in the Battery Service that the SIG accepted.

Characteristics: A characteristic is a fact or piece of data that the server wishes to share with the client and is always a component of the service. For instance, the Battery Level Characteristic shows the amount of battery life left in a device that a client can read.

Profiles: Profiles, in contrast to Services, cover a far wider range of topics. In terms of Services, Characteristics, even Connections and security needs, they are concerned with defining the behavior of the Client and Server. On the other hand, services and their specifications are only focused on the server-side implementation of these services and characteristics.

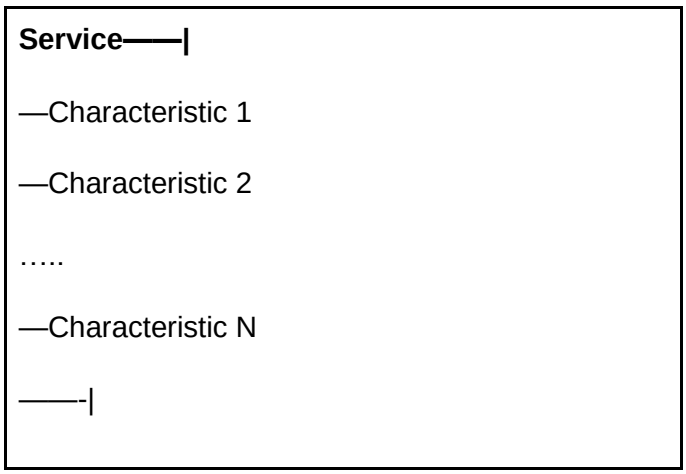


Figure 3. Simplified description of a Service in BLE

In BLE, there are six types of functions on Characteristics:

- **Clients:** send commands (described below) to the server that don't need a response. A command that does not need a response from the server is a write command.
- The server must respond to requests made by clients. Requests include things like read requests and write requests.
- The messages that the server delivers in response to a request are called **responses**.
- **Notifications** are communications that the Server sends to the Client to let them know when a certain Characteristic Value has changed. For this to be triggered and relayed by the Server, the Client must allow Notifications for the Characteristic of Interest. It's important to remember that a Notification doesn't require a Client response to show that they have received it.

- **Indications:** messages that the Server sends to the Client. Similar to notifications, they need a response from the client to let the server know whether they were successfully received. Notifications and Indications are exposed through the Client Characteristic Configuration Descriptor (CCCD) Attribute. By adding a "1" to this attribute's value, notifications are enabled, while indications are activated by providing a "2." When a "0" is input, both notifications and indications are turned off.
- **Confirmations:** The client sends confirmations to the server. These are the packets of acknowledgement that the Client sends to the Server to let it know that the Indication was successfully received by the Client.

BLE Advantages: [9]

- However, the power use is reduced when compared to other low-power technologies. BLE manages to reach the ideal and low power consumption by turning the radio off as frequently as possible and delivering little data at slow transfer rates.
- The published standard documentation can be downloaded for free. Most other wireless protocols and technologies require membership in the official organization or consortium in order to get the specification.
- The retail cost of modules and chipsets is lower than that of other comparable technologies as well.
- Particularly, it is included in the vast majority of smartphones available today.

BLE Limitations: [9]

- Data Throughput: The data throughput of BLE is constrained by the physical radio layer (PHY) data rate, which is the rate at which the radio sends data. Depending on the Bluetooth version being used, the rate changes. For versions of Bluetooth 4.2 and before, the rate is set at 1 Mbps. However, with Bluetooth 5 and beyond, the rate varies depending on the mode and PHY used. As with earlier versions, the rate can be 1 Mbps or, when the high-speed option is engaged, 2 Mbps.
- Range: Because BLE (and Bluetooth generally) were designed for short-range applications, its operational range is constrained. Several reasons restrict the BLE's operating range:
    - BLE runs in the 2.4 GHz ISM range, which is highly influenced by impediments such as metal objects, walls, and water that exist all around us (especially human bodies)
    - The performance and design of the BLE device's antenna.
    - The device's physical container.
    - Orientation of the device
- Internet Connectivity Requires a Gateway: To send data from a BLE-only device to the Internet, another BLE device with an IP connection is required to receive the data and then relay it to another IP device (or to the Internet).

1.2. NRF24 Network

This network is based on a specific transceiver by Nordic the NRF24L01 [11] , in general the NRF24L01 is a wireless transceiver module (works on SPI Protocol), which is used for sending and receiving data at an operating radio frequency of 2.4 to 2.5 GHz ISM band.

- This transceiver module consists of a frequency generator, shock burst mode controller, power amplifier, crystal oscillator modulator, and demodulator.
- When transmitting power is zero dBm it uses only 11.3 mA of current, while during receiving mode, it uses 13.5 mA of current.
- This module is designed for long distance and fast transmission of data.
- It is designed to work through an SPI protocol.
- Air data transmission rate of NRF24L01 is around 2 Mbps.
- Its high air data rate combined with power saving mode makes it very favorable for ultra-low power applications.
- Its internal voltage regulator controls a high-power supply rejection ratio and power supply range.
- This module has a compact size, and can easily be used in confined spaces.
- This module is designed to operate at 3.3 volts.
- This module has an address range of 125 and it can communicate with six other modules. By using this feature, we can use it in mesh networks and other networking applications.

Multiceiver is an operation available on the nRF24L01. Multiple Transmitters Single Receiver is an abbreviation. Each RF channel is conceptually separated into six parallel data channels, which are referred to as Data Pipes.

A data pipe, in other terms, is a logical channel within the physical RF Channel. Each data pipe can be configured and has its own physical address (Data Pipe Address). This can be seen in the Figure 4.

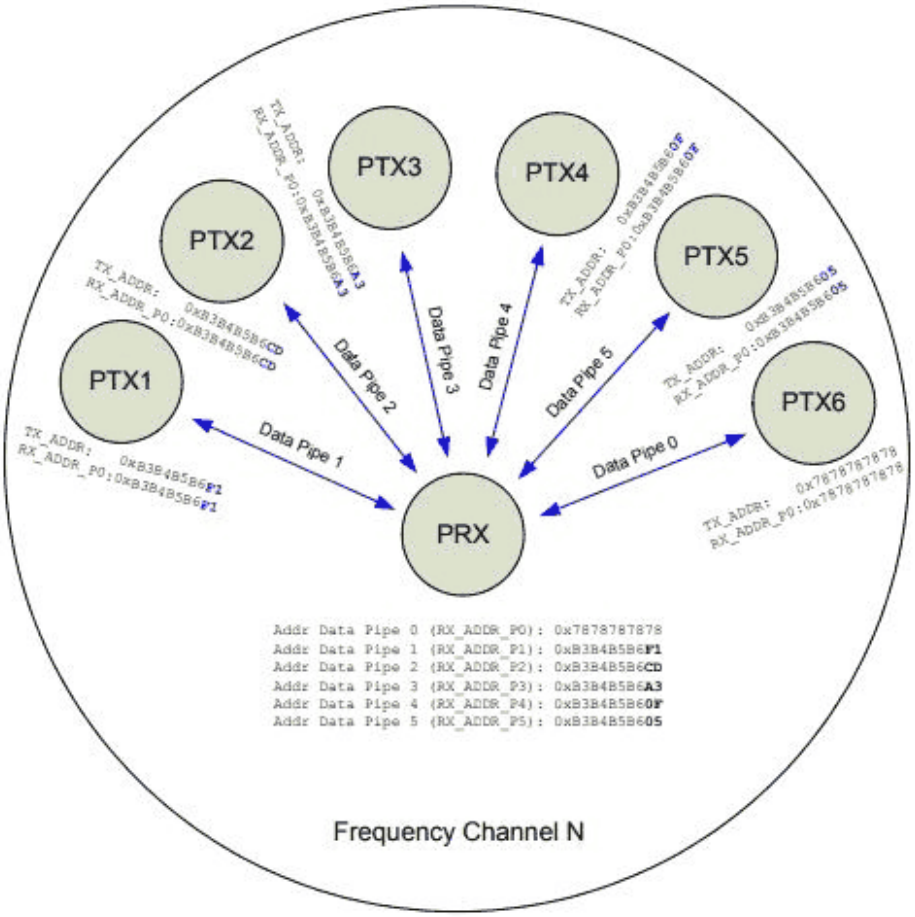


Figure 4. Data pipes in NRF technology

Image acquired from : https://www.electronicwings.com

Upon this basis developers built more advanced libraries bringing features to the community that enable the developers to easily route messages to devices, create meshes and auto-heal networks.

2. Application Layer M2M Protocols

2.1. CoAP

In the Internet of Things, the Constrained Application Protocol (CoAP) is a customized web transfer protocol for usage with constrained nodes and constrained networks. CoAP is a protocol that uses Server/Client pattern (see Figure 5.), allows basic, restricted devices to connect to the Internet of Things via HTTP, even across constrained networks with poor bandwidth and availability. Machine-to-machine (M2M) applications such as smart energy and building automation are common uses. The Internet Engineering Task Force (IETF) created the protocol, which is described in IETF RFC 7252.

CoAP features:

- Web Protocol Used in M2M With Constrained Requirements
- Asynchronous Message Exchange
- Low Overhead
- Very Simple to Perform Syntactic Analysis
- (URI) Uniform Resource Identifier
- Proxy and Caching Capabilities

Based on [17], [18] HTML, HTTP/REST, and URIs are the three technologies that make up the Web. Only the latter two are important in situations when machines communicate with one another. Special data formats, often based on XML and its compact binary representation, EXI, or the JavaScript Object Notation (JSON, RFC 4627), are being defined at [18] to replace HTML in these applications.

Using HTTP to communicate

CoAP would be helpful even if we could only use it to communicate between CoAP end points, but it is only fully realized when combined with HTTP. The REST architectural style makes this possible by using proxies, or more broadly, intermediaries, who act as a server to a client and as a client to another server. (The term "proxy" is reserved in REST nomenclature for intermediates that are specifically set on a client.) It also features a "gateway" that functions as if it were the origin server; they are sometimes referred to as "reverse proxies" on the Web because they are less intrusive than a traditional gateway.) [18]

URIs are used by both CoAP and HTTP to identify resources. CoAP's URI schemes, such as CoAP URIs may be unknown to existing HTTP endpoints. A reverse-proxy intermediary can make a set of CoAP resources available at what appear to be ordinary http:// or https:// URIs, allowing older Web clients to transparently access CoAP servers (see Figure 5). A similar service could be provided by an interception proxy (RFC 3040) placed in a network location suitable for traffic interception that automatically redirects client requests to itself.

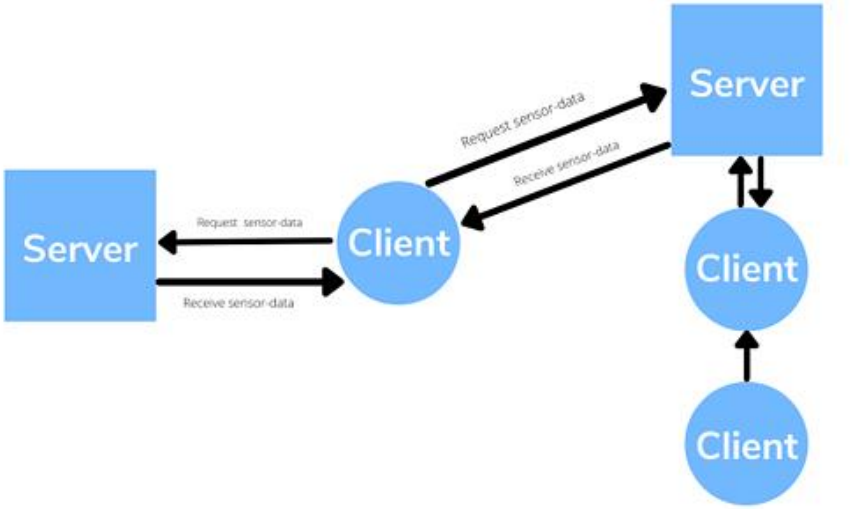


Figure 5. CoAP protocol Architecture schema

Image acquired from : https://www.emnify.com/

2.2. OMA LWM2M

The Open Mobile Alliance (OMA) has created a collection of protocols for machine-to-machine (M2M) or Internet of Things (IoT) device management and communication called Lightweight M2M (LWM2M).

LWM2M uses the CoAP protocol and can be sent over UDP or SMS. The architecture for CoAP is also Server and Client. On a device, LWM2M defines service as an Object and Resource, both of which are represented in an XML file.

The need to monitor and use distant sensors and devices in regions with intermittent connectivity and far from power connections will grow as the Internet of Things becomes more popular. LWM2M offers a standardized solution to manage these devices and communicate telemetry data collected by the sensors to the cloud in a timely and cost-effective manner [19].

LWM2M was created to reduce the amount of power and data used by low-power devices with limited processing capacity and bandwidth. When people or devices are far from a power source and need to use battery-powered local devices with a SIM card , the protocol is ideal [19].

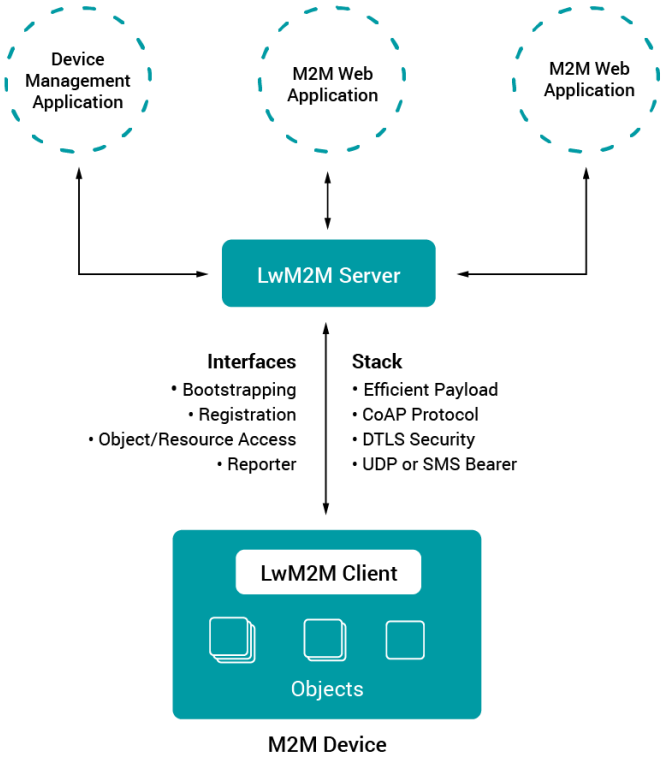


Figure 6. LWM2M protocol Architecture schema

Image acquired from: https://www.avsystem.com/

2.3. Matter

According to the published [29] specification of Matter Protocol is a new and very promising protocol, formerly known as Project CHIP, is an open-source standard aiming to improve compatibility among smart home devices. Supported by tech giants like Apple, Google, Amazon, and Zigbee Alliance, it tackles challenges of device fragmentation in the IoT world.

Matter's main goals are:

- Interoperability: It enables seamless communication between diverse smart devices, irrespective of their manufacturers, for a unified smart home experience.
- Security: Matter prioritizes data security and privacy, implementing advanced security features to ensure safe device communication.
- Reliability: The protocol focuses on strong and consistent connectivity to minimize common device communication issues.
- Scalability: Matter is designed to scale smoothly, accommodating new devices and services as the IoT landscape expands.
- Open Standard: As an open-source initiative, Matter encourages collaboration and innovation among developers to drive wider adoption.

Potential Implications and Applications:

- Enhanced User Experience: Users benefit from simplified setup and usage of smart devices, connecting and controlling different manufacturer's devices seamlessly.
- Faster Development: Developers can create IoT apps and services more easily, leveraging a standardized framework to speed up innovation.
- Reduced Fragmentation: Matter's common communication standard addresses fragmentation, boosting compatibility and decreasing issues.
- Increased Adoption: Backed by industry leaders, Matter could encourage more consumers to embrace smart home tech, assured of device compatibility and security.

- Ecosystem Growth: A standardized protocol could expand the IoT ecosystem, fostering collaboration among manufacturers, developers, and service provider

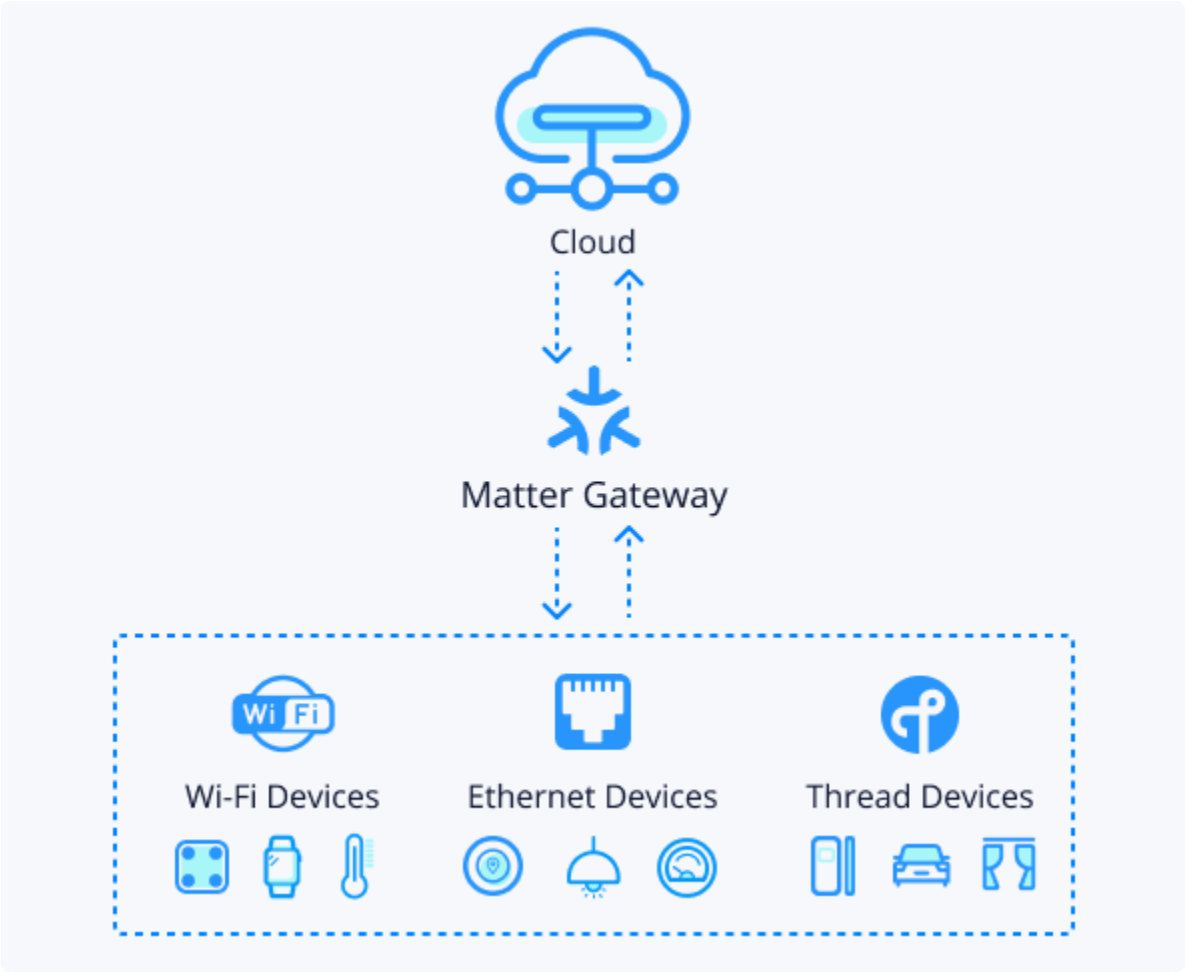


Figure 7. Matter Protocol Architecture schema

Image acquired from: https://www.dusuniot.com

Technically, Matter is a smart home interoperability protocol, it's not an entirely new protocol. Matter is an application layer over existing protocols that utilizes wireless technology based on Internet Protocol (IP), which Wi-Fi routers use to give each connected device an IP address.

Underlying networking technologies used by Matter include Thread, Wi-Fi, Ethernet and BLE. The Matter gateway functions as the local network's brain and controller, connecting smart devices, establishing permissions, and carrying out access control commands. Your Matter gadgets can be connected to other devices using Zigbee, Z-wave, and other protocols through gateway. The matter gateway also offers an internet bridging option.

In smart homes, the matter gateway operates admirably. Smart home appliances like lighting, door locks, and drapes that support several wireless protocols including Wi-Fi and Thread (Thread border router) can be added, reset, and controlled by an APP that supports Matter by using matter gateway.

Being an open-source project, developers can download the matter framework from Github and start developing their own project. After the desired code has been written, developers make command through the Matter framework to build the corresponding hex file based on the compatible-to-matter boards and then flash it.

2.4. MQTT

2.4.1. Introduction

MQTT stands for MQ - Telemetry Transport, MQTT and it is a publish/subscribe message transport protocol for clients and servers. It's light, open, and basic, and it's supposed to be simple to use. These qualities make it excellent for application in a variety of circumstances, including confined environments where a minimal code footprint is required and/or network bandwidth is limited, such as communication in Machine to Machine (M2M) and Internet of Things (IoT) contexts.

Based on [13] Andy Stanford-Clark (IBM) and Arlen Nipper (IBM) created the MQTT protocol in 1999. (Arcom, now Cirrus Link). Several needs were mentioned by the two inventors for the future protocol:

- Implementation is simple.
- Data Delivery Quality of Service
- Lightweight and economical bandwidth usage
- Constant awareness of the session

Based on MQTT specification [15] the protocol runs over TCP/IP and for the communication to happen each publisher and subscriber need to send their message with a specific topic to the broker. The term "topic" in MQTT refers to a UTF-8 string used by the broker to filter messages for each connected client. One or more topic levels make up the topic. A forward slash separates each topic level (topic level separator).

MQTT topics are quite light in comparison to a message queue. Before publishing or subscribing to a topic, the client does not need to create it. Without any prior initialization, the broker accepts any valid subject.

MQTT Packet Format

A 2-byte fixed header, a changeable header, and a payload make up a MQTT packet. The first two bytes of the fixed header will always be present in all packets, whereas the variable header and payload will not always be there.

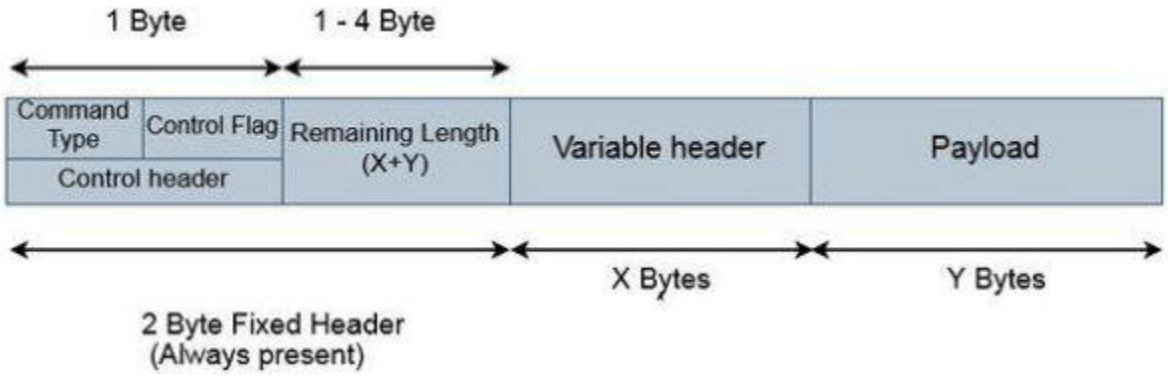


Figure 8. MQTT Packet Format

Figure acquired from: https://openlabpro.com/

The packets available for the communication are shown on Figure 9:

Message type	Value	Description
Reserved	0	Reserved
CONNECT	1	Client connect request to server or broker
CONNACK	2	Connect request acknowledgment
PUBLISH	3	Publish message
PUBACK	4	Publish acknowledgment
PUBREC	5	Publish receive
PUBREL	6	Publish release
PUBCOMP	7	Publish complete
SUBSCRIBE	8	Client subscribe request
SUBACK	9	Subscribe request acknowledgment
UNSUBSCRIBE	10	Unsubscribe request
UNSUBACK	11	Unsubscribe acknowledgment
PINGREQ	12	PING request
PINGRESP	13	PING response
DISCONNECT	14	Client is disconnecting
Reserved	15	Reserved

Figure 9. MQTT Message types

Figure acquired from : https://bytesofgigabytes.com/

2.4.2. QoS

The MQTT protocol has a feature called Quality of Service (QoS). The option to choose a quality of service that suits the client's network reliability and application logic is provided by QoS. QoS makes communication in unpredictable networks a lot easier because MQTT manages message retransmission and assures delivery (even when the underlying transport is problematic). [16]

The agreement between a message sender and receiver that details the delivery guarantee for a particular message is known as the Quality of Service (QoS) level. There are three levels of QoS in MQTT:

- At most once (0)
- At least once (1)
- Exactly once (2)

When discussing QoS in MQTT, it's important to remember the two sides of message delivery:

1. Message delivery to the broker from the publishing client.
2. Message delivery from the broker to the client who has subscribed.

QoS 0 - at most once

QoS has a minimum level of 0. The best-effort delivery is ensured by this service level. There is no guarantee that your order will arrive. The recipient does not acknowledge the communication, and the sender does not retain and resend it. Since QoS level 0 provides the same guarantees as the TCP protocol, it is sometimes referred to as "fire and forget."

Figure 10. QoS 0

Image acquired from: https://www.hivemq.com/

QoS 1 - at least once

A message will be delivered to the intended recipient at least once, according to Level 1 QoS. Until the sender gets a PUBACK packet from the recipient indicating receipt of the message, the message is stored. A message may be sent or sent several times.

The sender uses the packet's identification is included in each packet to associate the PUBLISH packet with the associated PUBACK message. If the sender does not get a PUBACK packet within a reasonable amount of time, the PUBLISH packet is resent. The receiver can respond immediately to a transmission with QoS 1. For instance, if the receiver is a broker, the broker sends the message to every subscriber before returning with a PUBACK packet.

PUBACK packet
Contains : msgId

A duplicate (DUP) flag is set if the publishing client delivers the message twice. This DUP flag is solely used for internal purposes in QoS 1, and neither the broker nor the client processes it. Regardless of the DUP flag, the message's receiver sends a PUBACK.

Figure 11. QoS 1

Image acquired from: https://www.hivemq.com/

QoS 2 - exactly once

The highest level of service in MQTT is QoS 2. This level makes sure that each communication is only delivered once to the designated recipients. QoS 2 is the slowest and safest quality of service level. The assurance is provided via at least two request/response flows (a four-part handshake) between the sender and the recipient. The sender and recipient utilize the packet number from the initial PUBLISH message to coordinate message delivery.

When a receiver receives a QoS 2 PUBLISH packet from a sender, it analyzes the message and sends a PUBREC packet back to the sender, acknowledging the PUBLISH packet. If the sender does not receive a PUBREC packet from the receiver, the PUBLISH packet is sent again with the duplicate (DUP) flag until the sender receives an acknowledgement.

PUBREC packet
Contains : msgId

When the sender gets a PUBREC packet from the recipient, the first PUBLISH packet can be safely discarded. The sender saves the receiver's PUBREC packet and replies with a PUBREL packet.

PUBREL packet
Contains : msgId

After receiving the PUBREL packet, the receiver can discard any previously stored states and respond with a PUBCOMP packet (the same is true when the sender receives the PUBCOMP). The receiver keeps a reference to the packet identification of the original PUBLISH packet until it finishes processing and transmits the PUBCOMP packet back to the sender. This step is necessary to prevent the message from being processed a second time. When the sender gets the PUBCOMP packet, the published message's packet identity is made accessible for reuse.

PUBCOMP packet
Contains : msgId

Figure 12. QoS 2

Image acquired from: https://www.hivemq.com/

Both sides are assured that the message has been delivered after the QoS 2 flow is through, and the sender has obtained delivery confirmation.

If a packet is lost along the route, it is the sender's responsibility to resend the message in a timely manner. It doesn't matter if the sender is a MQTT broker or a client. Each command message requires an appropriate response from the recipient.

2.4.3. Broker

An MQTT broker is an intermediary entity that enables MQTT clients to communicate. Specifically, an MQTT broker receives messages published by clients, filters the messages by topic, and distributes them to subscribers.A MQTT broker is the main software entity in the MQTT architecture. It operates similarly to a real estate broker, who investigates the backgrounds of all parties before starting a deal and after making sure that all applicable laws are adhered to.

The only difference is that MQTT brokers manage message transactions rather than monetary ones. According to [14], MQTT brokers enable exchanges between MQTT clients:

- Enables devices (also known as "client devices" or "clients") to seek a connection.
- Authenticate the devices using the connecting device's shared connection informations
- Make sure that the device can securely send and receive messages to and from other devices using Transport Layer Security (TLS) encryption once it has been authorized (as one option)
- The messages are saved on the server in order to be re-sent in case of an unintentional connection loss, on client-connect, client-disconnect, or any other scenario.

The whole architecture may be grown without having an impact on current client devices since a MQTT broker permits isolated communication between devices (clients). The MQTT broker, a single entity that does all of the heavy lifting, means that the client devices only need to perform little processing with limited bandwidth.

2.5. MQTT-SN

2.5.1. Introduction

There are numerous competing M2M and IoT protocols jostling for attention in the realm of M2M and IoT protocols. These protocols are meant to be lightweight for low-power devices in order to take advantage of the M2M world's low bandwidth limits. MQTT-SN is one such protocol that is built specifically for low-power M2M devices. The suffix SN denotes that this protocol is intended for use in sensor networks. The fact that existing Internet-based protocols' headers and footers have a substantial overhead is taken into account when building the MQTT-SN protocol.

When people hear the terms MQTT and MQTT-SN, they frequently become perplexed. They are not the same. MQTT is an M2M protocol that was supposed to be lightweight, although it requires TCP-IP to function. Despite the fact that MQTT is a light-weight protocol, which it is, it is not ideal for sensors and devices that do not have their own TCP-IP stack. TCP is an expensive protocol that ensures, among other things, time to live for packets, retry methods, large payloads, and other features that are not required for some M2M networks.

As a result, the MQTT-SN was born. The MQTT-SN was designed exclusively for sensor networks and does not require TCP-IP to function. It can operate over any transport layer such as BLE, LoRa, ZigBee etc.

Differences between MQTT and MQTT-SN [5]

The following are the primary differences between MQTT-SN and MQTT:

- In MQTT-SN, there are three CONNECT messages in contrast to one in MQTT. The extra two are used to expressly carry the Will topic and message.
- MQTT-SN can be used with both a simple media and UDP.
- Short, two-byte long topic ID messages replace subject names. This is to help with wireless network bandwidth issues.
- Without registering, you can utilize pre-defined topic IDs and short subject titles. Both the client and the server must use the same topic ID to access this feature. Topic titles that are only a few words long can be included in the PUBLISH message.
- A discovery approach is established to help clients find what they're looking for.
- The semantics of a "clean session" are extended to the Will feature, which means that not only the client's subscriptions, but also the Will topic and message, are permanent. During a session, a client can also change the Will topic and message.
- For the support of sleeping clients, a new offline keep-alive process has been defined. Battery-operated devices can enter a sleeping mode with this process, during which any communications destined for them are buffered at the server/gateway and given to them later when they wake up.

2.5.2. Gateway

MQTT-SN Architecture [5]

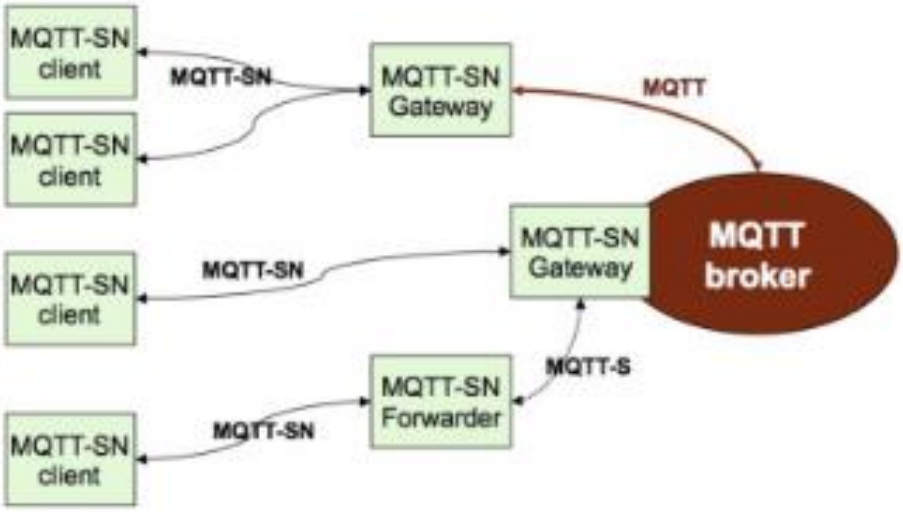


Figure 13. MQTT-SN Architecture

Figure 13. depicts the MQTT-SN architecture. MQTT-SN clients, MQTT-SN gateways (GW), and MQTT-SN forwarders are the three types of MQTT-SN components. MQTT-SN clients use the MQTT-SN protocol to connect to a MQTT server over a MQTT-SN GW. A MQTT-SN GW can be integrated with a MQTT server or not. The MQTT protocol is utilized between the MQTT server and the MQTT-SN GW in the case of a standalone GW. The translation between MQTT and MQTT-SN is its primary function. If the GW is not directly connected to their network, MQTT-SN clients can use a forwarder to connect to it.

The forwarder merely encapsulates the MQTT-SN frames it receives on the wireless side and delivers them to the GW unaltered; in the other direction, it decapsulates the frames it receives from the gateway and sends them to the clients, also unchanged. We may distinguish between two sorts of GWs, transparent and aggregating, based on how a GW conducts protocol translation between MQTT and MQTT-SN (see Fig. X). The parts that follow will clarify them.

A Transparent Gateway will establish and maintain a MQTT connection to the MQTT server for each connected MQTT-SN client. This MQTT connection is only used for end-to-end, near-transparent message exchange between the client and the server. Between the GW and the server, there will be as many MQTT connections as MQTT-SN clients connected to the GW. Between the two protocols, the transparent GW will conduct a "syntax" translation. Because all

message exchanges between the MQTT-SN client and the MQTT server are end-to-end, the client may access all of the server's operations and features.

An Aggregating GW will only have one MQTT connection to the server, rather than having one for each connected client. The GW is the endpoint for all message exchanges between a MQTT-SN client and an aggregating GW. The GW then decides which data should be passed on to the server.

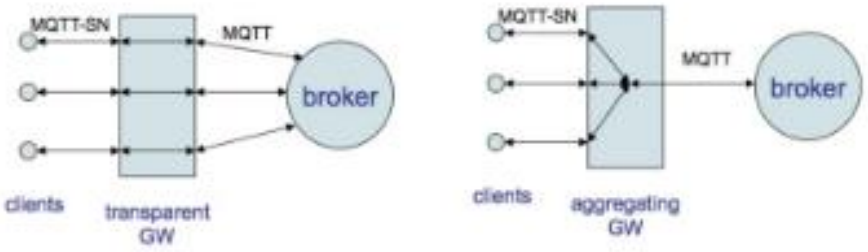


Figure 14. MQTT-SN Types of Gateways

Message Format

Message Header (2 or 4 octets)	Message Variable Part (n octets)

Figure 15. Message Format

Length (1 or 3 octets)	MsgType (1 octet)

Figure 16. Message header format

Based on the MQTT-SN Protocol specification [5], the available packets are described on Figure 17. :

MsgType Field Value	MsgType	MsgType Field Value	MsgType
0x00	ADVERTISE	0x01	SEARCHGW
0x02	GWINFO	0x03	reserved
0x04	CONNECT	0x05	CONNACK
0x06	WILLTOPICREQ	0x07	WILLTOPIC
0x08	WILLMSGREQ	0x09	WILLMSG
0x0A	REGISTER	0x0B	REGACK
0x0C	PUBLISH	0x0D	PUBACK
0x0E	PUBCOMP	0x0F	PUBREC
0x10	PUBREL	0x11	reserved
0x12	SUBSCRIBE	0x13	SUBACK
0x14	UNSUBSCRIBE	0x15	UNSUBACK
0x16	PINGREQ	0x17	PINGRESP
0x18	DISCONNECT	0x19	reserved
0x1A	WILLTOPICUPD	0x1B	WILLTOPICRESP
0x1C	WILLMSGUPD	0x1D	WILLMSGRESP
0x1E-0xFD	reserved	0xFE	Encapsulated message
0xFF	reserved		

Figure 17. List of actions and MsgType values

DUP (bit 7)	QoS (6,5)	Retain (4)	Will (3)	CleanSession (2)	TopicIdType (1,0)

Figure 18. Flags section

Length (octet 0)	MsgType (1)	Flags (2)	TopicId (3-4)	MsgId (5-6)	Data (7:n)

Figure 19. Publish packet

2.5.3. Clients

Clients are the end-nodes in the MQTT-SN network, clients can be subscribers and/or publishers as well.

2.5.4. Forwarders

A forwarder can be a physical device or part of a device, let's say a script on that device. The forwarder does the job of “translating” a packet on a given transport layer to another and vice versa. It is often called a bridge also.

2.5.5. Why to choose MQTT-SN

MQTT SN is a protocol created specifically for sensor networks. MQTT protocol and MQTT SN (MQTT for sensor networks) are two of the most used IoT protocols for creating IoT devices. This blog is for developers who want to learn when to use MQTT-SN (SN MQTT for sensor) and why it's better than MQTT message protocol.

Advantages

Auto-discovery MQTT SN

Agents must be informed of the broker's location in the MQTT setup. The end user's configuration overhead is increased as a result of this. However, with the MQTT-SN protocol, sensors and gateways can send messages that are understood by their counterparts and establish connections to communicate with one another. This makes it a lot easier to set up.

Bandwidth Reduction

In the MQTT-SN, the size of each packet that is transferred has been redesigned. Only the required parameter is sent in the CONNECT, for example. WILL and WILL Message have been separated into different packets and only sent when necessary. To save bandwidth, the total amount of data carried across the network is lowered to a greater extent. MQTT SN also supports four different QoS (Quality of Service) levels: 0, 1, 2, 1, and 3.

Topic IDs and Topic Names that have been pre-defined.

The topic names, as well as the topic IDs, can be predefined in the MQTT SN Gateway. The client does not need to use the topic names and can send packets directly using the ID. The topic IDs are limited to two bytes. Without topic IDs, short subject names of less than 2 bytes can be utilized. If the client wants to utilize a different topic, they can use the register command.

Processing Power is Reduced

The power required to create and communicate data is greatly reduced due to the reduction in packet size. There are further provisions such as Client Sleep, which prevents the gateway from sending or publishing messages to this client in the future. To get all of the packets received during the sleep phase, the client might send a resume message. This makes the publish subscribe messaging protocol ideal for sensors that are powered by batteries.

Connectionless

MQTT is a connectionless protocol that uses the TCP/IP protocol. TCP has a lot of connections overhead that aren't needed in MQTT-SN, which uses UDP. It is also independent of TCP/IP networks. This reduces the amount of data transfer and the amount of power consumed once again.

Medium Independence

In addition to wired and wireless sensor networks, it can be transmitted over Zigbee, Z-Wave, and Bluetooth. MQTT SN is primarily intended for embedded devices that communicate over non-TCP/IP networks such as ZigBee.

Disadvantages

- You need some sort of gateway, which is nothing else than a TCP or UDP stack moved to a different device. This can also be a simple device (e.g.: Arduino Uno) just serving multiple MQTT-SN devices without doing other jobs.
- MQTT-SN is not well supported.

Chapter Three - Approach design

1. Proposed system architecture

We plan to implement wireless communication using receive - transmit async pattern, where there will be a central node that will have all the communication mediums attached to it. The end-nodes, which are different wireless networks, will communicate with the central node using async receive - transmit. The central node will run scripts to send and receive messages from the end-nodes and forward them to the MQTT-SN Gateway or vice versa. Each technology will use its own protocols in the form of async receive - transmit to communicate with the central node.

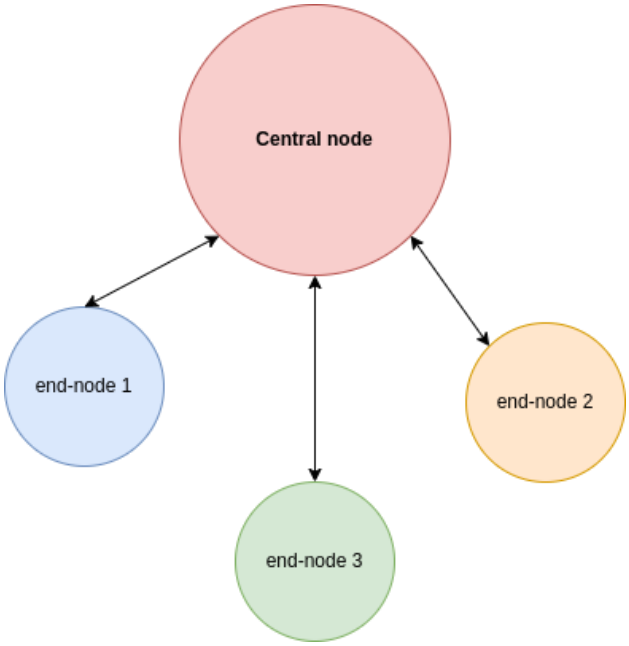


Figure 20. Our approach design

More specifically we will use BLE's Characteristics as channels, there will be a characteristic for Receiving only and another one for Sending only. This communication will be between the end-node and on the central-node side there will be a script sending and receiving back and forth packets from the BLE medium to a specific port using UDP. Similar goes for the NRF24 technology.

The general schema of our implementation is to have a central-node as forwarder of each wireless communication technology, gateway and broker. That central-node will be a Raspberry which already has two wireless technologies attached (Wi-Fi & BLE). Adding a nrf24l01 transceiver module we expanded the range to 3 technologies.

As our Raspberry Pi can communicate to these 3 protocols the next challenge would be to transfer packets of MQTT-SN from end-node to the Raspberry and then send them via UDP to the MQTT-SN port of RSMB 1884.

Analysis of devices connected to the central-node.

- The Wi-Fi device will connect to the RSMB as a regular MQTT Client making use of the 1883 port to connect directly to the broker.
- BLE will connect to the bridging script we created in Python and forward the messages back and forth to the gateway and then to the broker via UDP.
- Our nrf24Mesh [20] nodes will be connecting to the bridging script which is the Mesh master node as well and forward the messages.

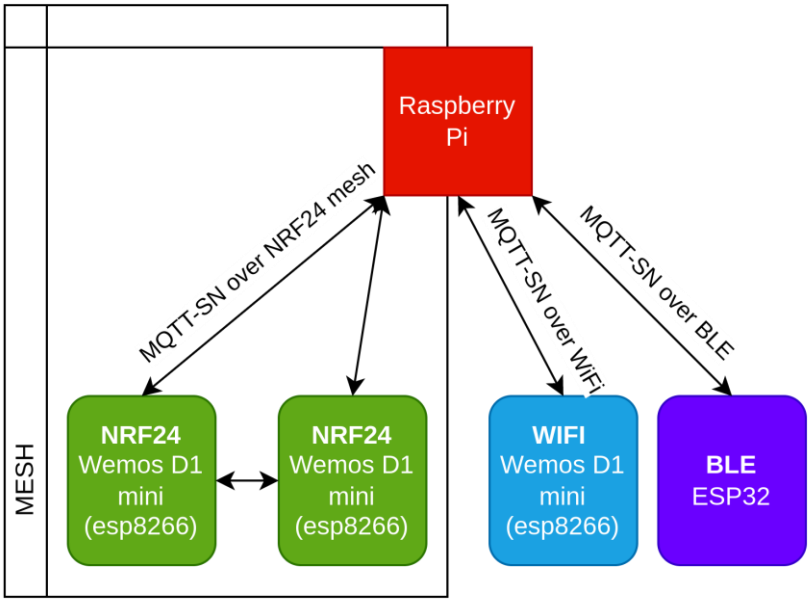


Figure 21. Our approach design - devices schema

For the purpose of this architecture, we used the two WSN (BLE, RF24) as async receive - transmit. The communication between end-node and forwarder implements the async receive - transmit design in each technology. Then the messages received through the readings will be parsed from the library created by John Donovan [21].

The library was made to create packets of MQTT-SN according to the specifications as also to parse packets and get the data needed for the communication.

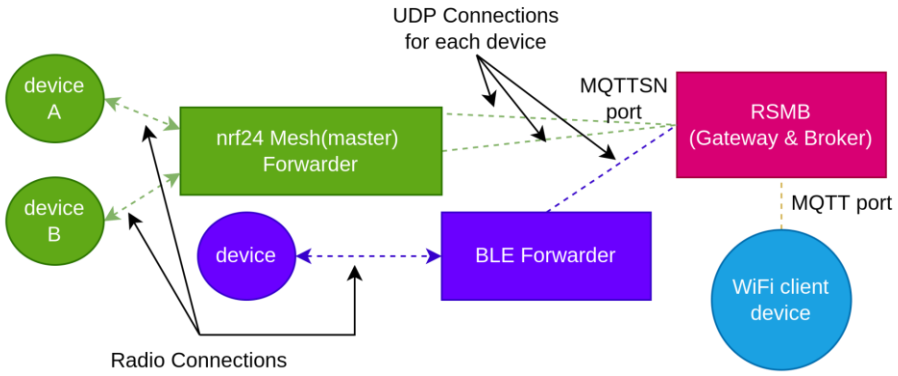


Figure 22. Our approach design - communication schema

2. Components

2.1. Development boards

The boards used for our implementation are described below with all their specifications:

2.1.1. Raspberry Pi 3B

Figure 23. Raspberry Pi

For more you can visit : https://www.raspberrypi.org/

2.1.2. ESP32

Our board model was: "*WEMOS-ESP32 ESP-WROOM-32 Wi-Fi + Bluetooth BLE - IOT Microcontroller with 18650 battery holder*"

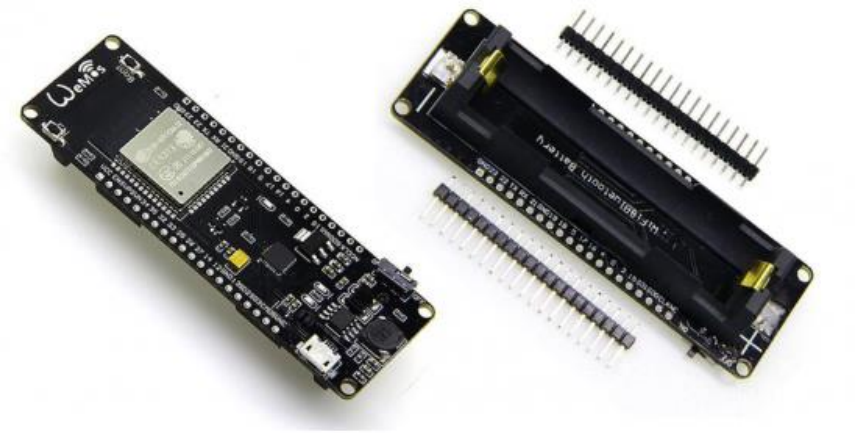


Figure 24. Wemos Esp32

Antenna and RF balun, power amplifier, low-noise amplifiers, filters, and a power management module are all built into the ESP32. The entire solution occupies the smallest amount of space on the printed circuit board. This board uses TSMC 40nm low power technology with 2.4 GHz dual-mode Wi-Fi and Bluetooth chips, which has the best power and RF attributes, is safe, dependable, and expandable to a variety of applications.

When working on ESP32 projects, you'll need to include a power source. Such issues could be solved with this little board. The ESP32 might run for 17 hours or more on an 3000mAH 18650 battery.

Features:

- 18650 charging system integrated.
- Indicate LED inside (Green means full & Red means charging)
- Charging and working could be at the same time.
- 1 Switch could control the power.
- 1 extra LED could be programmed (Connected with GPIO16[D0])
- 0.5A charging current
- 1A output
- Overcharge protection

- Over discharge protection
- Full ESP32 pins break out

Highlights:

- High performance-price ratio
- Small volume, easily embedded to other products
- Strong function with support LWIP protocol, FreeRTOS
- Supporting three modes: AP, STA, and AP+STA
- Supporting Lua program, easily to develop

This device was used as a client device for testing MQTT-SN over BLE.

2.1.3. ESP8266

Our board model was : *“Wemos D1 Mini”*

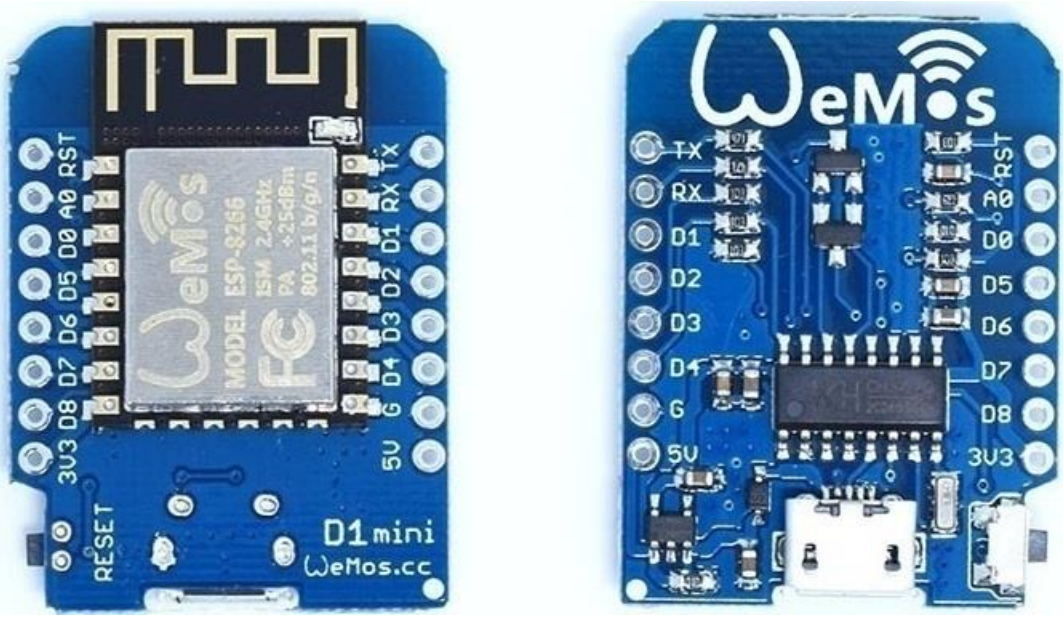


Figure 25. ESP8266 Wemos D1 Mini

The WeMos D1 small is a microcontroller development board with wireless 802.11 (Wi-Fi) capabilities. It converts the widely used ESP8266 wireless microcontroller into a complete development board. Programming the D1 small is as straightforward as programming any other Arduino-based microcontroller because the module has a built-in microUSB interface that allows it to be programmed directly from the Arduino IDE (ESP8266 functionality must be enabled via board manager).

Specifications

- Microcontroller: ESP-8266EX
- Operating Voltage: 3.3V
- Digital I/O Pins: 11
- Analog Input Pins: 1 (Max input: 3.2V)
- Clock Speed: 80MHz
- Flash: 4M bytes

Features

- 11 digital input/output pins, all pins have interrupt/pwm/I2C/one-wire supported (except D0)
- 1 analog input (3.2V max input)
- a Micro USB connection
- Compatible with Arduino
- Compatible with NodeMCU
- Compatible with MicroPython
- 4MB Flash
- Lots of Shields compatible in market

2.2. Sensors & Actuators

For the needs of a demonstration of our implementation we used a small number of sensors and actuators so the result of the communication can be more visual. The sensors we used were:

- Temperature sensors
- Ultrasonic distance sensors
- Pir motion sensor
- And LEDs as actuators

3. Network Analysis

3.1. General Scheme

3.1.1. Raspberry Pi as Gateway

The gateway acts as a sink node for all sensor nodes in a typical IoT system and represents the edge node. The Raspberry Pi 3B (shown in Figure 23.) was used to create the gateway, which is a small powerful computer that runs Linux. The Raspberry Pi's ARM (Advanced RISC Machines) CPU is powered by a Broadcom system-on-chip processor. Bluetooth and Wi-Fi transverses are included into this little PC [26] [27]. The Raspberry Pi was released in February of 2012, however its origins can be traced back to the University of Cambridge's Computer Laboratory in 2006. More information about the Raspberry Pi's architecture can be found on [28].

As a gateway we can choose to use PAHO as gateway of MQTT-SN and a broker of our choice, or RSMB that is a gateway and MQTT Broker at the same time. Both options were tested in order to examine deliverability or other malfunctions.

In our implementation we used both and concluded that the broker that suits our needs is RSMB for the reason that it is more compact.

3.1.2. Raspberry Pi as Forwarder

In order to forward the traffic between the end-node and the gateway we created a Python script that forwards the packets received from the transport layer (BLE, NRF24 etc.) to the UDP port of the RSMB and vice versa.

3.1.3. NRF Mesh

The NRF Mesh network is designed to be a highly efficient and flexible communication system, using a combination of mesh nodes and a master node to transmit data and messages. The mesh nodes, in this case 2 ESP8266 units, work together to create a network of nodes that can transmit data over a wide area. The mesh nodes are equipped with the necessary hardware and software to perform their role in the network, and they communicate with each other in real-time to ensure that messages are transmitted quickly and accurately.

The master node, located on a Raspberry Pi, acts as the central hub of the NRF Mesh network, coordinating the flow of information between the mesh nodes and the external world. The master node is responsible for executing the NRF Forwarder script, which acts as the bridge between the mesh network and the external communication protocols. The script ensures that messages from the mesh nodes are transmitted to their final destination using the best available path.

The ability to transmit data over the best available path is a key feature of the NRF Mesh network, as it ensures that data can be transmitted quickly and efficiently even in situations where some parts of the network may be unavailable. This allows for reliable communication even in challenging environments, such as in areas with limited network coverage or during periods of network congestion.

In conclusion, the NRF Mesh network is a highly efficient and flexible communication system, designed to transmit data and messages over a wide area using a combination of mesh nodes and a master node. Its ability to transmit data over the best available path and its adaptability to changing network conditions make it an ideal solution for a wide range of applications and environments.

3.1.4. Bridges

The bridge-mechanism plays a crucial role in ensuring the smooth and efficient functioning of the forwarders. It acts as the intermediary between the UDP port and the wireless protocol, allowing for seamless communication between the two. The script that makes up the bridge-mechanism is highly optimized and designed to handle high volumes of data transfer in real-time, ensuring that messages are transmitted quickly and accurately. This is particularly important in applications where time is of the essence, such as in real-time monitoring systems and automated control systems.

The bridge-mechanism is also designed to be flexible and adaptable to changing technology and communication protocols. This allows for seamless integration with new and emerging technologies, ensuring that the forwarders are always up to date and capable of handling the latest developments in the field. Additionally, the bridge-mechanism is designed to be secure, protecting sensitive data from being intercepted or compromised during transmission.

In conclusion, the bridge-mechanism is a crucial component of the forwarders, and its efficient and reliable performance is essential to the success of the overall system. The combination of high-speed data transfer, adaptability, and security makes the bridge-mechanism a key element in the design of cutting-edge communication systems and applications.

Chapter Four- Implementation

1. Introduction

In our implementation, we have set up a system that incorporates heterogeneous wireless sensor networks. These networks consist of sensor nodes that utilize different wireless communication technologies, including nrf24, Bluetooth Low Energy (BLE), and Wi-Fi.

To facilitate the integration of data from these diverse sensor nodes into a unified network, we have implemented a gateway node. This gateway node acts as an intermediary between the sensor nodes and the MQTT broker.

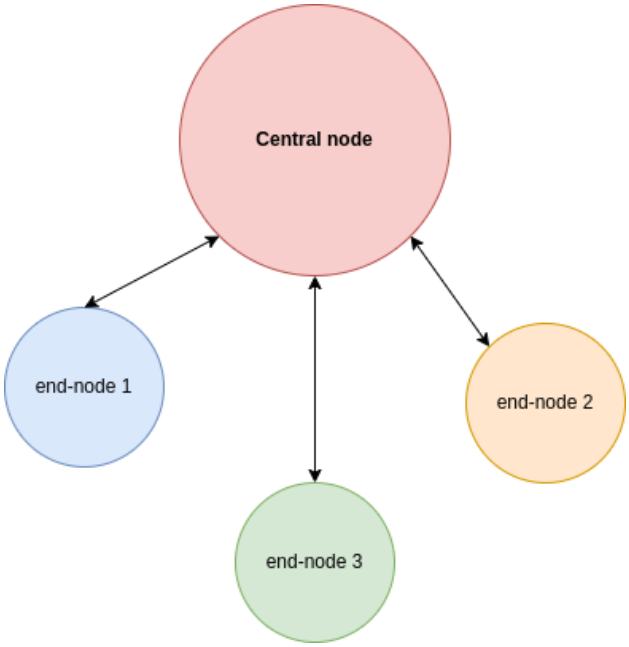


Figure 26. Overview of our schema

To enable the transmission of MQTT-SN (MQTT for Sensor Networks) packets between the sensor nodes and the gateway node, we leverage the RX (Receive) and TX (Transmit) channels of each wireless technology utilized in the sensor nodes. These channels serve as the underlying communication infrastructure for exchanging MQTT-SN packets.

When a sensor node generates data, it encapsulates that data into MQTT-SN packets and transmits them over the appropriate wireless communication technology's RX channel. These packets contain information regarding the data source, the actual sensor data, and other necessary metadata. Upon receiving the MQTT-SN packets on their respective RX channels, the gateway node forwards these packets to a Bridging Script. This script is responsible for extracting the MQTT-SN packets, converting them into MQTT protocol format, and sending them to the MQTT broker. The MQTT broker acts as a central hub where the data from different types of

sensor nodes converges.

Conversely, when the MQTT broker sends data to the sensor nodes, the gateway node receives the MQTT packets and passes them to the Bridging Script. The Bridging Script then converts the MQTT packets into MQTT-SN format and sends them back to the appropriate sensor node using the corresponding TX channel of the wireless technology employed by that node. By utilizing this approach, we are able to integrate and exchange data seamlessly between sensor nodes using different wireless technologies within the same MQTT network. This allows for efficient data collection, processing, and analysis across the heterogeneous wireless sensor networks.

More analytically we can see on Figure 27. how the flow of the communication takes place.

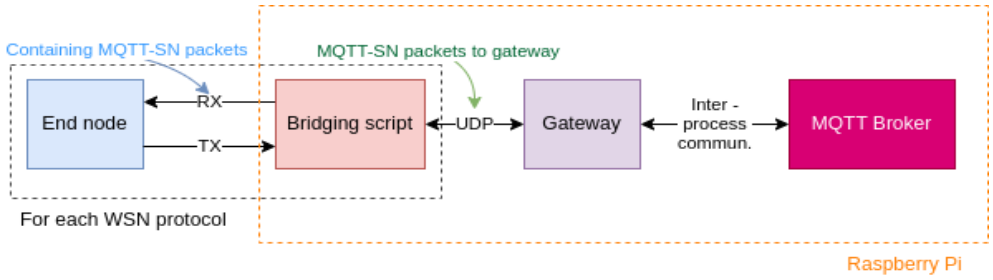


Figure 27. Flow of communication

It is important to notice that there is a need of following the “flow” of the library for example, you need to wait for acknowledgements, first send the connect message before everything else, and send acknowledgements if the QoS suggests so.

MainLoopCode (see Figure 28.) refers to the flow described above as well as whatever extra we want to apply in our implementation such as sensors and actuators

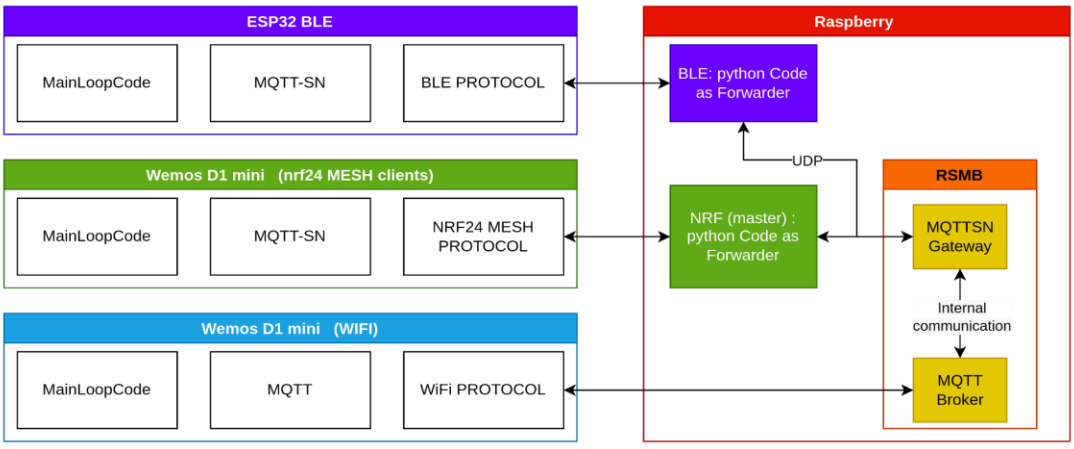


Figure 28. Schema description of how the code will work for all technologies

2. Libraries

2.1. The libraries we used on End-node:

- **BLE End-node:**

ESP32_BLE_Arduino is a software library for ESP32 microcontroller board that simplifies the development of Bluetooth Low Energy (BLE) applications. It provides APIs to interact with BLE devices, including standard and custom profiles and services. The library makes it easy to advertise, scan, connect, and interact with BLE devices and provides event-driven callbacks for BLE events. Overall, ESP32_BLE_Arduino is a flexible and powerful library for developing BLE applications on the ESP32 platform. More specifically the libraries that were used are:

- BLEDevice[1]
- BLEServer[2]
- BLEUtils[3]

- **NRF End-node:**

The nRF24 library is an open-source software library for the nRF24L01+ wireless transceiver chip. It simplifies the development of wireless communication applications on microcontroller platforms like Arduino. The library provides an easy-to-use interface for controlling the nRF24L01+ chip, including configurable data rates, packet handling, and transmission power. It supports point-to-point and multi-node communication, interrupt-driven data transmission and reception, and optional hardware SPI. The library also includes examples demonstrating how to build wireless communication systems, such as a chat app and sensor network. Overall, the nRF24 library is a popular and versatile library for developing wireless communication applications on microcontroller platforms. More specifically the libraries that were used are:

- RF24[4]
- RF24Network[5]
- RF24Mesh[6]

- **MQTT-SN-messages**

The **mqtt-sn-arduino** library follows the MQTT-SN protocol specification to create and

[1] https://github.com/nkolban/ESP32_BLE_Arduino
[2] https://github.com/nkolban/ESP32_BLE_Arduino
[3] https://github.com/nkolban/ESP32_BLE_Arduino
[4] https://github.com/nRF24
[5] https://github.com/nRF24
[6] https://github.com/nRF24

send packets over the network. The MQTT-SN protocol uses a message-oriented approach where each message is encapsulated in a packet.

The library provides functions for creating and sending packets for all standard MQTT-SN message types, such as connect, subscribe, publish, and disconnect. Each packet is constructed based on the specific message type and parameters provided by the developer.

For example, to send a publish message, the developer would use the MQTTSNPacket_publish function provided by the library. This function takes in parameters such as the topic name, payload, and QoS level, and constructs a packet according to the MQTT-SN protocol specification. The packet is then sent over the network using the underlying transport protocol, such as UDP or Bluetooth Low Energy (BLE). Similarly, for other message types such as subscribe and connect, the library provides functions for creating and sending the corresponding packets.

Overall, the mqtt-sn-arduino library abstracts away the details of packet creation and network transport, allowing developers to focus on the content and behavior of their messages while ensuring compliance with the MQTT-SN protocol specification.

➔ Library for creating mqttsn-packets made by John Donovan [7]

2.2. The libraries we used on the central-node

- **BLE communication** (Python)

Bluepy is a Python library that simplifies communication with Bluetooth Low Energy (BLE) devices on Linux-based platforms. With Bluepy, developers can use Python scripts to discover, connect to, and interact with BLE devices, including support for notifications that allow devices to send updates asynchronously.

➔ Bluepy [8]

[7] https://bitbucket.org/MerseyViking/
[8] https://github.com/IanHarvey/bluepy

- **NRF communication** (Python)

RF24Mesh is a Python library that simplifies the creation of wireless mesh networks using the NRF24L01+ radio module. It builds upon the RF24Network library and provides automatic network topology management, dynamic addressing, and routing algorithms for a self-organizing wireless mesh network of up to 250 nodes. The library allows nodes to join or leave the network dynamically without affecting the network's operation, making it easy to create reliable wireless networks for various applications. RF24Network is using the original RF24 library that makes use of the protocol.

- ➔ RF24Mesh[9]
- ➔ RF24Network[10]
- ➔ RF24[11]

While we send & receive our data back and forth between end-nodes and the Raspberry Pi using the RF24 libraries , we needed a library to handle those packets and forward them to the gateway. For that purpose, our approach was to use UDP for inter-process communication. Because we wanted to use one single UDP socket for Read and for Write (due to the gateway using one UDP port), we needed a way to handle the states of the socket (writing state & reading state) , for that purpose we chose the library “Select”.

- **Select**[12] (Python)

The select library is particularly useful when dealing with UDP packets, as UDP is a connectionless protocol and does not have a mechanism to ensure packet delivery or order. When receiving UDP packets, you may need to handle multiple incoming packets at once, and the select library can help you efficiently manage this process.

To use select with UDP packets, you can create a socket using the socket.socket() function and specify the socket type as socket.SOCK_DGRAM. Then, you can add the socket to a list of sockets to be monitored by select.select(). When a packet is received on the socket, select.select() will indicate that the socket is ready for I/O and you can read the data from the socket using the socket.recvfrom() function.

[9] https://github.com/nRF24
[10] https://github.com/nRF24
[11] https://github.com/nRF24
[12] https://docs.python.org/3/library/select.html

By using select in this way, you can efficiently handle multiple incoming UDP packets without blocking, allowing your application to continue processing other tasks while waiting for incoming data.

Overall, the select library is a valuable tool for managing UDP packet reception, especially when dealing with a large number of incoming packets or when you need to handle multiple sockets simultaneously.

3. Examples of MQTT-SN packets on end-nodes using the MQTT-SN-Arduino library

→ **Connection to the gateway:**

In order to check if the MQTT-SN (MQTT for Sensor Networks) client is currently connected to a broker or not. If the client is not currently connected, the code attempts to connect to the broker using the mqttsn.connect() method. The parameters passed to the mqttsn.connect() method are:

- flags: a set of flags used to specify options for the connection
- 10: the duration in seconds for which the client will keep trying to connect to the broker
- "esp32": a string that specifies the client name or identifier to be used in the connection.

If the connection is successful, the code will return from the function. If the connection is not successful, the client will keep trying to connect for the specified duration.

Code:

```
if (!mqttsn.connected()){
        // if not connected - connect
        mqttsn.connect(flags, 10, "esp32");  // Flags , Duration=10 , clientName
        return;
}
```

→ **Registering a topic :**

This part of the implementation of the MQTT-SN protocol, refers to the registration of a topic for publishing messages.

In MQTT-SN, topics are used to classify and organize messages that are sent between devices. When a device wants to publish a message, it specifies the topic it wants to publish to, and all devices that have subscribed to that topic will receive the message.

The code mqttsn.register_topic(TOPIC_PUB) is registering a topic with the MQTT-SN client library, using the variable TOPIC_PUB as the name of the topic. This means that the device running the code is now able to publish messages to that topic.

Depending on the implementation, there may be additional parameters that can be set when registering a topic, such as the QoS level of the messages that will be published to that topic.

Code:

mqttsn.register_topic (TOPIC_PUB); //topic of publishing messages

→ **Creating a publish message :**

For the creation of a publish message to be sent with its parameters:

- flags = **DQQRWCTT** - stands for DUP - QoS - Retain - Will - CleanSession - TopicIdType (only QoS and TopicIdType requires two bits)
  For example of the flags variable below is a representation of QoS-1:
  uint8_t flags = 0b00100000; // the 0b prefix indicates a binary constant
- topicPubID = The ID created upon topic registration for the desired publish topic for the message to be sent to.
- Message = the actual message
- strlen(message) = the size of the message

Code:

mqttsn.publish(flags , topicPubID, message, strlen(message));

4. Bridging scripts

The bridging scripts are the key-components for making our implementation possible. There are two scripts , the first script is for using the BLE medium to sending back and forth to the gateway in form of UDP packets and the other the script is for doing the same process but for the NRF technology .

These scripts essentially act as a middleware layer that allows devices that communicate over BLE & NRF. Below there is the code of those scripts:

Initializations of the BLE_bridge_script

```
#!/usr/bin/env python3

import struct
from bluepy.btle import *
import socket
import time
import select

def current_milli_time():
    return round(time.time() * 1000)

print("Setting up...")

#UDP Socket Settings
UDP_IP = "127.0.0.1"
UDP_PORT = 1884
serverAddressPort   = ("127.0.0.1", 1884) #more compact
connected=False;
bufferSize = 1024
ble_address="30:AE:A4:14:7E:0A"
oldvalue=""
readFromBLEChar="6e400003-b5a3-f393-e0a9-e50e24dcca9e"
writeToBLEChar="6e400002-b5a3-f393-e0a9-e50e24dcca9e"

print("Starting...")
per = Peripheral(ble_address, "public")
print("BLE connection established....")

# Create a UDP socket at client side
UDPClientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
UDPClientSocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

sockets=[]
sockets.append(UDPClientSocket)
```

The main loop of the BLE_bridge_script

```
try:  # TRYCATCH IS FOR SAFELY CLOSING THE SOCKET
      while True:
            try:
                  notify = per.getCharacteristics(uuid=readFromBLEChar)[0]
                  data=notify.read()

            except BTLEException:
                  print("--> BTLE Exceptions - Retry connection")
                  Try:
                        #if exception reconnect
                        per = Peripheral(ble_address, "public")
                  except BTLEException:
                        pass
                  continue

            readable, writable, exceptional = select.select([UDPClientSocket], [UDPClientSocket],
[])
            if UDPClientSocket in readable:
                  # Handle incoming connections

                  response , address = UDPClientSocket.recvfrom(bufferSize)
                  print("response::::: "+str(response))
                  c = per.getCharacteristics(uuid=writeToBLEChar)[0]
                  c.write(response)


            if UDPClientSocket in writable:
                  if data!=oldvalue:
                        UDPClientSocket.sendto(data, serverAddressPort)

            oldvalue=data
finally:
      UDPClientSocket.close() #if you don't close it....rsmb thinks you re still ON
      print("Exiting......")
```

Initializations of the NRF_bridge_script

```
#!/usr/bin/env python3

from RF24 import *
from RF24Network import *
from RF24Mesh import *
from struct import pack, unpack
from time import sleep, time
import numpy as np
import socket
import select
import sys, os
import struct

#vars
connectedBool=False;
bufferSize = 1024
oldvalue=""
sockets=[]
nodes=[]

#UDP Socket Settings
UDP_IP = "127.0.0.1"
UDP_PORT = 1884
serverAddressPort = (UDP_IP, UDP_PORT)    #more compact

#functions
def findUdpSocketOfNode(list, platform):
      for i in range(len(list)):
            if list[i][0] == platform:
                  return list[i][1]
      return False

def checkIncomingFromUdp(list,mesh):
      for i in range(len(list)):
            readable, writable, exceptional = select.select([list[i][1]], [list[i][1]], [])    #,1
            if list[i][1] in readable:
                  response , address = list[i][1].recvfrom(bufferSize)  # Handle incoming conn
                  print("UDPresponse ::::: "+str(response))

                  if not mesh.write(to_node,bytearray(response), ord('M')):
                        if not mesh.checkConnection():      # If a write fails, check connectivity
to the mesh network
                              print("[ERROR]====--->  Cannot connect/send-> Renewing Address")
                              if not mesh.renewAddress():
```

```
                                    if not mesh.begin():
                                          print("---* Cannot begin mesh")
                              else:
                                    print("Send fail, Test OK")
                        else:
                                    print("Send OK:")


def assignNodesToSocketsTable(from_node,nodes):
      try:
            #for from_node in nodes:
            if from_node not in nodes:
                  nodes.append(from_node)
                  UDPClientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)      #Create a
UDP socket at client side
                  UDPClientSocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
                  sockets.append([from_node,UDPClientSocket])
                  print("sockets list "+str(sockets))
                  #UDPClientSocket.close()

      except:
            print("error on addressing ......")

# RADIO SETUP
radio = RF24(25,0);
network = RF24Network(radio)
mesh = RF24Mesh(radio, network)
```

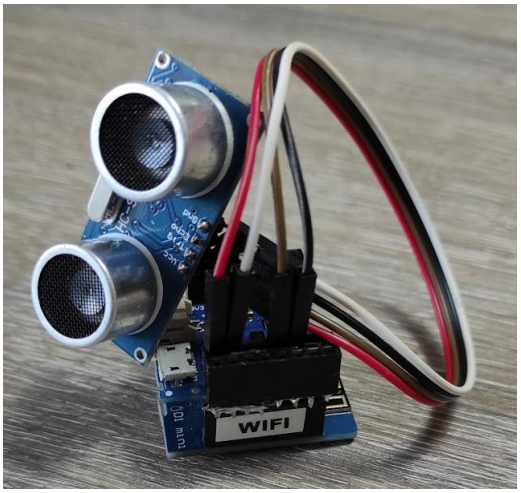
The main loop of the NRF_bridge_script

```
mesh.setNodeID(0)
print("Start nodeID (Master's): ", mesh.getNodeID())

if not mesh.begin():    # BEGIN MESH
      print("[Error]=---> Cannot begin Mesh")

radio.setPALevel(RF24_PA_MAX)       # POWER AMPLIFIER
radio.printDetails()

print("Entering main loop:.....")
try:
      while True:
            mesh.update()
            mesh.DHCP()

            checkIncomingFromUdp(sockets,mesh)

            while network.available():
                  header, payload = network.read(29)  #api frame len 29
                  from_node=header.from_node
                  assignNodesToSocketsTable(from_node,nodes)
                  if chr(header.type) == 'M':

                        to_node=from_node
                        readable, writable, exceptional =
select.select([findUdpSocketOfNode(sockets,from_node)], [findUdpSocketOfNode(sockets,from_node)],
[])
                        if findUdpSocketOfNode(sockets,from_node) in readable:
                              response , address =
findUdpSocketOfNode(sockets,from_node).recvfrom(bufferSize) # Handle incoming connections
                              print("UDPresponse ::::: "+str(response))

                              if not mesh.write(to_node,bytearray(response), ord('M')):
                                    if not mesh.checkConnection():
                                          print("[ERROR]====--->  Cannot connect/send-> Renewing
Address")
                                          if not mesh.renewAddress():
                                                if not mesh.begin():
                                                      print("---* Cannot begin mesh")
                                    else:
                                          print("Send fail, Test OK")
                              else:
                                          print("Send OK:")
                        if findUdpSocketOfNode(sockets,from_node) in writable:
```

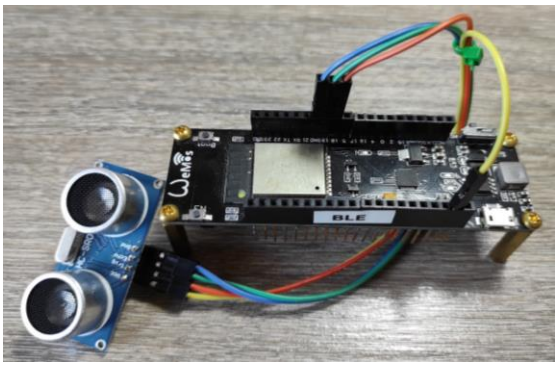
```
                              if payload!=oldvalue:
                                    findUdpSocketOfNode(sockets,from_node).sendto(payload,
serverAddressPort)
                        oldvalue=payload
                  else:
                        print("Rcv bad type {} from 0{:o}".format(header.type,header.from_node));

finally:
      for node in nodes:            #close all udp connection of all nodes
            findUdpSocketOfNode(sockets,node).close()
      print("Exiting......")
```

5. Photos

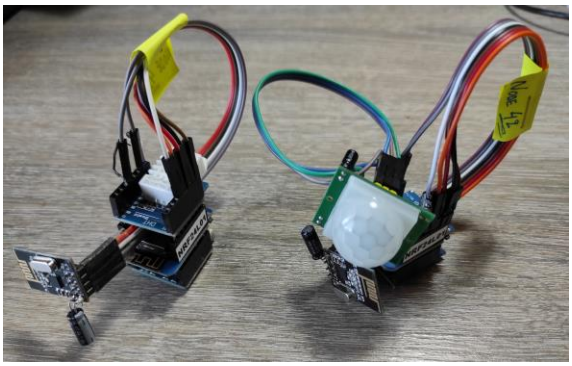
Raspberry Pi acting as Forwarder, Gateway and Broker.



ESP8266 as Wi-Fi end-node



ESP32 as BLE end-node



ESP8266s as nrf24 end-node (mesh nodes)

Chapter Five - Evaluation

1. Methodology

Our evaluation and experiments are focused on measuring the performance of MQTT-SN protocol implementation over three different wireless sensor networks - BLE, NRF24, and Wi-Fi - as well as their combinations. The goal of the experiments is to evaluate the efficiency and reliability of the MQTT-SN protocol over these different networks and to identify any potential issues or limitations.

In the first experiment, we are testing each wireless sensor network in isolation to evaluate the performance of MQTT-SN over each technology separately. We are measuring the packet loss and time consumption across 10 samples for each technology. This will give us a baseline for the performance of each wireless sensor network and allow us to compare them. For that to happen we are measuring the performance on a roundtrip basis. More analytically the end-node we are measuring is a publisher and a subscriber and we are sending messages to the gateway and we are comparing what messages we got back and how much time this procedure consumed.

In the second experiment, we are testing combinations of the wireless sensor networks to evaluate their performance when used together. Specifically, we are testing the communication between NRF24 and Wi-Fi, Wi-Fi and BLE, and BLE and NRF24. We are again measuring the packet loss and time consumption across 10 samples for each combination of wireless sensor networks.

By conducting these experiments and evaluating the performance of MQTT-SN over different wireless sensor networks and their combinations, we can gain insights into the strengths and weaknesses of the protocol and identify any potential areas for improvement.

Evaluating M2M communication on heterogeneous WSN infrastructure can be multifactored, therefore our evaluation narrowed to specific stats and parameters. Table 1. shows the evaluation schemes used to obtain our data at level of **QoS 1** (Quality of service 1: for enabling acknowledgements on our publish messages). In each scheme the firmware of each device used has the same code and is isolated from external factors (no sensors attached or other communications enabled).

Each node is a publisher and subscriber of the same topic and for the purpose of testing time consumption we needed one global timer therefore all nodes use the same gateway and broker. For each loop in the Arduino code on the end-nodes there is **500ms delay** (to avoid congestion) contained in our measurements.

Experiment 1 - Single-end WSN technology roundtrip messages		
1	BLE **roundtrip** message analysis	We are measuring the roundtrip message time in milliseconds on end-node (device) and total messages sent and received
2	Nrf24 **roundtrip** message analysis	
3	Wi-Fi **roundtrip** message analysis (using regular MQTT protocol)	

Table 1. Single-end WSN technology roundtrip messages evaluation method

A visual example of what we are trying to measure is shown below

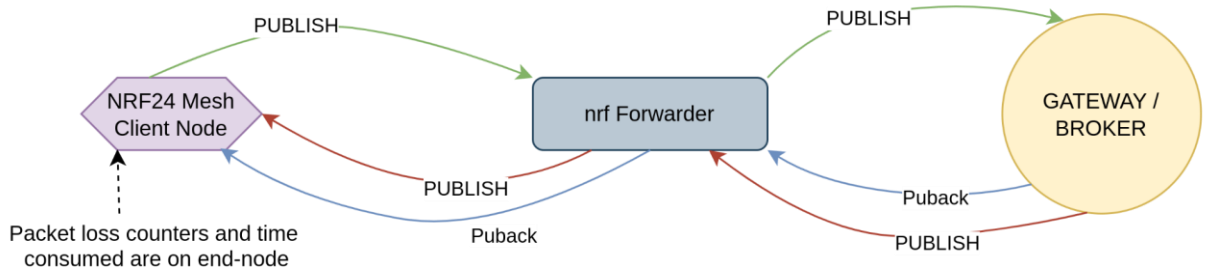


Figure 29. Visual representation of Experiment 1

In this experiment the packet loss data is recorded on the node-end as also the time duration.

More analytically on each node we keep counters of how many:

- Publish sent
- Publish received
- Publish acknowledgment sent
- Publish acknowledgment received

The time duration is measured from the time our end-node sent a publish message until it arrived back to the end-node (as the end-node is a subscriber).

Experiment 2 - End-to-end communication		
1	Wi-Fi ~ nrf24 (**end-to-end**)	Measuring time duration for a message to be sent and received on end-to-end and total messages sent and received. Time is measured by combining each nodes data with brokers timestamps of each message. Packet loss is calculated on each end-node. See Figure 30.
2	Wi-Fi ~ BLE (**end-to-end**)	
3	BLE ~ nrf24 (**end-to-end**)	

Table 2. End-to-end communication evaluation method

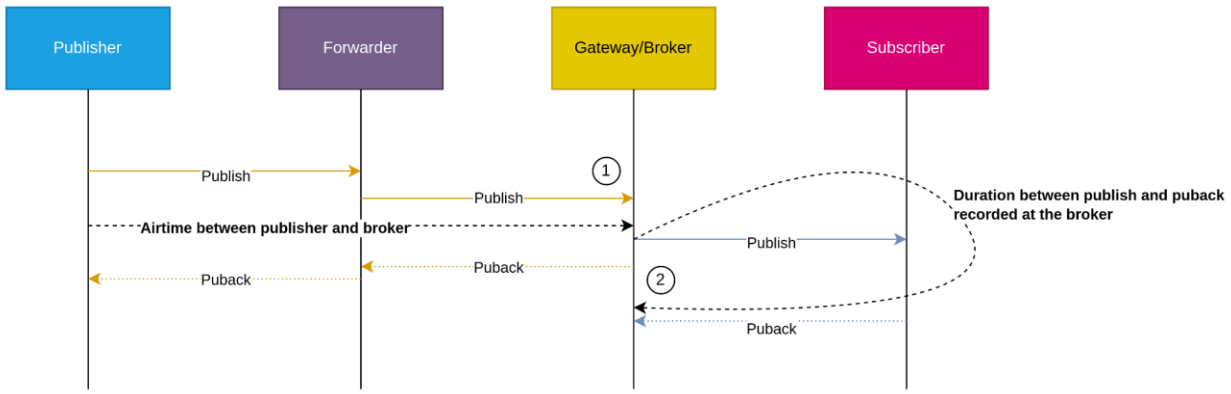


Figure 30. Visual representation of Experiment 2

(1) : Is the moment a publish message arrives to the broker from the “publisher end-node”.

(2) : Is the moment when the “subscriber end-node” responds with acknowledgement to the published message.

In this experiment the packet loss data is recorded on the node-end but the time duration is recorded on the logs of the broker since we cannot have a global timer that is in sync with all nodes.

2. Evaluation tests & results

2.1 Experiment 1 - Single technology roundtrip messages using regular MQTT-SN protocol

In this scenario we are executing an experiment on each technology isolated to measure each technology performance to the implementation of the MQTT-SN protocol. For each technology we will send publish messages to the gateway. Our end-node is a Publisher and Subscriber to the same topic so the messages will return back to the sender.

The measurements are taking place at the end-node as the end-node is recording to 4 counters some of the events (Publishes sent, Publishes received, Pubacks sent & Pubacks received). The time consumption of each roundtrip message is also recorded by the end-node and the time we are measuring refers to the duration between Publish sent and Publish received of the same message on the same node.

Objective:

Our objective is to measure the performance of the MQTT-SN protocol implementation for a single technology by evaluating the roundtrip messages.

Experimental Setup:

- The setup consists of an Arduino device acting as an end-node and a gateway.
- The Arduino device is both a publisher and subscriber to the same topic.
- The gateway facilitates communication between the Arduino device and the MQTT broker by converting MQTT-SN messages to MQTT protocol.

Performance Metrics:

We record the following performance metrics for each technology:

- Publishes Sent: The number of publish messages sent by the end-node.
- Publishes Received: The number of publish messages received by the end-node.
- Pubacks Sent: The number of puback messages sent by the end-node.
- Pubacks Received: The number of puback messages received by the end-node.
- Roundtrip Time: The time consumption between sending and receiving a publish message on the same end-node.

Execution:

- The experiment starts by deploying the Arduino device and ensuring its connection to the MQTT-SN gateway.

- The Arduino device initiates the publishing process to the specified topic.
- The end-node records the counters for each event (publishes sent, publishes received, pubacks sent, pubacks received) throughout the experiment.
- Additionally, the end-node measures the roundtrip time for each message by recording the timestamps when the message is sent and received.

Iterations:

- We perform multiple iterations of the experiment to gather sufficient data.
- The experiment can be repeated with different parameters or conditions to assess various scenarios or configurations.

Data Analysis:

We collect and analyze the recorded data for each technology.

- Performance metrics such as average roundtrip time, success rates (publishes sent vs. publishes received), and other relevant statistics are calculated.
- We compare the technologies based on the recorded metrics to evaluate their MQTT-SN protocol implementation performance.

By following this evaluation plan, we can assess the performance of the MQTT-SN protocol implementation for each technology by measuring and comparing the recorded counters and roundtrip times. The time consumption is calculated on the end-node by calculating the duration between on a publish sent and a publish received.

2.1.1 Metrics of BLE roundtrip evaluation

With BLE technology as a transport layer and we will send messages from the end-node (NRF) to the gateway. On our end node we have 4 counters counting some certain events:

- Publish sent by node
- Publish received by node
- Pubacks sent by node
- Pubacks received by node

The time consumption is calculated on the end-node by calculating the duration between on a publish sent and a publish received.

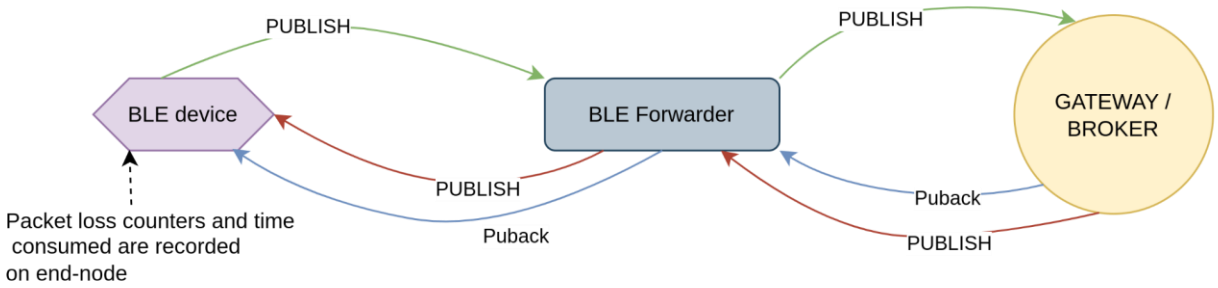


Figure 31. BLE Roundtrip time evaluation

Samples	Publish sent	Publish received	Puback sent	Puback received	Packet Loss (%)	Roundtrip avg time
1	334	167	167	167	50%	2175 ms
2	325	162	162	162	50%	2178 ms
3	336	168	168	168	50%	2200 ms
4	340	170	170	170	50%	2181 ms
5	320	160	160	160	50%	2186 ms
6	334	167	167	167	50%	2175 ms
7	325	162	162	162	50%	2178 ms
8	336	168	168	168	50%	2200 ms
9	334	167	167	167	50%	2185 ms
10	336	168	168	168	50%	2200 ms

Table 3. Results of BLE Roundtrip time evaluation

	Duration of measurement:	Publish sent	Publish received	Puback sent	Puback received	Packet Loss (%)	Roundtrip AVG time
Averages	10 min	334.3	167.1	167.1	167.1	50%	2473.3 ms

Table 4. Averages of BLE Roundtrip time evaluation

Our hypothesis for these abnormal results on this evaluation is that it has to do with congestion of the transport medium. Trying to test our hypothesis for further insights, we tested the same scenario with different delay: **650ms** and the results were as expected (slower transmission and the messages were relatively decreased). So, the hypothesis has been proved wrong and the problem does not rely on congestion issues.

Samples	Publish sent	Publish received	Puback sent	Puback received	Packet Loss (%)	Roundtrip avg time
1	334	167	167	167	50%	2175 ms
2	325	162	162	162	50%	2178 ms
3	336	168	168	168	50%	2200 ms
4	340	170	170	170	50%	2181 ms
5	335	168	168	168	50%	2170 ms

Table 5. Results of BLE Roundtrip time evaluation with different delay in-between messages (650ms)

It is observed that the percentage of big packet loss is strangely high. This indicates that we have to proceed to further investigation. An assumption after this evaluation is that 50% of the "puback" messages are not received and that's the reason the end-node sends again. The reason may vary to many reasons such as:

- The client did not get the message,
- Broker missed the acknowledgement sent by the client (more unlikely),
- The client did not send the acknowledgement,
- The forwarder may have missed the acknowledgement due to it being busy transmitting other messages either by client or by broker.

And more if we investigate it deeper and deeper. Next, we are planning to try to run the same evaluation with different delays in between of 750ms, 1000ms and 3000ms

2.1.2 Metrics of NRF24 roundtrip evaluation

Next, we will try with the NRF technology as a transport layer and we will send messages from the end-node (NRF) to the gateway. On our end node we have 4 counters counting some certain events:

- Publish sent by node
- Publish received by node
- Pubacks sent by node
- Pubacks received by node

The time consumption is calculated on the end-node by calculating the duration between on a publish sent and a publish received.

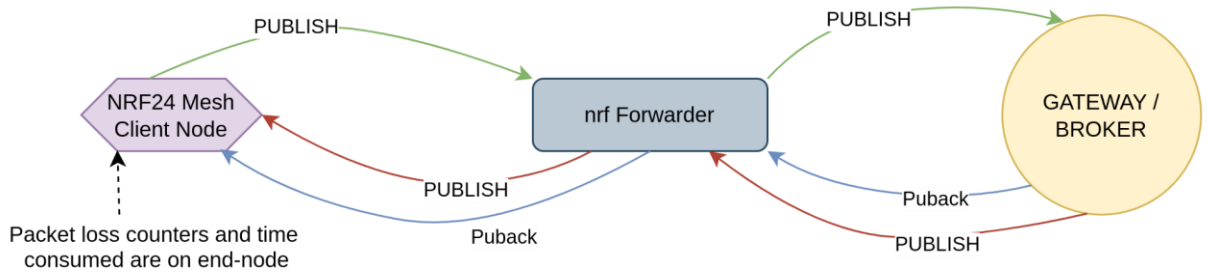


Figure 32. NRF24 Roundtrip time evaluation

Samples	Publish sent	Publish received	Puback sent	Puback received	Packet Loss (%)	Roundtrip avg time
1	166	166	166	166	0 %	1461 ms
2	168	168	168	168	0 %	1492 ms
3	167	167	167	167	0 %	1478 ms
4	166	166	166	166	0 %	1450 ms
5	165	165	165	165	0 %	1458 ms
6	170	170	170	170	0 %	1495 ms
7	167	167	167	167	0 %	1477 ms
8	169	169	169	169	0 %	1494 ms
9	168	168	168	168	0 %	1490 ms
10	162	162	162	162	0 %	1460 ms

Table 6. Results of NRF24 Roundtrip time evaluation

	Duration of measurement:	Publish sent	Publish received	Puback sent	Puback received	Packet Loss (%)	Roundtrip AVG time
Averages	10 min	166.8	166.8	166.8	166.8	0%	1475.5 ms

Table 7. Averages of NRF24 Roundtrip time evaluation

After the evaluation of RF24 implementation, the results were much better than BLE's, it is proven that with this technology our message delivery is more robust. The speed of the communication is faster in this technology however it prompts us to investigate further the implementation of the bridge script for possible delays.

2.1.3 Metrics of Wi-Fi roundtrip evaluation

Next, we will try with the regular Wi-Fi technology as a transport layer and we will send messages as simple MQTT from the end-node (Wi-Fi) to the broker. On our end node we have 4 counters counting some certain events:

- Publish sent by node
- Publish received by node
- Pubacks sent by node
- Pubacks received by node

The time consumption is been calculated on the end-node by calculating the duration between on a publish sent and a publish received.

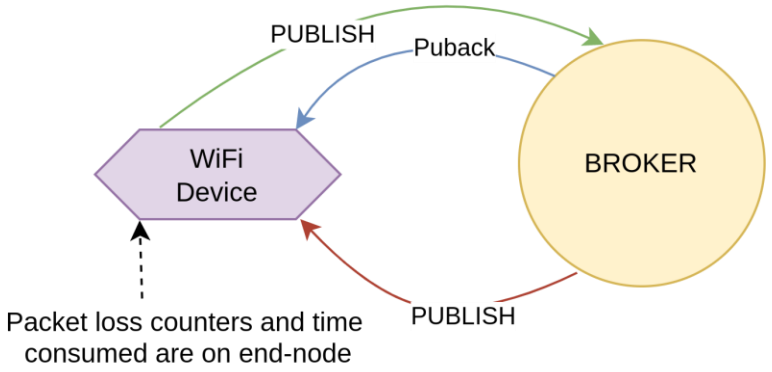


Figure 33. Wi-Fi Roundtrip time evaluation

Samples:	Publish sent	Publish received	Puback sent	Puback received	Packet Loss (%)	Roundtrip avg time
1	549	549	549	549	0 %	1001 ms
2	548	548	548	548	0 %	995 ms
3	548	548	548	548	0 %	997 ms
4	550	550	550	550	0 %	1002 ms
5	552	552	552	552	0 %	1042 ms
6	548	548	548	548	0 %	955 ms
7	550	550	550	550	0 %	1042 ms

8	548	548	548	548	0 %	998 ms
9	549	549	549	549	0 %	1003 ms
10	547	547	547	547	0 %	993 ms

Table 8. Results of Wi-Fi Roundtrip time evaluation

	Duration of measurement:	**Publish sent**	**Publish received**	**Puback sent**	**Puback received**	**Packet Loss (%)**	**Roundtrip AVG time**
Averages	10 min	548.9	548.9	548.9	548.9	0%	1002.8 ms

Table 9. Averages of Wi-Fi Roundtrip time evaluation

Wi-Fi's implementation was capable of providing faster data transfer speeds compared to previous WSNs evaluated. Apart from the faster communication, Wi-Fi has a high message deliverability rate, meaning that the data transmitted over Wi-Fi is less likely to be lost or corrupted during transmission, resulting in a more reliable communication experience.

Our assumption is that our bridging script creates a bottleneck in heterogeneous communication due to Wi-Fi does not make use of that script, that's why those results are better.

2.2 Experiment 2 - End-to-end communication

In our evaluation plan, we focus on measuring the end-to-end time consumption of the messaging system. This allows us to assess the efficiency and performance of message delivery from node-A (the publisher) to node-B (the subscriber) through the MQTT-SN protocol. While it's important to consider a holistic view of the system and various factors that may impact performance, our specific focus is on the end-to-end time consumption. This metric provides valuable insights into the overall efficiency of the messaging system.

To measure the end-to-end time consumption, we specifically measure the time taken between the moment a "publish" message is received by the broker from node-A and the moment the corresponding "puback" message is received by node-B. This time measurement is facilitated by the timer provided in the broker's log. By capturing this duration, we gain insights into the efficiency of message transmission and acknowledgement between the publisher and subscriber. Any delays or inefficiencies in the system can be identified by analyzing these time measurements.

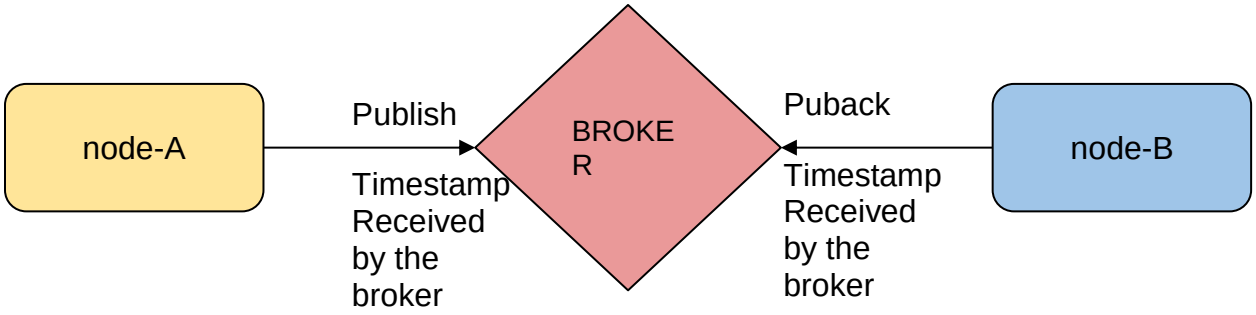


Figure 34. Evaluation method on end-end transmission duration

It's important to note that the end-to-end time consumption may be influenced by various factors beyond the messaging system itself. Factors such as network latency, node processing time, and external influences can impact the overall time taken for messages to be delivered. However, in this specific experiment, we are primarily focusing on the end-to-end time consumption as a metric to evaluate the performance of the MQTT-SN protocol implementation.

By analyzing the time taken for messages to be delivered from node-A to node-B and the corresponding acknowledgements, we can identify potential bottlenecks or areas of inefficiency within the system. Taking into account the limitations of this specific evaluation plan, we can gain valuable insights into the performance of the MQTT-SN protocol implementation by analyzing the end-to-end time consumption measurements and identifying any areas for improvement.

2.2.1 Metrics: Wi-Fi ~ nrf24 (end-to-end)

Evaluating packets sent and received between a NRF and a Wi-Fi device in the context of the evaluation of MQTT-SN implementation between those two WSNs.

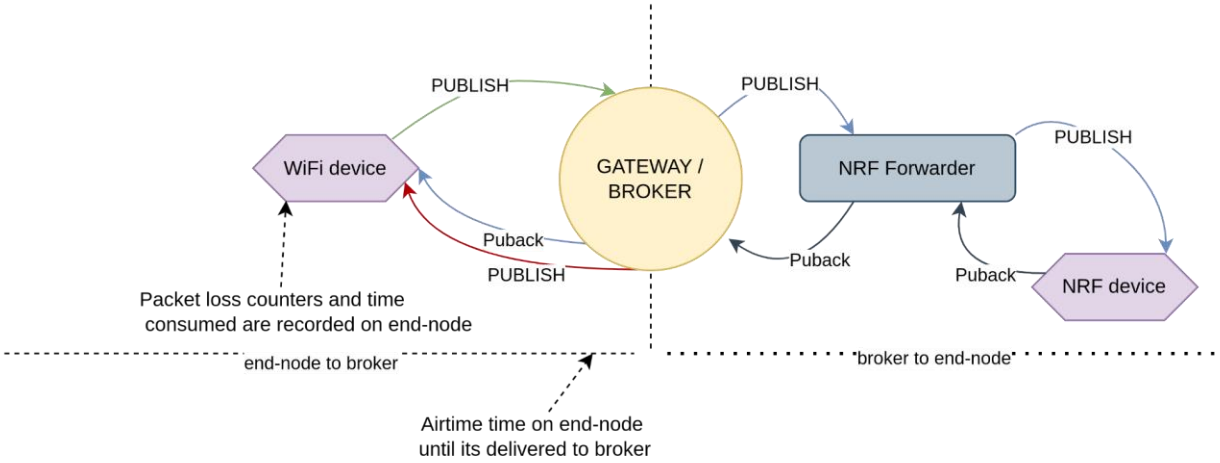


Figure 35. Representation of messaging process during evaluation Wi-Fi ~ nrf24 (end-to-end)

Samples	Publish sent by A node (NRF)	Publish received by A node (NRF)	Publish sent by B node (WI-FI)	Publish received by B node (WI-FI)	Average time consumed (based on evaluation method on Figure 35.)
1	196	776	600	796	1300 ms
2	185	737	570	757	1145 ms
3	207	814	630	816	1265 ms
4	186	802	570	851	1175 ms
5	208	792	620	838	1394 ms
6	187	812	580	861	1445 ms
7	209	764	620	822	1194 ms
8	186	784	570	829	1355 ms
9	208	796	610	843	1496 ms
10	187	810	560	869	1245 ms

Table 10. Results of Wi-Fi ~ NRF (end-to-end)

2.2.2 Metrics: Wi-Fi ~ BLE (end-to-end)

Evaluating packets sent and received between a BLE and a Wi-Fi device in the context of the evaluation of MQTT-SN implementation between those two WSNs.

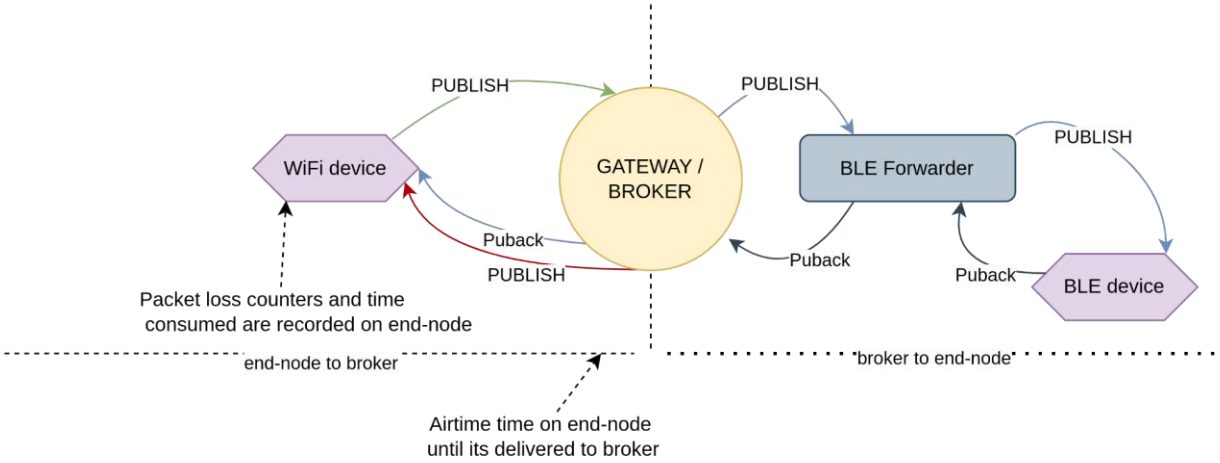


Figure 36. Representation of messaging process during evaluation Wi-Fi ~ BLE(end-to-end)

Samples	**Publish sent by A node**	**Publish received by A node**	**Publish sent by B node**	**Publish received by B node**	**Average time consumed** (based on evaluation method on Figure 36.)
10min	340	475	610	780	1524 ms
10min	327	461	594	758	1399 ms
10min	315	446	577	735	1729 ms
10min	338	480	622	791	1490 ms
10min	325	458	591	754	1670 ms
10min	318	451	584	743	1442 ms
10min	341	477	613	784	1600 ms
10min	326	459	592	755	1412 ms
10min	314	446	577	734	1665 ms
10min	340	481	621	791	1546 ms

Table 11. Results of Wi-Fi ~ BLE (end-to-end)

In all the tests run we observed that:

BLE was sending more messages than those delivered to the GW (this could be a problem related to the bridging script in a way of denial of service due to the script was servicing another request at the moment and ignored the next given request)

The PubAcks (Publish Acknowledgements) that were delivered to the end-node by the GW were half the amount of the actual publish tried to be sent by the end-node (BLE). That is the reason why as an amount of Publish messages the end-node received the half messages sent by itself. That means that those publish messages were never received by the GW.

It also seems that half messages sent by the Wi-Fi node arrived to the BLE device, which gives us a strong feeling that the bridging script ignores messages and our guess is that it has to do with either with the fact that the script is a single-thread application either that the library or the physical medium cannot manage that many transactions so it comes to congestion.

2.2.3 Metrics: BLE ~ nrf24 (end-to-end)

Evaluating packets sent and received between a NRF and a BLE device in the context of the evaluation of MQTT-SN implementation between those two WSNs.

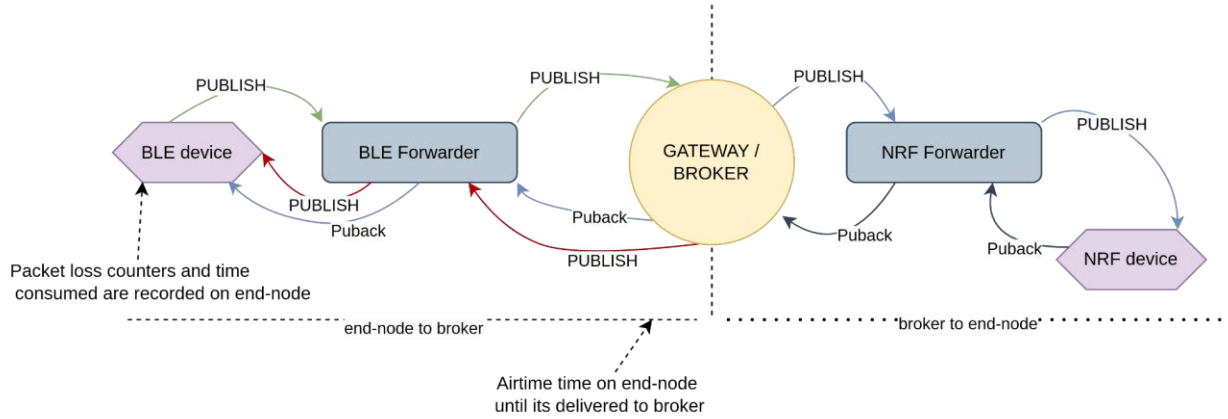


Figure 37. Representation of messaging process during evaluation BLE ~ nrf24 (end-to-end)

Samples	**Publish sent by A node**	**Publish received by A node**	**Publish sent by B node**	**Publish received by B node**	**Average time consumed** (based on evaluation method on Figure 37.)
1	520	720	200	170	1439.2 ms
2	509	705	194	165	1236.4 ms
3	529	735	206	175	1312.8 ms
4	498	692	193	162	1397.6 ms
5	541	748	210	178	1463.5 ms
6	516	713	197	164	1506.9 ms
7	531	738	208	176	1557.3 ms
8	505	700	191	161	1594.6 ms
9	543	752	212	180	1606.2 ms
10	518	716	199	165	1615.1 ms

Table 12. Results of BLE ~ nrf24 (end-to-end)

Within the scope of the RTT measurements we observe that the Wi-Fi device is 3 times faster and sends 2 times more messages than the BLE node and more specifically we can see that also at the end-end measurement tests. Also, here we have a strong feeling that the bridging script has something to do with these results. Talking of that, it is important to note that bridging scripts runs python libraries that use C wrappers which makes the complexibility of the communication even greater.

The final outcome of our evaluation is that our bridging script is constrained to our capabilities (software and hardware) there comes a need of an universal bridge (WSN to UDP) , so we can proceed to a better evaluation.

3. Overall charts

End-end average time consumption

In the chart below, we're going to take a closer look at the three technologies: Wi-Fi, BLE and NRF comparing their end-end behavior. More specifically we are viewing the behavior of the experiments that we examined interaction between Wi-Fi ~ NRF24 , Wi-Fi ~ BLE , BLE ~ NRF24 .

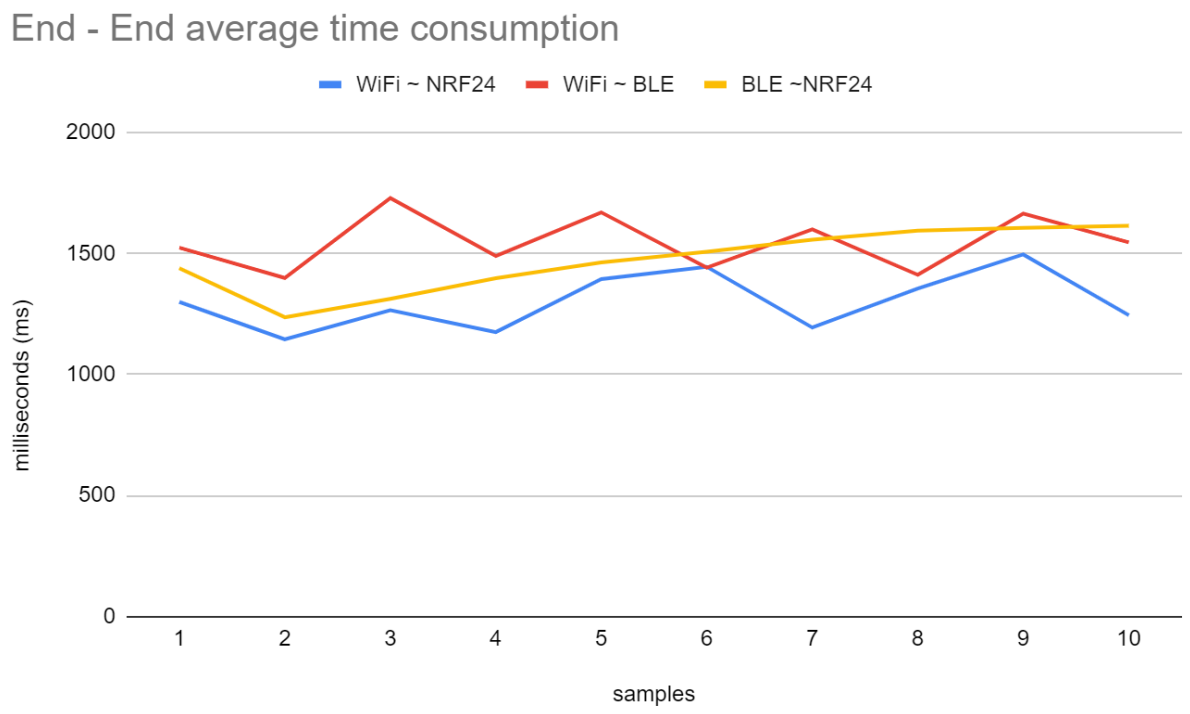


Figure 38. All end-end metrics on a single plot

From our evaluations the observation that prevails is that in our implementations the best performance in end-end perspective is the interaction between Wi-Fi and NRF.

Diving deeper to this interaction we investigated the data of our samples and it comes out that in our implementations the stability of the results were slightly uneven. By evaluating the R^2 in each end-end interaction we observe that the best interaction in perspective of better R^2 (more linear results) is Wi-Fi ~ BLE and not Wi-Fi ~ NRF as observed before in the perspective of which interaction is faster. In the charts below we can see the trend lines regarding our samples in each interaction.

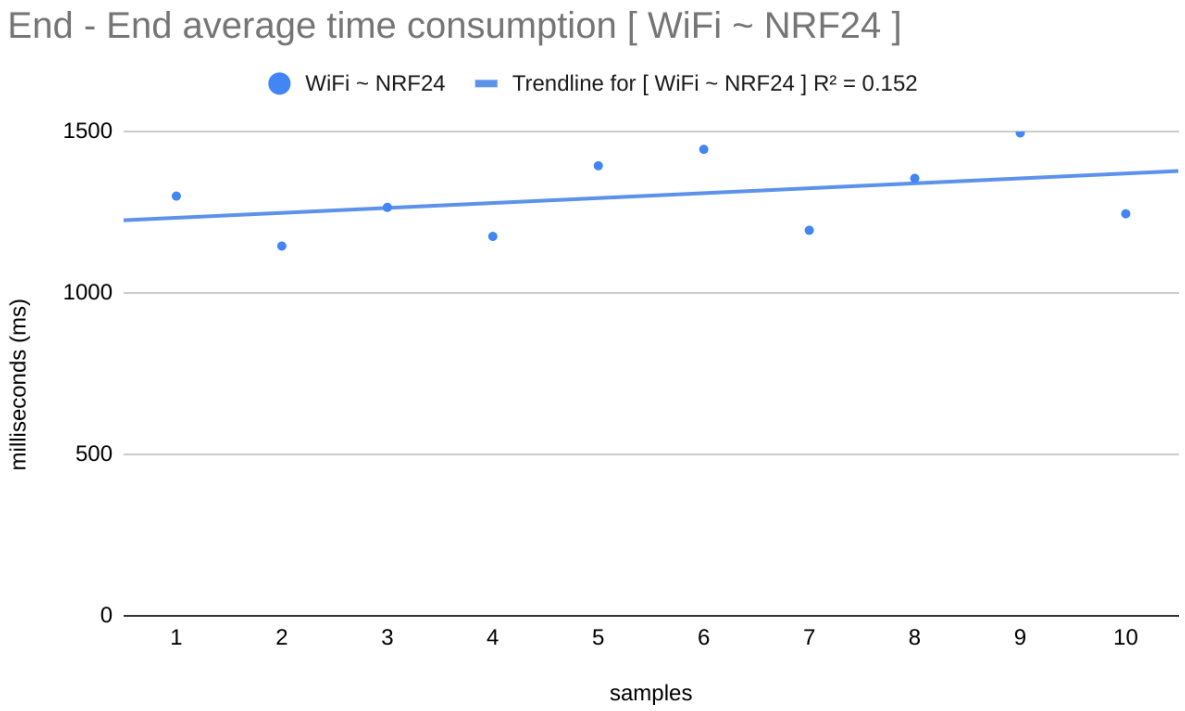


Figure 39. Wi-Fi ~ NRF24 trend line of measurements

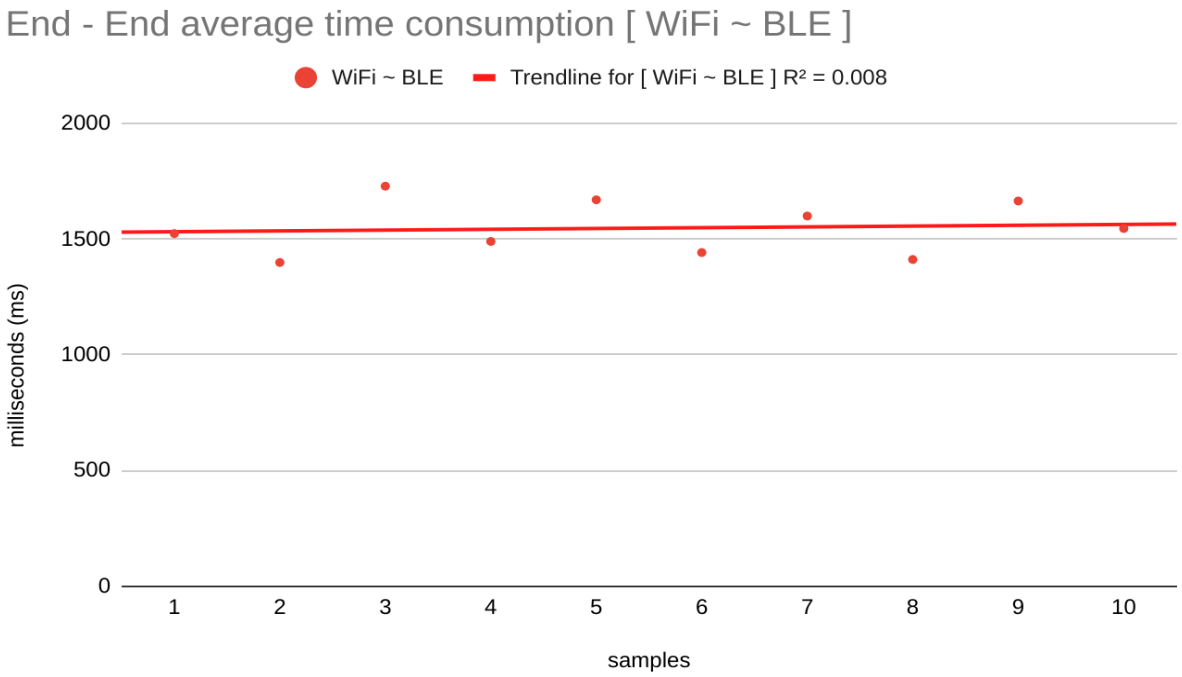


Figure 40. Wi-Fi ~ BLE trend line of measurements

Below we can examine all the end-end interactions in the same plot.

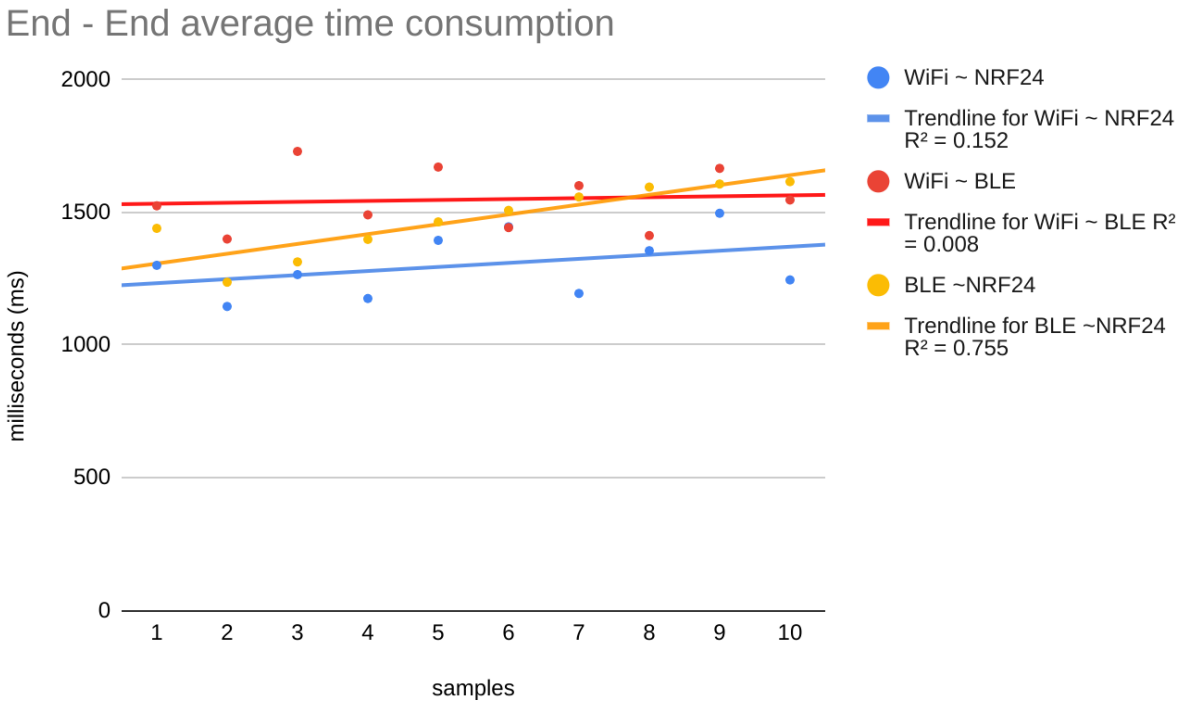


Figure 41. All end-end metrics trendlines on a single plot

Taking a look at the above overall chart we can make conclusions about the end-end experiment. We can see that the best implementation in matter of robust results is Wi-Fi~ BLE and the fastest is Wi-Fi ~ NRF.

End-End :	**Wi-Fi ~ NRF24**		**Wi-Fi ~ BLE**		**BLE~NRF24**
R^2 :	0.152	0.008	(best)	0.755	(worst)

Roundtrip time metrics charts

In this section there are charts presenting the roundtrip time measurements in each wireless technology. We can see that the fastest is Wi-Fi and the slowest is BLE.

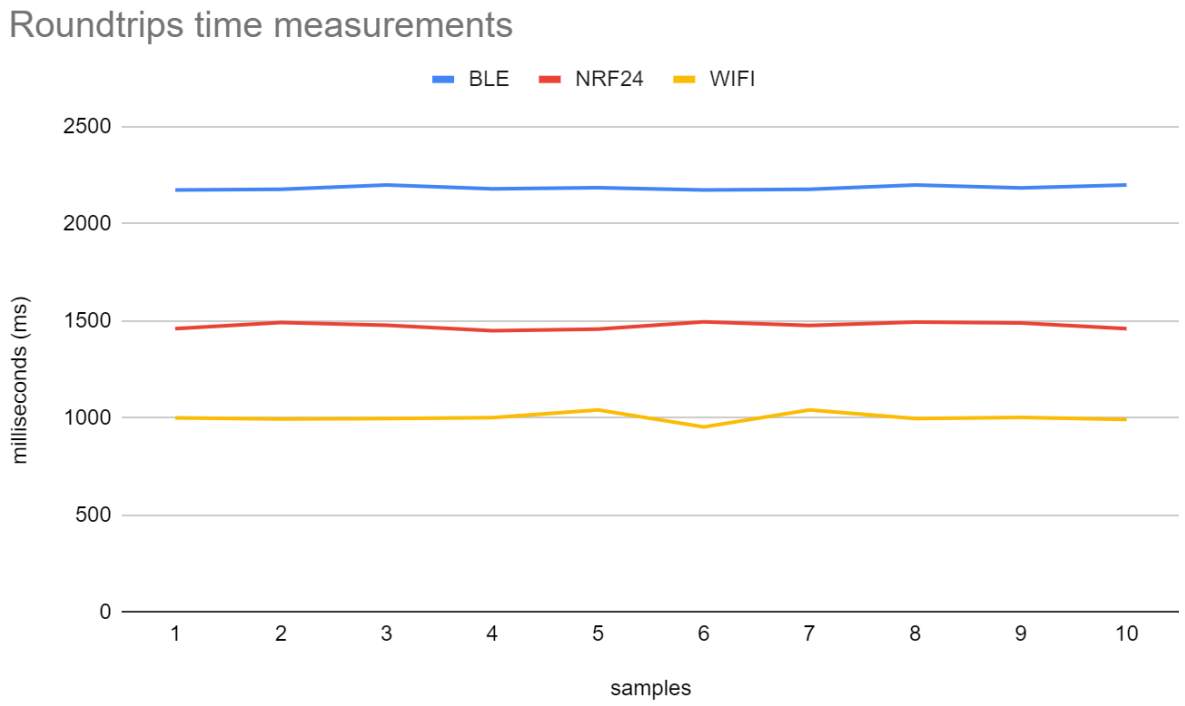


Figure 42. Roundtrip time metrics chart

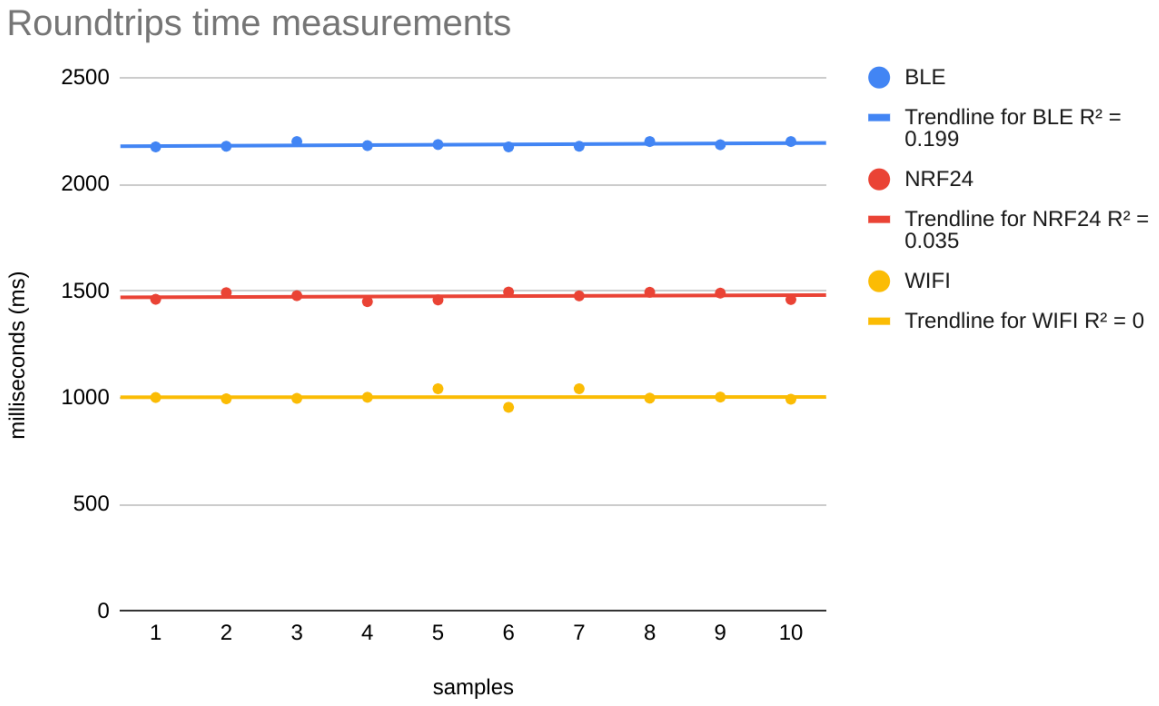


Figure 43. Roundtrip time metrics trendlines

Taking a look at the above overall chart we can make conclusions about the single wireless technology roundtrip time experiment. We can see that the best implementation in matter of robust results and the fastest is Wi-Fi .

End-End:		**Wi-Fi**	**BLE**	**NRF24**
R^2	:	0 (best)	0.199 (worst)	0.035

Chapter Six - Conclusion & future work

1. Conclusion

In our recent experiments, our focus was set on optimizing the communication efficiency within our Arduino BLE implementation that facilitated MQTT-SN messaging. To this end, we started investigating by adding varying delays between loops to gauge their influence on message transmission. As we progressively increased the delay intervals from 750ms to 1000ms and eventually to 3000ms, we stumbled upon an intriguing revelation: rather than experiencing the expected degradation in performance, we observed a profound enhancement in message deliverability. We investigated the behavior of our communication system's roots in response to this unanticipated development, taking special attention to the bridging script that converts BLE data into UDP packets on our Raspberry Pi.

Delay (ms)	***RTT (ms)***	***Packet Loss***
500 ms	2175 ms	50 %
650 ms	2230 ms	50 %
750 ms	2502 ms	50 %
1000 ms	3964 ms	50 %
3000 ms	5880 ms	0%

Table 13. Results of Roundtrip time and packet loss with different in-between transmission delays in the BLE implementation

Driven by curiosity and a need for clarity, we initiated an in-depth investigation into the bridging script's behavior and its potential interaction with asynchronous packet management. This attempt led us down a path of a demanding analysis, where we evaluated the script's ability to handle packets in an asynchronous manner. It was during this investigative phase that we made use of the Asyncio library in Python to examine any upcoming optimized results. The results of this exploration were nothing short of remarkable: the Asyncio library not only eliminated any unforeseen delay in message transmission but also gave us a 100% message deliverability rate.

The results were profound, causing our understanding to change its perspective of the interaction between communication delays, packet management, and deliverability. As a result, our experiment not only shed light on a previously undiscovered aspect of our communication system but also empowered us to re-evaluate and enhance our bridging mechanisms. Armed with these insights, we are now assured to further optimize our implementation, leveraging the newfound advantages of asynchronous packet handling to our Arduino BLE setup towards optimized levels of efficiency and reliability.

In contrast to the previous implementation, in this new approach that we used Asyncio it was seamlessly easy, we just had to make a function containing the asynchronous event that will occur, then you have to attach this function in the Asyncio's event_loop that will run indefinitely so we can get the async event. Here is a small example of code for the async management of the UDP packets.

```
async def listenToUdp(message):
    while True:
        # Wait for data from the UDP server
        data, addr = udp_socket.recvfrom(1024)
        print("Incoming from UDP:")
        print(data)
        # When data is received from the UDP server,
        # write it to the BLE device's write characteristic
        device.char_write(WRITE_CHARACTERISTIC_UUID, data)
```

```
loop=asyncio.get_event_loop()  # get the loop in a variable
asyncio.ensure_future(listenToUdp()) # attach our async function in the loop
loop.run_forever()    # run the loop forever
```

After using this library for our implementation, we tested the deliverability varying the delay between the message transmission from the end-node. In contrast to our first implementation (using "Select" library), by using Asyncio the messages have been delivered to the end-node async without any correlation with the stage of the end-node (transmitting or being in pause due to delay function). The results were stable despite the delay of the transmissions.

As we evaluated the Roundtrip time between the two implementations in the same experiment. We concluded that the implementation of the Asyncio library is two times faster than our previous implementation (Select).

Samples	***Implementation with Select (ms)***	***Implementation with Asyncio (ms)***	***Differentiation (%)***
1	2175	1110	51.03
2	2178	1100	50.51
3	2200	1105	50.23
4	2181	1092	50.07
5	2170	1098	50.6

Table 14. Comparison of the two implementations with both having delay between messages at 500ms in the BLE implementation

As described in Table 14. the implementation of Asyncio is two times faster. The below chart describes the comparison between the two libraries as well as it describes the stability of the transmissions despite the delay given between transmissions in each implementation.

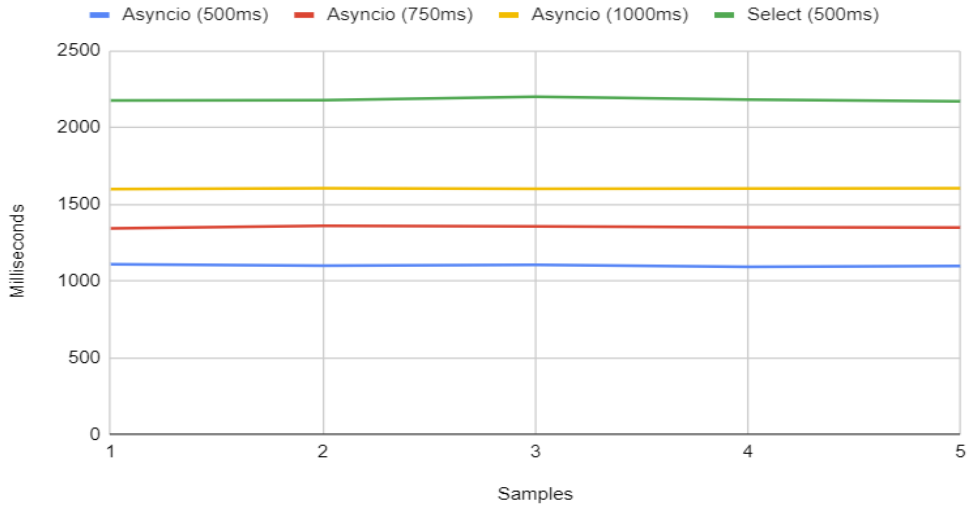


Figure 44. Chart showing the average time duration of a roundtrip message back to end-node in the BLE implementation

From our evaluations we concluded that the factors of failures or delays can be many, and the data of our evaluation conducted shows us that there is a need for more investigation on each part of the implementation.

The struggle that we faced during the evaluations was that without a shared or central time clock we cannot measure time. We evaluated the process of the messages by the scope of the broker. Calculating time consumption, we need first to have the transmission speed of our transport medium. That is something relative due to various factors such as interference, congestion, and processing delays (hardware or physical distance).

The most notable example that this area needs more research is the fact there are no bridging scripts available for connecting the packets of a WSN to the gateway via UDP.

There are many unstable factors that can interrupt or corrupt the process of communication between heterogeneous WSNs. There is a need for a framework that can handle all scenarios in a transparent way so developers can adapt to each WSN. The closest implementation to that perception seems to be the Paho Project by Eclipse.

2. Future work

Our next challenges will be to try the Paho approach on the end-nodes as well as give a try on the Matter Protocol approach. Examining these approaches as well we can conclude if the forwarders, we created can get optimized.

Further analysis could be made on the evaluation tests in different QoS cases, where it may have a direct effect on the round-trip time (RTT) but provide more reliability to the exchanged data. The same study could be applied by using another data exchange protocol, such as CoAP and a comparative study may be resultant if compared to the results of this thesis.

Comparison could be made on performance between using the python library “Asyncio” instead of “Select” on the “forwarding scripts” due to cases of bottleneck suspicions we had regarding this section of the implementation and finally were proved right.

Chapter Seven - References

Abbreviations

M2M - Machine to machine
WSN - Wireless Sensor Network
CoAP - Constrained Application Protocol
OMA - Open Mobile Alliance
LWM2M - Lightweight Machine to Machine
MQTT - MQ Telemetry Transport
PUB / SUB - Publish / Subscribe
MQTT-SN - MQ Telemetry Transport for Sensor nodes
GW - Gateway
QoS - Quality of Service
RSMB - Really Small Message Broker
RF24 - Radio Frequency 2.4 GHz
BLE - Bluetooth Low Energy
GATT - Generic Attribute Profile
GAP - Generic Access Profile
SPI - Serial peripheral Interface
I2C - Inter-Integrated Circuit
RTT - Round trip time

References

[1] Fakhraddin, H. (2019). Toward IoT : Implementation of WSN based MQTT Data Protocol (Dissertation). Retrieved from http://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-92453

[2] da Rocha H., Monteiro T.L., Pellenz M.E., Penna M.C., Alves Junior J. (2020) An MQTT-SN-Based QoS Dynamic Adaptation Method for Wireless Sensor Networks. In: Barolli L., Takizawa M., Xhafa F., Enokido T. (eds) Advanced Information Networking and Applications. AINA 2019. Advances in Intelligent Systems and Computing, vol 926. Springer, Cham. https://doi.org/10.1007/978-3-030-15032-7_58

[3] U. Hunkeler, H. L. Truong and A. Stanford-Clark, "MQTT-S — A publish/subscribe protocol for Wireless Sensor Networks," 2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08), Bangalore, 2008, pp. 791-798, doi: 10.1109/COMSWA.2008.4554519.

[4] Y. Upadhyay, A. Borole and D. Dileepan, "MQTT based secured home automation system," 2016 Symposium on Colossal Data Analysis and Networking (CDAN), Indore, 2016, pp. 1-4, doi: 10.1109/CDAN.2016.7570945.

[5] Stanford-Clark, A., & Truong, H. L. (2013). Mqtt for sensor networks (mqtt-sn) protocol specification. International business machines (IBM) Corporation version, 1(2).

[6] M, S. (2020, March 23). M2M/ IoT Protocols: An Introduction. Retrieved October 19, 2020, from https://www.happiestminds.com/blogs/m2m-iot-protocols-an-introduction/

[7] Deepsubhra Guha Roy, Bipasha Mahato, Debashis De, Rajkumar Buyya, Application-aware end-to-end delay and message loss estimation in Internet of Things (IoT) — MQTT-SN protocols, Future Generation Computer Systems,Volume 89,2018, Pages 300-316,ISSN 0167-739X,https://doi.org/10.1016/j.future.2018.06.040.

[8] *Esp32 Bluetooth Low Energy (BLE) on Arduino Ide*. Random Nerd Tutorials. (2019, June 4). Retrieved January 31, 2022, from https://randomnerdtutorials.com/esp32-bluetooth-low-energy-ble-arduino-ide/

[9] Afaneh, M. (2020, May 19). *The basics of Bluetooth Low Energy (BLE)*. Novel Bits. Retrieved January 31, 2022, from https://www.novelbits.io/basics-bluetooth-low-energy/

[10] Afaneh, M. (2020, June 16). *Bluetooth 5 speed: How to achieve maximum throughput for your ble application*. Novel Bits. Retrieved January 31, 2022, from https://www.novelbits.io/bluetooth-5-speed-maximum-throughput/

[11] Last Minute Engineers. (2020, December 18). In-depth: How NRF24L01 wireless module works & interface with Arduino. Last Minute Engineers. Retrieved January 31, 2022, from https://lastminuteengineers.com/nrf24l01-arduino-wireless-communication/

[12] Cabé, B. (2020, September 1). *Using MQTT-SN over Ble with the BBC Micro:bit*. Benjamin Cabé. Retrieved January 31, 2022, from https://blog.benjamin-cabe.com/2017/01/16/using-mqtt-sn-over-ble-with-the-bbc-microbit

[13] Team, T. H. M. Q. (n.d.). *MQTT Essentials - Introducing the MQTT protocol* . Introducing the MQTT protocol - MQTT essentials: Part 1. Retrieved January 31, 2022, from https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/

[14] The Complete Mqtt Broker Selection Guide. (n.d.). Retrieved January 31, 2022, from https://www.catchpoint.com/network-admin-guide/mqtt-broker

[15] Eurotech, International Business Machines Corporation (IBM). (n.d.). *MQTT V3.1 Protocol Specification*. MQTT v3.1 Protocol Specification. Retrieved January 31, 2022, from https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html

[16] Team, T. H. M. Q. (n.d.). *MQTT Essentials - Quality of Service*. Quality of service 0,1 & 2 - mqtt essentials: Part 6. Retrieved January 31, 2022, from https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/

[17] Bormann, C., Castellani, A. P., & Shelby, Z. (2012). Coap: An application protocol for billions of tiny internet nodes. IEEE Internet Computing, 16(2), 62-67.

[18] C. Jennings, Z. Shelby, and J. Arkko, (2011) “Media Types for Sensor Markup Language (SENML),” IETF Internet draft, work in progress.

[19] *LWM2M: Lightweight M2M*. Software AG. (n.d.). Retrieved February 2, 2022, from https://www.softwareag.com/en_corporate/resources/what-is/lwm2m.html

[20] *Optimized high speed nRF24L01+ driver class documentation*. Optimized high speed nRF24L01+ driver class documentation: Optimized High Speed Driver for nRF24L01(+) 2.4GHz Wireless Transceiver. (n.d.). Retrieved January 2, 2022, from https://nrf24.github.io/RF24/

[21] *MQTT-SN Messages Library*. Bitbucket - MQTT-SN Messages Library. (n.d.). Retrieved February 2, 2022, from https://bitbucket.org/MerseyViking/

[22] Roldán-Gómez, J., Carrillo-Mondéjar, J., Castelo Gómez, J. M., & Ruiz-Villafranca, S. (2022). Security Analysis of the MQTT-SN Protocol for the Internet of Things. *Applied Sciences*, *12*(21), 10991.

[23] Ibr-Alg. (n.d.). *IBR-alg/wiselib: A generic algorithms library for heterogeneous, distributed, embedded systems.* GitHub. Retrieved February 2, 2022, from https://github.com/ibr-alg/wiselib

[24] R. U. Islam, K. Andersson and M. S. Hossain, "Heterogeneous wireless sensor networks using CoAP and SMS to predict natural disasters," 2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Atlanta, GA, USA, 2017, pp. 30-35, doi: 10.1109/INFCOMW.2017.8116348.

[25] R. Bruno and S. Bolettieri, "Design and Implementation of a COAP-Based Broker for Heterogeneous M2M Applications," 2018 IEEE International Congress on Internet of Things (ICIOT), San Francisco, CA, USA, 2018, pp. 1-8, doi: 10.1109/ICIOT.2018.00008.

[26] Harshada Chaudhari, (2015) *Raspberry Pi Technology: A Review*, International Journal of Innovative and Emerging Research in Engineering, Vol.2, Issue 3

[27] Raspberry Pi. (n.d.). *Teach, learn, and make with Raspberry Pi*. Raspberry Pi. Retrieved February 1, 2022, from https://www.raspberrypi.org/

[28] Dennis, A. K. (2016). Raspberry pi computer architecture essentials. Packt Publishing Ltd.

[29] *Matter protocol specifications*. (n.d.). Connectivity Standards Alliance. Retrieved July 29, 2023, from https://csa-iot.org/wp-content/uploads/2022/11/22-27349-001_Matter-1.0-Core-Specification.pdf