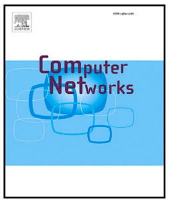
Computer Networks 197 (2021) 108338

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



Performance evaluation of CoAP and MQTT with security support for IoT environments

Victor Seoane, Carlos Garcia-Rubio, Florina Almenares, Celeste Campo [*]

University Carlos III of Madrid, Av. de la Universidad, 30, Leganés (Madrid), Spain

ARTICLE INFO

Keywords:
Internet of Things
CoAP
MQTT
Security
Performance evaluation

ABSTRACT

World is living an overwhelming explosion of smart devices: electronic gadgets, appliances, meters, cars, sensors, camera and even traffic lights, that are connected to the Internet to extend their capabilities, constituting what is known as Internet of Things (IoT). In these environments, the application layer is decisive for the quality of the connection, which has dependencies to the transport layer, mainly when secure communications are used. This paper analyses the performance offered by these two most popular protocols for the application layer: Constrained Application Protocol (CoAP) and Message Queue Telemetry Transport (MQTT). This analysis aims to examine the features and capabilities of the two protocols and to determine their feasibility to operate under constrained devices taking into account security support and diverse network conditions, unlike the previous works. Since IoT devices typically show battery constraints, the analysis is focused on bandwidth and CPU use, using realistic network scenarios, since this use translates to power consumption.

1. Introduction

World is living an overwhelming explosion of smart devices, electronic gadgets connected to the Internet to extend their capabilities. The current trends of users and manufacturers show even the fridges, the furniture or the lamps are starting to be connected to the network. This unprecedented number of *physical objects or "things" embedded with electronics, software, sensors, actuators, and connectivity to enable objects to exchange data with the manufacturer, operator and/or other connected devices* is known as Internet of Things (IoT) [1].

The network has to face the first major problem associated to the IoT: a huge amount of connected devices. This problem implies that network solutions should be easily scalable and interoperable to assure the correct implementation of IoT [2]. But handling a huge number of devices is not enough, the quality of the performance and security are also important. The quality is typically represented by the Quality of Service (QoS).

In addition to the difficulty of handling such a huge number of network nodes while assuring the required quality, these devices are heterogeneous and often present additional constrains that limit their operation; mainly, power consumption and storage constraints. For these reasons, lightweight communication protocols have been developed and implemented to deal with these challenges. Focusing on the application layer, there is no a unique protocol which serves all the applications [3]: the choice is highly situation-dependent. Specifically, Constrained Application Protocol (CoAP) specified in [4] and Message Queue Telemetry Transport (MQTT) defined in [5,6] are the two most used standards today, and they stand as the two most powerful protocols for IoT, although other protocols have been also used to a lesser extent, such as Advanced Message Queuing Protocol (AMQP), Data Distribution Service (DDS), Extensible Messaging and Presence Protocol (XMPP) or even Hypertext Transfer Protocol (HTTP).

As the application layer is decisive to define the reliability, the latency and the overhead of the connection, this work aims to offer a detailed analysis of the advantages and drawbacks offered by the two most popular protocols for the application layer, CoAP and MQTT. The modes of operation of both protocols and their differences are explored through the use of open-source implementations of both protocols. Furthermore, since security vulnerabilities of the IoT communications can lead to threats such as message forgery, tampering or eavesdropping, guidance from RFC7925 [7] will be followed to secure the data exchange carried out by IoT devices. In particular, the proposed cipher suites recommended by Tschofenig et al. [7] will be tested in terms of network overhead and CPU usage increase, comparing them to the non-secured scenario where the communications are not ciphered. These two factors will determine the feasibility of the ciphering and authentication methods in such a constrained environment, showing the trade-off between security and performance.

[*] Corresponding author.
E-mail addresses: vseoane@it.uc3m.es (V. Seoane), cgr@it.uc3m.es (C. Garcia-Rubio), florina@it.uc3m.es (F. Almenares), celeste@it.uc3m.es (C. Campo).

https://doi.org/10.1016/j.comnet.2021.108338
Received 10 April 2021; Received in revised form 24 June 2021; Accepted 14 July 2021
Available online 28 July 2021

The main goal of this paper is to perform a detailed analysis of the latency, bandwidth, and CPU usage, analysing their capabilities and their feasibility to operate under device constraints. For that, a realistic network scenario was deployed using a Raspberry Pi, as network node, and a switch to emulate and configure the diverse network conditions (e.g. packet losses).

This paper is organized as follows: Section 2 describes the IoT protocols used in this study: CoAP and MQTT, highlighting advantages, limitations, implementations and features analysed. Then, Section 3 analyses the related and previous work, identifying our contributions beyond the literature. Section 4 explains the network scenario, configurations, development and experiments performed to evaluate CoAP and MQTT. We present an analytical study of these protocols in Section 5, and then the results of the experiment are shown, discussed and compared in Section 6. Finally, Section 7 concludes the paper and sketches our future work.

2. IoT application layer protocols

As said before, there are several standardized protocols for IoT application layer to lead Machine-to-Machine (M2M) communication. This section briefly presents the two most popular protocols: CoAP and MQTT.

2.1. CoAP protocol

CoAP is defined in the RFC7252 [4] as a lightweight and simple application protocol. It is a request–response application protocol based on an asynchronous exchange of messages with optional reliability, as it runs over User Datagram Protocol (UDP) which does not offer any reliability.

In a CoAP network we have two types of nodes (Fig. 1): CoAP servers, usually constrained devices (sensors or actuators), that can be accessed or controlled using a REST API, and CoAP clients, devices that want to retrieve some information or request some action from the server. HTTP clients could also communicate with CoAP servers using a proxy, which communicates itself with the CoAP server using the CoAP protocol. The way clients and servers discover each other is outside the scope of this paper, but fundamentally they can do it in two ways, through DNS (mDNS and DNS-SD), or through CoAP resource discovery (which can be multicast or based on directory).

CoAP defines four types of messages:

- **Confirmable** (CON). These messages require an acknowledgement to be sent by the other communicating part. When the network does not cause packet losses, each CON message trigger exactly one return message of type Acknowledgement or type Reset. If no ACK or RST is received, after a certain time the CON message is assumed to be lost and it is retransmitted.
- **Non-confirmable** (NON). These messages do not require an acknowledgement, offering no reliability.
- **Acknowledgement** (ACK). An ACK message acknowledges that a particular CON message did arrive. It is also able to carry the response to the request, a process known as piggybacked response.
- **Reset** (RST). This message reports that a particular message (CON or NON) was received, but it cannot be properly processed. This event usually happens when the receiver has rebooted and has forgotten some state that is required to correctly interpret the message. Provoking a Reset message (e.g., by sending an Empty CON message) is also useful to check of the liveness of an endpoint: *CoAP ping*.

The format of the messages of the protocol has been designed to be simple and light in order to reduce the typical overhead caused by the headers of the protocols (see Fig. 2). All the messages start with a fixed-size 4-byte header, which is mandatory. Then, they could be followed by a variable-length Token value (between 0 and 8 bytes), a sequence of zero or more CoAP Options in Type-Length-Value (TLV) format and an optional payload. Only the 4-byte header is mandatory, the rest is optional.

The fields in the header include: *Version* (`Ver`), which identifies the version of the CoAP protocol; *Type* (`T`) indicates the type of CoAP message: 0 for CON, 1 for NON, 2 for ACK and 3 for RST; *Token Length* (`TKL`) determines the length of the variable-length token field; *Code* is an 8-bit field which indicates if the message is a request (0), a success response (2), a client error response (4) or a server error response (5), as well as indicating request method or response code; *Message ID* is used to match messages ACK or RST with the CON message that caused them.

It is important to notice that the *message ID* does not match requests and responses, but the request (or the response) with its ACK or RST. It is also used to detect message duplication. The token value should be used instead if the matching request–response is required. This is the purpose of the optional token field.

The options field allows to add a list of one or more options to the request/response (e.g. Content-Format, Max-Age, ETag...). Probably the most important options are `Uri-Host`, `Uri-Path`, `Uri-Port` and `Uri-Query`. These options allow to specify the target resource of a request and to locate it inside the server's hierarchy through the composition of an Uniform Resource Identifier (URI). The schemes for URIs composition are the following ones (the usage of one or another depends on whether the communication is being secured using Datagram Transport Layer Security (DTLS) over UDP or not:

```
coap-URI="coap:""//" host[ ":"port]path-abempty["?" query]
coaps-URI="coaps:""//" host[ ":"port]path-abempty["?"query]
```

Focusing on the requests, there exist four methods: `GET` to retrieve the current information specified through the request URI, `POST` to create or update a resource, `PUT` to update or create a resource with the given representation, and `DELETE` to remove a resource identified by the URI. The method of the requests is specified in the `Code` field of the CoAP header [4].

Typically, for the IoT, the most frequent method is GET. It is used to retrieve information from a node of the network. In the default behaviour, the node replies one answer with the current measurement, however an optional observe extension has been defined in [8] to instruct the server to send notifications to the client whenever the state of the resource changes. This is similar to the publish/subscribe model used in MQTT and MQTT for Sensor Networks (MQTT-SN), although a bit different (there is no broker). We will not consider the observe option in this work.

Having understood the types of CoAP messages and the messages exchange process, three scenarios can be defined for a simple retrieval of information (GET) depending on the required level of reliability and the immediate availability of the response at the server. These three scenarios are shown in Fig. 3 and their impact on the network are analysed in this work.

The two firsts ones deal with Confirmable messages. In the scenario A, the response is carried along with the Acknowledgement (piggybacked), while in the scenario B there is a separated response. Both situations could happen and it depends on the implementation of the server and on the immediate availability of the information. However, servers are expected to prefer piggybacked responses to save network resources both at the client and at the server. Finally, the scenario C deals with Non-confirmable messages, so no ACK is needed and the response is also sent in a NON message.

Scenarios A and B provide reliability. In the scenario A, if the GET message is lost the transmitter will notice as the Acknowledgement will not arrive. Then, it will be able to retransmit the message after a certain period of time. If the Acknowledgement is the lost message, the timer of

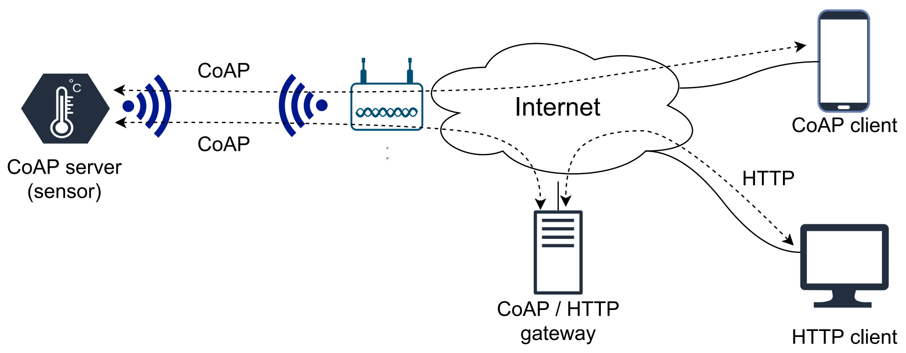


Fig. 1. CoAP system overview.

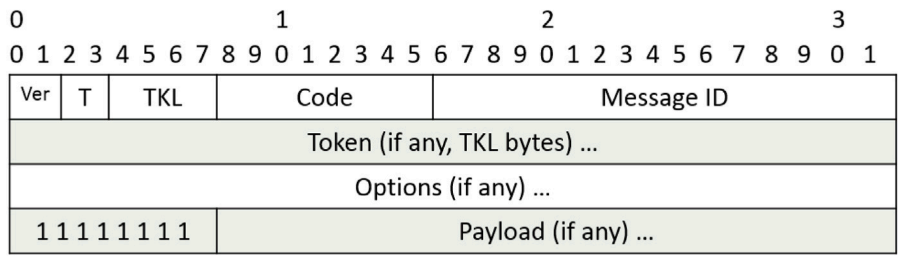
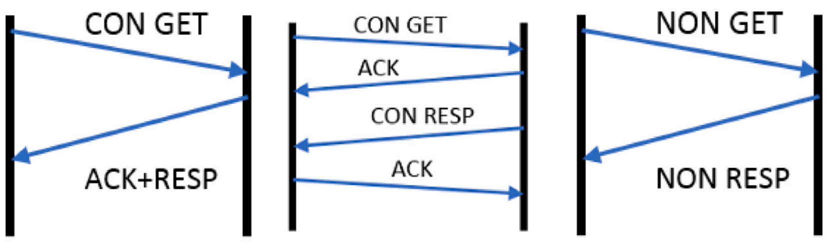


Fig. 2. CoAP message format.



(a) Scenario A (b) Scenario B (c) Scenario C

Fig. 3. Possible scenarios for a CoAP retrieval of information (GET).

the transmitter will eventually expire, retransmitting the GET message. For the scenario B, same reasoning applies for both CON messages (the request and the response).

By default, CoAP messages are transported over UDP, but they can also be sent over DTLS. According to RFC7252, four security modes are defined:

- **NoSec**. DTLS is disabled. If needed, security should be provided at lower layers, using IP Security (IPsec).
- **PreSharedKey (PSK)**. DTLS is enabled and the device keeps a list of pre-shared keys associated to the nodes with which can communicate using these keys. Key derivation functions are used to obtain the keys that secure the connection. This scheme corresponds to symmetric cryptography.
- **RawPublicKey**. DTLS is also enabled in this mode. The device has an asymmetric key pair (public and private) that has been validated somehow (out-of-band). Asymmetric cryptography is used to secure the session key exchange.
- **Certificate**. Similar to the previous one, but in this case, the public key pair comes with an X.509 certificate that binds it to its subject and has been signed by some trusted authority, compliant with a Public Key Infrastructure (PKI).

2.1.1. Strengths and limitations

CoAP was designed to operate under constrained devices, lightening the message exchanges and saving resources at the network nodes. The main advantages of the use of CoAP for the IoT can be summarized as follows [9]:

- **Delay reduction** for the data transmission. This is achieved by using a compact header and carrying the messages over UDP instead of Transmission Control Protocol (TCP), as HTTP does, for example.
- **Minimization** of **power consumption** as a result of the overhead reduction achieved by the use of UDP and a very compact header.
- Thanks to **asynchronous** data push, devices can sleep most of the time and just send information when there is a state change. This leads to a further reduction in power consumption.
- **Reduced complexity** compared to protocols such as HTTP, relaxing the hardware requirements for IoT devices.
- Certain **flexibility** for achieving reliability. CoAP defines the Confirmable messages to counteract (if needed) the unreliable behaviour of UDP, offering the devices and the users the flexibility to use them or not.
- **End-to-end** principle, because no intermediate brokers or gateways are needed.
- **Interoperability** with existing standards (e.g. HTTP) and networks.

These strengths of the CoAP protocol also lead to some of its possible limitations [10]:

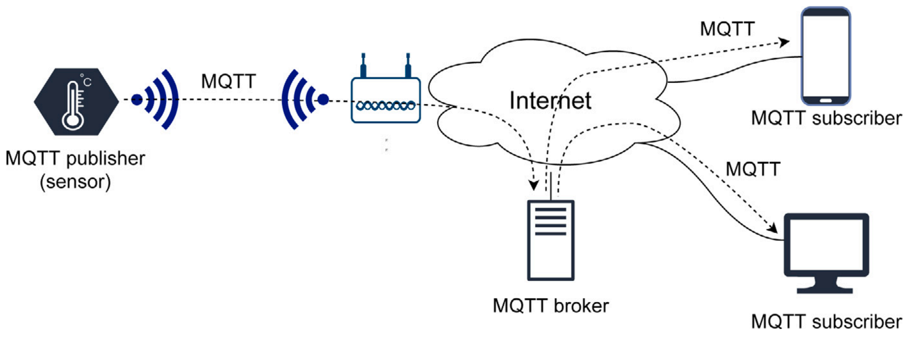


Fig. 4. MQTT system overview.

- **Unreliability**. If Non-confirmable messages are used, reliability is not achieved as the protocol runs over UDP. Moreover, the use of CON messages only confirm message arrival, not taking into account any possible errors.
- Lack of **congestion control** for Non-confirmable messages, giving them the possibility to overrun the network.
- **Immaturity**: the protocol is gaining popularity but it is still evolving. The open-source distribution is leading to different implementations of the protocol that might not be interoperable with each other.

2.1.2. Implementations

There exist several open-source as well as private implementations of CoAP. They can be grouped according to the devices they are designed for (constrained devices, servers, browser-based applications or smartphones) [11]. For this study, the focus was on implementations for constrained devices and open source, which are summarized as follows:

- **erbium**[1] for the Contiki operating system, which is a not popular OS and lacks of multiplatform. One of its biggest disadvantage is that there is no available documentation.
- **libcoap**[2] implemented in the C programming language. It can be used both in Contiki and POSIX systems, being multiplatform and interoperable. It is well-documented and provides some examples. It is one of the most popular implementations.
- **microcoap**[3] implemented also in C. In contrast to libcoap, it can only run over Arduino or POSIX systems. The library does not implement the whole RFC.
- **cantcoap**[4] is an implementation in C, which focuses on the encoding and decoding of messages, leaving the messages control (i.e., timeouts, retransmissions, messages matching) for the user. This added complexity has no interest for the purpose of this work.
- **californium**[5] is implemented in Java programming language. It is focused on the server side, although it can be used to implement a client. It is not designed for constrained devices, so complexity and resource consumption could be bigger than the ones especially designed for constrained devices.

For our work the choices were between libcoap and californium. The rest of implementations were discarded due to already exposed reasons, i.e., unknown operating systems, lack of multiplatform, excessive complexity and/or lack of documentation. So, the final decision was made according to the systems they were designed for; therefore, **libcoap** was selected for being especially designed for constrained devices (that are more frequent in the IoT scene).

2.2. MQTT protocol

MQTT aims to be simple, open, lightweight and to offer a high bandwidth-efficiency, being a good candidate for machine communications in constrained environments. It is an open standard of OASIS [5].

MQTT is a *publish/subscribe* messaging protocol specially suitable for the IoT. Its main feature is the reliability as it runs over TCP or over similar protocols which offer the same features: ordered, lossless and bi-directional communications [5].

As other publish/subscribe protocols, it has a topic-based architecture: exchanged data is classified by hierarchically organized topics in such a way that every message is associated with a topic. The clients publish the messages to a certain topic and this message is received by every other client subscribed to this topic (selective messaging). Then, messages are only routed to destinations with matching topic interests.

The topic-based publish/subscribe protocols, like MQTT, typically present a physical architecture consisting of three types of nodes: the publishers, the subscribers and the broker [12] (Fig. 4):

- A **publisher** is any client that publishes a message associated to any topic, typically a sensor. It is producer of the data. However, the publisher condition is not exclusive: a client can be publisher and subscriber at the same time of different topics or even of the same topic.
- A **subscriber** is any client that requests for information subscribing to certain topics: it is the consumer of the data. As the publisher case, subscriber condition is not exclusive.
- A **broker** is a device that acts as the central server for the information. It is in charge of receiving and maintaining the topic interests of the subscribers and of receiving and routing the published messages to the clients subscribed to its topic. It acts as an intermediate filter for the clients, selecting the messages and filtering by topic to only send the relevant information for every client.

The main advantage of the architecture shown in Fig. 5 is that it leaves all the complexity for the broker, so the clients can be really simple and lightweight, because implementing this architecture drastically reduces the number of connections that a client must handle to communicate with all the nodes of the scenario: just one (with the broker). By contrast, the broker must handle a high number of connections but, typically, this is not a problem since the broker does not have the constraints that the clients have.

The MQTT messages exchanged, between such nodes, consists of a series of messages called MQTT Control Packets. The structure of a MQTT Control Packet consists of three elements: a fixed header, a variable header and a payload. The fixed header is mandatory and it is present in all MQTT Control Packets. The variable header and the payload are only suitable for some kinds of MQTT Control Packets [5]. The variable header can contains additional information, depending of the type of message. The payload contains the application information, the data to be exchanged.

[1] https://github.com/contiki-ng/contiki-ng/wiki/Documentation:-CoAP
[2] https://libCoAP.net/
[3] https://github.com/1248/microcoap
[4] https://github.com/staropram/cantcoap
[5] https://www.eclipse.org/californium/

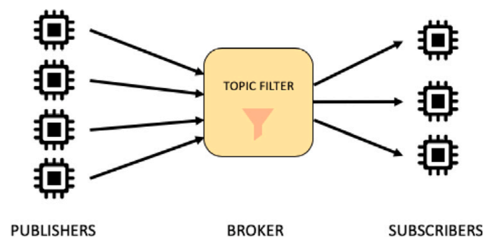


Fig. 5. MQTT data exchange architecture.

The fixed header format is shown in Fig. 6. The MQTT Control Packet type field defines the types of packets such as connection request (i.e., CONNECT (1)), publish (i.e., PUBLISH (3)), subscribe (i.e., SUBSCRIBE (8)), unsubscribe (i.e., UNSUBSCRIBE (10)), ping (i.e., PINGRESP (13) and PINGREQ), disconnect (i.e., DISCONNECT (14)) and acknowledgements (i.e., CONNACK (2), PUBACK (4), SUBACK (9) and UNSUBACK (11)). Besides, values 5, 6 and 7 define the PUBREC, PUBREL and PUBCOMP messages respectively. These messages are additional messages triggered by a PUBLISH message when certain levels of QoS are required. The following four bits are flags specific to each type of MQTT Control Packet, e.g., with PUBLISH messages, these bits allow to specify the level of QoS for the data exchange.

In a typical scenario, both the subscribers and the publishers start their connection to the server at any time by the sending of a CONNECT message and the reception of the corresponding CONNACK. Once they are connected, each client subscribes to the topics it is interested in with the sending of a SUBSCRIBE message and the reception of the corresponding SUBACK. Any other client publishes information on a topic with the sending of a PUBLISH message to the broker and then, the broker forwards the PUBLISH message to the clients subscribed to its topic. In these PUBLISH messages exchanged (both client→broker and broker→client), three modes of QoS can be defined [12]:

- **QoS 0** (At most once delivery). The messages are delivered according to the capability of the network with no MQTT retransmission or acknowledgement. The PUBLISH message is delivered at most once. It is the simplest mode and it offers the lowest overhead: a single PUBLISH message for a single exchange.
- **QoS 1** (At least once delivery). The PUBLISH messages must be acknowledged by a PUBACK message, otherwise they are retransmitted. This QoS level assures that the message reaches the destination, but this could happen more than once due to retransmissions.
- **QoS 2** (Exactly once delivery). The PUBLISH message is the first message of a four-way handshake that assures that the message arrives exactly once. The messages that form the handshake are PUBLISH, PUBREC, PUBREL and PUBCOMP.

By default, MQTT messages are transported over TCP, but they can also be carried over Transport Layer Security (TLS) acting as an intermediate layer to provide security services.

2.2.1. *Strengths and limitations*

The main advantages of the use of MQTT as protocol for the IoT can be summarized with the following list:

- **Low overhead**, due to the use of a compact header and small packets, making the protocol feasible for low bandwidth connection.
- **Reliability** as the protocol runs over TCP, the ordered arrival of the messages is assured by the underlying layer of the network.
- **QoS flexibility**. The protocol defines different QoS levels to fit different network requirements. Moreover, the level of QoS is chosen for every topic subscription for every client, leading to a great flexibility.
- **Flow and congestion control** provided by TCP, avoiding possible device/network overflow problems.
- **Selective messaging**. Through a topic-based architecture, MQTT allows to selectively choose the information a node want to receive.
- **Client simplicity**. Due to the architecture of the protocol, the implementation of the client is very simple: the complexity is left for the broker. This enhances the energy efficiency.

However, MQTT also presents some important limitations:

- **Need of a broker**. The simplicity at the clients is achieved at the expense of having a broker that acts as intermediate keeping a high number of connections.
- **Connection-oriented**. The protocol runs over TCP so it is connection-oriented. The connections between the broker and the clients are kept alive, worsening the energy efficiency. For this reason, MQTT-SN was designed for very constrained devices. This modification allows the clients to run MQTT over UDP at the expense of including one additional device in the network: a gateway that connects the MQTT-SN devices with the broker and the rest of the MQTT network.
- **Security issues**. Since MQTT does not add secure communication support by default, TLS must be added as intermediate layer between MQTT and TCP to secure the communications.
- **Single point of failure**. The broker is a single point of failure in the network, if it goes offline the whole network is useless.

2.2.2. *Implementations*

There exist several open-source implementations of MQTT, in particular:

- **paho MQTT**.[6] Multiplatform implementation of MQTT protocol developed by Eclipse. Its source code is available in Java, C, Python and some others programming languages. It is a client library so it does not include the implementation of the broker, the most important node in the MQTT architecture. It is well-documented and it is regularly updated.
- **mosquitto**.[7] Mosquitto is a C implementation of MQTT broker and client. It is also property of Eclipse Foundation. Although the choice of the programming language is limited to C, it is a complete implementation (broker+client) and it is up to date and well-documented. Moreover, it is one of the most popular implementations in the literature.
- **moquette**.[8] It is a Java implementation of a MQTT broker. Its software licence is property of Apache. It is usually used in conjunction with Paho to build up a complete system (paho for clients, moquette for the broker).

As **mosquitto** offers an implementation for the whole scenario using just a single implementation, it was the final chosen one.

2.3. *Summarizing features of IoT protocols*

A summary of the main features of each IoT protocol described previously is shown in Table 1.

[6] https://www.eclipse.org/paho/
[7] https://mosquitto.org/
[8] https://moquette-io.github.io/moquette/

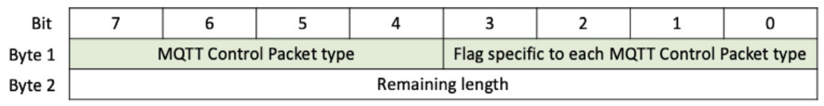
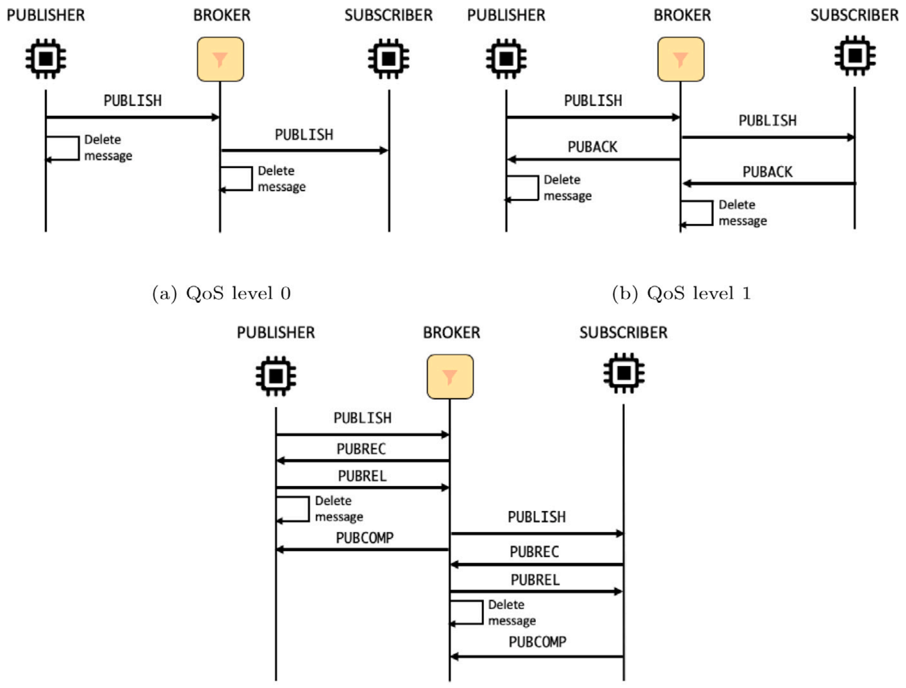


Fig. 6. MQTT fixed header format.



(a) QoS level 0 (b) QoS level 1

(c) QoS level 2

Fig. 7. MQTT Quality of Service (QoS) levels.

Table 1
Summary of IoT protocols, CoAP and MQTT.

	CoAP	MQTT
Standardization body	IETF	OASIS
Architecture	Request–Response, Resource “observer"	Publish–Subscribe
Transport	UDP	TCP (UDP can be used in MQTT-SN)
Security	DTLS (transport), OSCORE (Object Security for Constrained RESTful Environments - application)	TLS (transport), client authentication
Header	4B+variable	2B+variable
Methods for RESTful	GET, POST, PUT, DELETE	No support
Message Types	4: CON, NON, RST, ACK	15: CONNECT, CONNACK, PUBLISH, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBSCRIBE, SUBACK, UNSUBSCRIBE, UNSUBACK, PINGREQ, PINGRESP, DISCONNECT, AUTH
Resource id	URI	Topics
Strengths	lightweight, end-to-end principle, asynchronous and synchronous, flexibility and reliability (through CON messages), interoperability	asynchronous, reliability, Client simplicity, selective messaging
Limitations	unreliable transport	broker-dependent, only connection-oriented

2.4. At what layer should we deal with security support?

Security is always required and it is recommended to be used at different layers. So a security protocol at the application layer may be used in conjunction with security protocols at the transport layer, doubling up security services, such as confidentiality, integrity and/or authentication. Nevertheless, in IoT scenarios, overhead added by security protocols must be optimized, because it is a critical issue due to IoT devices can be constrained in terms of memory, storage, processing, energy, etc.

CoAP specification defines the use of DTLS, but also Object Security for Constrained RESTful Environments (OSCORE) is defined to be used at the application layer. MQTT does not define a standard mechanism to provide security at the application layer (except an authentication control packet, AUTH, and related properties). In scenarios where IoT and non-IoT devices interact, the use of TLS/DTLS provides interoperability. Most CoAP implementations supports only DTLS/TLS. Likewise, MQTT implementations are compatible with the use of TLS. In addition, DTLS/TLS comes with a builtin key exchange protocol and authentication for different applications' requirements, unlike OSCORE that relies on pre-shared keys. This does not currently have a native key exchange mechanism, although standardized mechanisms are being evaluated by the IETF, for instance, Ephemeral Diffie–Hellman Over COSE (EDHOC) [13]. This could become a system vulnerability, as well as not providing forward secrecy.

On the other hand, although OSCORE has a moderately better performance than DTLS 1.2 regarding radio transmission overhead, RTT and memory usage according to [14], DTLS supports short headers, which have quite small overhead; therefore, overhead would be very similar between DTLS short headers, DTLS v1.3 and OSCORE, according to the IETF LWIG Working Group [15].

OSCORE encrypts only the data that are part of the payload, thus allows decreasing overhead and increasing bandwidth usage and battery lifetime of the device. It is recommended to be used when intermediary (non-trusted) proxies are part of the end-to-end communication; besides, it is also proposed for secure one-to-many group communications.

3. Related work

In this section, related works focused on the performance of the protocols for the application layer of IoT are analysed. Firstly, we analyse the works in chronological order, and then we group studies developed with a Raspberry Pi, similar to our work.

In [12], Thangavel et al. propose the design of a common middleware that supports CoAP and MQTT to offer a common programming interface, easing the interoperability of the Wireless Sensors Networks (WSNs). Such middleware was implemented and the performance of both protocols was tested. The performance indicators used were the *delay* and the *total data transferred per message* and the experiments were conducted for different loss rates. The implementations used for CoAP and MQTT were open-source implementations: `libcoap` and `mosquitto`, respectively. The results of the experiments revealed that MQTT has lower delay than CoAP at low packet loss rates and higher delay than CoAP at high loss rates. Besides, for small message lengths and loss rates equal to or less than 25%, CoAP generates lower additional traffic than MQTT to ensure reliability [12]. Although this work offers a very complete hardware description of the network scenario and a general analysis of CoAP and MQTT in lossy environments, it does not consider the different modes of operation of both protocols neither the secure transmission of messages. In particular, the trade-offs security versus performance and reliability versus performance are not examined.

In [16], CoAP performance is assessed using californium libraries and emulating the data transmission in a Mobile Ad-Hoc Network (MANET). The assessment is compared with HTTP, therefore, the results show that CoAP performance is better than HTTP with respect to delivery rate, delay, and overhead, mainly when using confirmable messages to transmit small sized data.

Nitin Naik [17] explains the problem that arises with the different messaging protocols for the IoT: no one supports all the requirements of all types of IoT systems. For this reason, it is important to analyse the characteristics of the available messaging protocols, focusing on the strengths and the limitations of each one. The work aims to be a tool for the users to decide the appropriate messaging protocol for their IoT specific system according to different balances between features (e.g., delay, bandwidth, power consumption, latency...). The protocols used for the comparison were three protocols specifically designed for IoT: MQTT, AMQP, CoAP, and their natural reference protocol HTTP. This work results to be a good guide for an ordinary user to choose among the four cited messaging protocols. However, it does not consider dynamic network conditions: the network is considered as lossless.

Morabito et al. evaluated the performance of CoAP, MQTT, and HTTP through 4G and Wi-Fi connections in vehicular scenarios [18]. They compared two service provisioning approaches: cloud-based vs. edge-based, transmitting small-sized messages from an in-car on-board unit. Their results showed CoAP outperforms MQTT (QoS 1 and 2) and HTTP from throughput and latency perspective in different case studies of the Vehicular Network (VN).

Larmo et al. [19] looked at the impact of the protocol stack on performance over a narrowband IoT (NB-IoT) link. They used a simulated scenario of NB-IoT protocol stack (with 7 base stations and 3 sectors) in which small reports are sent from a sensor device to a central cloud storage over a last mile radio access link. The results found that while CoAP performs consistently better in terms of throughput, latency, coverage, and system capacity, MQTT also works when the system is less loaded.

Finally, we highlight four papers that use Raspberry Pi for measurement the performance of CoAP and other IoT protocols. In [20], Thota and Kim discuss and analyse the efficiency, usage, and requirements of MQTT and CoAP, using a Raspberry Pi with Raspbian OS and a temperature sensor in a simple experiment which contains one publisher, server and broker. They conclude that as the size of the message being sent increases, CoAP handles more data than MQTT. In [21], Tandale et al. compare the performance of CoAP, MQTT and HTTP REST using a Raspberry Pi 3 and the aiocoap,[9] mosquitto, and django[10] implementations of the protocols. They compare the number of bytes consumed and the delay for each of the protocols under two different networks: 4G and high speed broadband connection. The results show that CoAP is more efficient in terms of time and bandwidth for smaller payloads and its performance deteriorates as payload size increases. In [22], Liri et al. investigated and analysed four protocols, namely, CoAP, MQTT, MQTT-SN and QUIC, to understand the overhead of obtaining data from an IoT device at a sink to potentially disseminate this data downstream. For evaluation, the experiments were performed in an emulated environment using VirtualBox VMs, as well as a subset of them that was also run using Raspberry Pi. They measure delay and total packets sent. Results show that in terms of overhead, CoAP is the most efficient protocol. Another key finding from the experiments is that for IoT protocols that use a fire-and-forget paradigm, such as CoAP NON, MQTT QoS 0 and MQTT-SN QoS 0, the wait or keep alive timers play a crucial role in performance. In these last works, the study is conducted using Raspberry-Pi, but they do not consider the effects of different security mechanisms in CoAP performance. Moraes et al. compared AMQP, CoAP (confirmable with piggybacked response), and MQTT (QoS 1) in terms of throughput, message size, and packet loss [23]. They deployed two scenarios with three sensors connected to a platform based on Raspberry Pi: the first experiment with a path (without network failures) and the second one with two paths/routes to the sensors (taking into account failures). The experiment results indicated CoAP protocol provides the best results, followed by MQTT.

Regarding our previous work, Martí et al. presented an energy consumption and network traffic study of CoAP and MQTT-SN [24]. The experiments presented evaluate the performance of these protocols with different network configurations. The results showed that the performance of these two light application protocols is pretty similar when the total transmission energy consumption is obtained. Nevertheless, MQTT-SN is more efficient than CoAP since its client nodes have less complexity than CoAP clients. In [25], our previous work showed results of the CoAP analysis. So, this paper extends this last work with MQTT measurements and comparing them with the obtained by CoAP.

Summarizing, Table 2 shows the differences of our study respect to the related works.

[9] aiocoap (https://github.com/chrysn/aiocoap) is an implementation of CoAP written in Python 3 using its native asyncio methods to facilitate concurrent operations while maintaining an easy to use interface. It supports DTLS, the observe option, as well as CoAP over TCP, TLS, and WebSockets, among other CoAP options.

[10] Django (https://github.com/django/django) is a Python-based free and open-source web framework.

Table 2
Comparison with related works.

Work	Security support	Network Conditions	Hardware-based IoT Scenario
Thangavel et al. [12]	No	loss rates	L2 switch, laptop (servers), BeagleBoard-xm (gateway/subscriber/publisher), netbook (Wanem)
Gao et al. [16]	No	loss rates, client mobility	No
Naik [17]	No	No (lossless)	No
Morabito et al. [18]	No	Vehicle speed	VN: Edge-server, Cloud-server, In-Car On-Board Unit (Raspberry Pi 3)
Larmo et al. [19]	No	NB-IoT stack in ns-3	No
Thota and Kim [20]	No	No	Desktop, Raspberry Pi
Tandale et al. [21]	No	No	Raspberry Pi, public cloud servers
Liri et al. [22]	No	No	Raspberry Pi, VMs (mainly)
Moraes et al. [23]	No	Network failures	Raspberry Pi, sensors
Our work	PSK, PKI	loss rates	L2 switch, Raspberry Pi, laptop

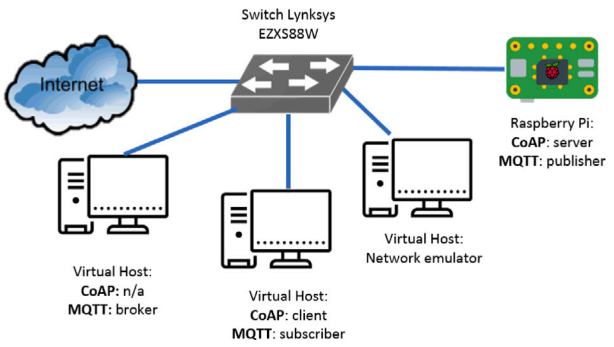

Switch Lynksys EZXS88W
Internet
Raspberry Pi: CoAP: server MQTT: publisher
Virtual Host: Network emulator
Virtual Host: CoAP: n/a MQTT: broker
Virtual Host: CoAP: client MQTT: subscriber

Fig. 8. Physical network scenario for CoAP and MQTT analysis.

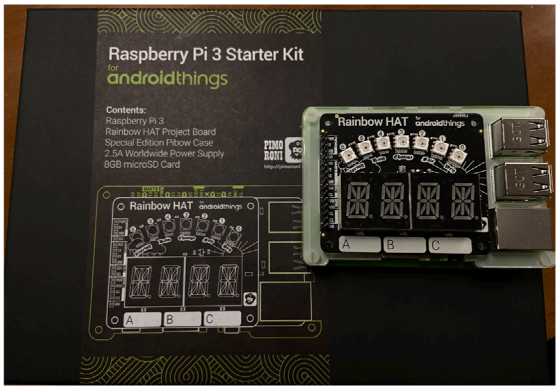


Fig. 9. Raspberry Pi 3 Starter Kit for Android Things.

4. Methodology

This section will depict the process of designing and implementing the scenario to test the CoAP and MQTT protocols. This process implies the deployment of the network scenario, the study of the possibilities offered by the selected well-known implementations of the protocols (i.e., libcoap for CoAP and mosquitto for MQTT), the configuration and launching of the devices and the measurement setup.

4.1. Scenario setup

When evaluating the performance of a network protocol, it is important to set realistic network conditions to be consistent with real scenarios. For this reason, an isolated network scenario was deployed to run the experiments on it. Fig. 8 illustrates the deployed network scenario for the analysis of the IoT protocols.

The devices that form the scenario and their purpose are depicted in Table 3. It is important to remark that the broker device is only used in MQTT scenarios as they need three devices for a single data exchange: publisher (server), subscriber (client) and broker.

A Raspberry Pi 3 was chosen to act as server because many times it is a device that is used to implement IoT (in home automation, for example). Even if we cannot consider it as a constrained device, it is a good approach. In particular, the Starter Kit Raspberry Pi 3 for Android Things was used (Fig. 9). It is important to remark that Android Things was not used during developing, it is just the denomination of the kit. This set includes a jacket that is attached to the Raspberry Pi and it includes buttons, sensors and different displays. For performance experiments, a remote temperature sensor is used.

To emulate a network like the ones in Figs. 1 and 4, we connect all the elements of our network scenario using the Ethernet switch. In order to emulate different network conditions, the program NetEm (Network Emulator) is used. This program allows the user to establish diverse network conditions such as packet corruption, duplication, reordering, delay or loss at the output of a network interface. It was necessary to set a virtual machine to emulate the network conditions as NetEm works in the Linux kernel and introduces the losses before sending the packet through the interface. Then, if NetEm had been used on the server (or on the client) the packets could not have been captured, it would have been as if they had never existed. Using a virtual host as network emulator and forcing the traffic to pass through it solves this problem: the packets can be captured at the client (or at the server) and lost packets can be detected.

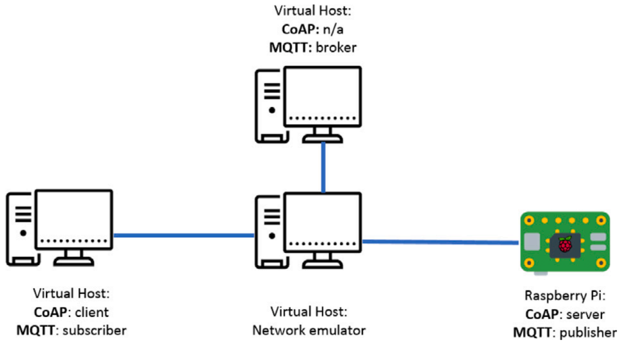

Virtual Host: CoAP: n/a MQTT: broker
Virtual Host: CoAP: client MQTT: subscriber
Virtual Host: Network emulator
Raspberry Pi: CoAP: server MQTT: publisher

Fig. 10. Logical network scenario for CoAP and MQTT analysis.

Taking this into account, the logical network scenario considers the network as isolated and forces the traffic between server, client and broker (in the case of MQTT) to pass through the network emulator. This logical scenario is shown in Fig. 10.

In addition, network configuration was necessary at the client to force the traffic sent to the server or to the broker to pass through the network emulator. This is achieved by adding two specific routes for the IP addresses of the server and the broker via the IP address of the network emulator. Analogous configuration was done for the server and for the broker. The configuration for the network emulator forces

Table 3
Description and purposes of the network scenario elements.

Element	Device	Purpose
Switch	Lynksys EZXS88W	To isolate the whole network scenario from the outside (Internet).
Server	Raspberry Pi 3 + Rainbow HAT	To read information from a sensor and to run the server application to send it.
Client	Virtual machine (LINUX)	To run the client application to get the information.
Network emulator	Virtual machine (LINUX)	To emulate the desired network conditions (different **packet loss** rates) with NetEm.
Broker (MQTT)	Virtual machine (LINUX)	To run the broker application and to act as intermediate between MQTT client and server. It is not used in CoAP scenarios.

the device to accept any traffic coming from the client or from the broker and any traffic sent to the client or to the broker and then route it. Same ciphersuites and security architectures were considered: PSK and PKI to check both the symmetric and asymmetric cryptography. Recommendations by [7] for TLS Profiles will be followed to meet the security requirements for CoAP and MQTT protocols.

With this scenario setup we can emulate losses such as those that can occur in a connection through the Internet between the different elements of the network (CoAP clients and servers, or publishers, subscribers and MQTT broker) and focus on studying the behaviour of the application layer protocols and security schemes.

It is important to note that in IoT environments, wireless interfaces such as ZigBee (IEEE.802.15.4), Bluetooth Low Energy (BLE), Z-Wave, Low-power Wi-Fi (IEEE 802.11ah), LoRaWAN or WiFi (IEEE 802.11) are frequently used. If the communication between the CoAP client and server, or between the MQTT publisher/subscriber and broker is a one hop communication using one of these wireless interfaces, then the network model that we propose would not be suitable. We should need to consider the details of the mechanisms of the wireless network (PHY and MAC layers) that would solve most of the losses due to transmission errors. We have decided in this work to focus on a wired environment (Ethernet) that better allows us to draw conclusions about the impact on the performance of the different application and security level protocols.

4.2. *TLS/DTLS decryption*

In order to evaluate the performance of the protocols, request and response packets must be correlated and matched to analyse parameters such as the latency of the communication or the total consumed bandwidth. So, Wireshark was used to capture the packets passing through the interface and obtain the parameters and statistics.

When securing CoAP using DTLS or MQTT using TLS, captured packets are encrypted and the correlation cannot be performed in an easy way. Decryption was needed to correctly match the exchanged messages. For this, session keys were obtained using the `libssl` library and provided to Wireshark.

4.3. *CoAP analysis*

Once the network scenario has been correctly configured, the analysis of CoAP protocol can be done. During this section, the possibilities offered by `libcoap` are analysed.

4.3.1. *Client and server implementation*

The libcoap implementation offers a complete and friendly use of the whole CoAP library. It includes two examples: `client.c` and `coap-server.c`,[11] which are simple implementations of a CoAP client and a CoAP server, respectively, that can be launched from a command terminal. These applications allow the user, amongst many other options, to specify through command line the use of confirmable or non-confirmable messages, an optional token for the CoAP header, the deployment or not of secure communications using DTLS and the CoAP URI of the resource in the server.

Moreover, the server offers two resources called `time` and `async` that allow to examine the differences between a piggybacked and a separate responses. The CoAP URI and the description of both resources is offered in Table 4. Thanks to these two resources and the possibility of specifying the use of non-confirmable messages, the three modes outlined in Fig. 3 can be simulated.

The `async` resource includes a delay between the sending of the ACK and the sending of the separate response. In order to make a fair comparisons of both response modes (piggybacked and separate), the code of the libcoap server was modified to set this delay to zero: the server uses a separate response when the client demands the async resource but it immediately sends it after the ACK with no delay.

The CoAP example server was modified to change the message sent when `time` or `async` resources are demanded. When a request for any of these two resources is received, the Raspberry Pi must sense the temperature, using the Rainbow Hat Jacket connected to it. Then, the CoAP server must receive this temperature value and attach it to the response. The programming interface of the Rainbow Hat is coded in Python, so a simple Python script for retrieving the temperature from the sensor is used. Then, it is sent to the CoAP response for the async and time resources.

With respect to the security support, these example applications do not allow to easily specify a particular cipher suite when using DTLS: they enable the use of the whole cipher suite list offered by DTLS 1.2. Then, the client sends a set of 49 possible cipher suites in the second `ClientHello` message of the DTLS handshake and the server chooses by default a certain predefined cipher suite. However, this behaviour can be corrected modifying the libcoap code to force the negotiation to result in a particular cipher suite. Following recommendations from [7], two cipher suites were included: one for PreShared Key (PSK), PSK-AES128-CCM8, and one for Public Key Infrastructure (PKI), ECDHE-ECDSA-AES128-CCM8. A PKI was deployed in ad-hoc mode to generate the certificates required.

- **PSK** (*Pre-Shared Key*): This scheme assumes that there exist a pre-shared secret between the two parties of the communication. This secret can be directly used as master secret to cipher the communication or it can be used to derive it.
- **AES128** (*Advanced Encryption Standard*): It is a block cipher scheme with a fixed block size of 128 bits used for encryption. The key size can be 128, 192 or 256 bits. In this case, 128 bits are used.
- **CCM8** (*Counter with CBC-MAC*): It is a mode of operation for block ciphers with fixed block sizes of 128 bits. It is used for hash purposes.

Finally, including the above-mentioned modifications, these examples allow to emulate the three scenarios depicted in Fig. 3, to obtain the temperature from the sensor, to secure the communication using DTLS and to specify the desired cipher suite.

[11] https://libcoap.net/doc/reference/4.2.0/

Table 4
Description of time and async resources in libcoap.

Resource	CoAP URI	Description
`time`	coap(s)://IPserver/time	The current time and date are sent using CoAP with a piggybacked response.
`async`	coap(s)://IPserver/async	A configurable message is sent with CoAP using a separate response.

4.3.2. Experiment cases

Once the implementation code has been modified to meet the requirements and the network scenario has been configured, the parameters to evaluate must be defined. In particular, there are three major aspects to evaluate with the run experiments:

- The different levels of reliability offered by the CoAP protocol represented by scenarios A, B and C (Fig. 3).
- The repercussion on the CPU usage and on the network performance of the distinct security modes: no security, PSK and PKI.
- The effect on the communication performance of diverse network conditions, defined by different packet loss rates.

To cover all the cases, nine experiments were run in the scenario shown in Fig. 10 for each value of packet loss rate: the three scenarios (A, B and C) in conjunction with the three security modes (NOSEC, PSK, PKI). Besides, five different packet loss rates were tested (0%, 5%, 10%, 15% and 20%), leading to 45 total experiments for CoAP evaluation. Each experiment consisted of the sending of 500 CoAP GET requests by the client and each experiment was monitored using Wireshark network analyser. The requests were run both serial mode and parallel mode, in order to facilitate the task of matching the requests and responses and automatically process them to obtain the latency of the communication.

Later on, the total bytes transmitted for every message and the time difference between the request and the response could be extracted from the Wireshark capture, correlating the messages using Matlab. This information will be used as an estimate of the bandwidth and the latency of the diverse modes of the protocol.

With respect to CPU usage analysis, PERF tool for Raspberry Pi OS was employed. This tool allows to monitor, amongst other parameters, the CPU cycles and instructions consumed by the 500 CoAP requests.

4.4. MQTT analysis

Following the same structure of the CoAP analysis, firstly, mosquitto's possibilities was analysed for broker, publisher and subscriber implementation. Then, the security support of the protocol and the cases to simulate are defined.

4.4.1. Broker, publisher and subscriber implementation

Mosquitto implementation offers a complete MQTT broker that can be configured through a configuration file.[12] This configuration file allows to define the listening ports, the TLS configuration (certificates, version, ciphers, etc.), the logging level and other parameters such as the maximum QoS level or the maximum number of simultaneous connections. The TLS configuration was mainly modified.

In order to analyse the impact on the communication of the distinct levels of QoS and to send the temperature measured by the Raspberry sensor, the clients (both publisher and subscriber) must be flexible enough to adapt themselves to these circumstances. Even though mosquitto is a broker implementation, it offers an additional package named `mosquitto-clients`. This package includes two tools named `mosquitto_pub` and `mosquitto_sub`, that implements a simple publisher and a simple subscriber respectively. The `mosquitto` broker and the `mosquitto_sub` tools remain active until the process is killed. By contrast, the `mosquitto_pub` just sends a single publish and it ends its execution.

[12] https://mosquitto.org/man/mosquitto-conf-5.html

As in the case of CoAP, different ciphering and authentication schemes for MQTT protocol are evaluated following guidance of RFC7925 [7]. Once again, PSK and PKI architectures are tested, comparing the bandwidth and the CPU usage associated to these two different security approaches.

The proposed cipher suites do not depend on whether TLS or its datagram version (DTLS) is used. Then, the same cipher suites evaluated in CoAP over DTLS are tested in MQTT over TLS:

- **PSK-AES128-CCM8** for PSK architecture, assuming that both parties share a secret. This can be achieved in mosquitto by launching the publisher/subscriber specifying a certain secret and the identity of the client through the command line. In the case of the broker, it has a file containing all the pre-shared secrets and the identity of the clients.
- **ECDHE-ECDSA-AES128-CCM8** for PKI, using *secp256r1* as elliptic curve. In the case of MQTT, the broker will be authenticated by a certificate signed by a trusted Certification Authority (CA), so the clients are able to verify its identity. It is possible to also generate certificates for the clients but it is not usual as it increases the occupied bandwidth and the complexity of the clients. We create our own PKI for this purpose.

4.4.2. Experiment cases

Analogous to the experiment cases of the CoAP protocol, there also exist three major aspects to evaluate with the MQTT experiments:

- The effect of the different levels of reliability, in the case of MQTT represented by the QoS. MQTT standard defines three levels (QoS 0, QoS 1 and QoS 2) represented in Fig. 7.
- The repercussion on the devices CPU usage and on the network performance of the use of TLS over MQTT with no security, PSK and PKI modes.
- The effect on the MQTT communication performance of diverse network conditions, defined by different packet loss rates.

Following CoAP methodology, nine experiments were run in the scenario shown in Fig. 10 for each value of packet loss rate: the three levels of QoS (0, 1 and 2) in conjunction with the three security modes (NOSEC, PSK, PKI). These nine experiments were repeated for five different packet loss rates (0%, 5%, 10%, 15% and 20%), resulting in 45 total experiments. Each experiment consisted of the publishing of 500 MQTT messages by the publisher, monitoring them using Wireshark network analyser.

With respect to CPU usage analysis, as in the case of CoAP protocol PERF tool was employed. In this case, the process to monitor was the MQTT application located on the publisher's side (Raspberry Pi).

Using the 45 above-mentioned experiments, the bandwidth and CPU usage analysis could be carried out but in order to analyse the latency, the scenario shown in Fig. 10 needed to be modified. In that scenario there was no way but external clock synchronization to analyse the time delay between the transmission of the `PUBLISH` message by the publisher and the reception by the subscriber of the `PUBLISH` message sent by the broker. Following [12], the publisher and subscriber were run in the same virtual machine to avoid clock synchronization. Then, the `PUBLISH` message sent by the publisher reaches the broker going through the network emulator and then returns back to the same virtual machine to reach the subscriber. The scenario to measure the latency is shown in Fig. 11

Then, a new set of 45 experiments were run in the new scenario to analyse the latency offered by MQTT protocol, leading to a total of 90 experiments.

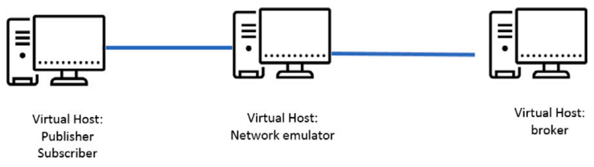


Fig. 11. Logical network scenario for MQTT latency analysis.

Table 5
Length of different CoAP messages.

Message	App layer (bytes)	On wire (bytes)
CON (request)	9	62
ACK	4	62
ACK + answer	28	72
CON (answer)	28	72
NON (request)	9	62
NON (answer)	28	72

5. Analytical performance study

In this section we are going to obtain analytically some performance metrics of the CoAP and MQTT protocols under certain circumstances which we can then contrast in Section 6 with the results obtained in the experiments.

5.1. Analytical study of the CoAP protocol

The proposed scenario and the CoAP protocol are simple enough to try to approach them by an analytical model and subsequently contrasting the results with measurements in a real experiment.

5.1.1. No encryption and no loss

We are going to first estimate how many bytes are necessary to obtain information from a sensor using unencrypted CoAP and assuming a lossless network, depending on whether we are in scenario A, B or C in Fig. 3.

Initially we will assume a network without losses. CoAP messages have a defined format, which we find in Fig. 2. CoAP messages have a fixed 4 bytes header, plus other optional fields that may vary depending on the type of message or implementation. Specifically, the *Token* field can be between 0 and 8 bytes long, the *Options* field length depends on the length of the URI or the headers included in the request or response (most of them optional, as in HTTP), and the *Payload* length depends of the data being returned by the server. The CoAP implementation in our experiment uses a *Token* length of 0 bytes, it includes the URI option in the request (5 bytes) and the `ETag`, `Content-Format` and `Max-Age` headers in the responses (9 bytes). The data returned by the server is 15 bytes long. The *Token* field length and the options included in the message may change in other CoAP implementations. With these values, the size of the various CoAP messages at the application layer is summarized in Table 5. To these lengths we must add the UDP header (8 bytes), IP header (20 bytes in IPv4 and 40 bytes in IPv6; in the experiment we have used IPv4), plus the link layer header, tail and padding (IEEE 802.3 in our experiment). Taking these overloads into account, the total length of the messages (linux cooked-mode capture, SLL) is also shown in Table 5.

Taking into account the exchange of messages shown in Fig. 3, the total number of packets and bytes exchanged to access the information in the server when no encryption is being used and there are no losses in the network is shown in Table 6.

Regarding the delay, it is approximately the same in all three cases, and depends on the time it takes for the server to obtain the sensor reading (*T_sensing*), and on the round trip time (*RTT*) delay in the network. In our experiment we use a Raspberry Pi that gets the value from a temperature sensor. The time it takes to obtain this reading is approximately 720 ms, so this delay dominates the others.

Table 6
Number of CoAP packets and bytes exchanged.

Scenario	Packets	Bytes
A	2	134
B	4	258
C	2	134

5.1.2. Effect of encryption

As we have commented before, to send messages securely, CoAP uses the DTLS 1.2 protocol. This involves, on the one hand, the exchange of initial messages to establish the secure connection (handshake) and, on the other, the encryption of the data that is exchanged (CoAP request and response messages).

Regarding the DTLS handshake, it involves the exchange of messages presented in Table 7, which are basically the same for PSK and PKI, except for the *Certificate* message that is sent in the case of PKI and not in the case of PSK (the message size shown in the table corresponds to the on wire length of the message, including link, network and transport layer headers).

Following this exchange of messages, the CoAP application-level messages are sent encrypted. Since AES128-CCM8 is used in both cases, the messages occupy the same regardless of whether PSK or PKI is used (Table 8).

Finally two *close_notify alert* messages are sent to indicate that the secure session is ending (Table 9).

Therefore, the number of packets and bytes sent in each of the CoAP scenarios are as shown in Table 10.

Regarding the delay in obtaining the response, it has three components. First, the transmission of messages between client and server. Second, the time it takes for the server (the Raspberry Pi) to obtain the sensor reading. Finally the time it takes to encrypt the request and the response. Calling *RTT* the round trip time, *T_sensing* the time it takes to get the sensor reading, and assuming that the encryption/decryption time of the request and response messages is *T_encrypt* and *T_decrypt*, the delay from the start of the handshake until the client obtains the reading from the sensor, ignoring the transmission time of the messages, is in the three scenarios *4 RTT + 2 T_encrypt + 2 T_decrypt + T_sensing*. Of all these times, the one that dominates in the case of our experiment is *T_sensing*, 720 ms, which makes the delay in the three scenarios quite similar.

5.1.3. Effect of losses

The effect of losses on the performance metrics is more complex to deal with analytically. We will now consider how network losses affect the delay in getting the response for the case where no encryption is being used.

As CoAP uses the UDP transport protocol, the loss of messages must be detected and corrected (retransmitted) at the application layer. To do this, the CoAP protocol implements in the confirmed mode (scenarios A and B) an *ACK_TIMEOUT* retransmission timer with a default value of 2 s. If after transmitting a CON message this timer expires without receiving ACK (either because the CON message has been lost or because the ACK has been lost), the CON message is retransmitted with a certain randomness factor *ACK_RANDOM_FACTOR*, to avoid repeated collisions.

Calling p the probability that a packet is lost due to a transmission error or network congestion, and $q = 1 - p$ the probability that it arrives successfully, calling again T_sensing the time it takes for the server to obtain the sensor reading and RTT the round trip time in the network, the average delay in getting a response when using a confirmed (CON) message will be:

$$\begin{aligned} delay = {} & T\_sensing + q^2 \times RTT + (1-q^2) \times q^2 \\ & \times (RTT + ACK\_TIMEOUT) + \\ & + (1-q^2)^2 \times q^2 \times (2RTT + ACK\_TIMEOUT) + \\ & + (1-q^2)^3 \times q^2 \times (3RTT + ACK\_TIMEOUT) + \cdots \end{aligned} \tag{1}$$

Table 7
DTLS handshake messages length.

	Message	PSK (bytes)	PKI) (bytes)
⟶	Client Hello	297	203
⟵	Hello Verify Request	104	104
⟶	Client Hello with cookie	329	235
⟵	Server Hello, (Certificate), Server Key Exchange, Server Hello Done	214	953
⟶	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message	174	169
⟵	New Session Ticket, Change Cipher Spec, Encrypted Handshake Message	318	302
	TOTAL	1,436	1,966

Table 8
CoAP over DTLS messages length.

Message	Bytes
CON (request)	82
ACK	77
ACK + answer	101
CON (answer)	101
NON (request)	82
NON (answer)	101

Table 9
DTSL final messages length.

	Message	PSK (bytes)	PKI) (bytes)
⟶	Encrypted Alert (close_notify)	75	75
⟵	Encrypted Alert (close_notify)	75	75

Table 10
Total number of CoAP over DTLS packets and bytes.

Scenario	Packets	PSK (bytes)	PKI (bytes)
A	10	1,769	2,299
B	12	1,854	2,453
C	10	1,769	2,299

Table 11
CoAP delay against probability of packet loss.

p	Delay (ms)
0	720
0,05	936
0,1	1,189
0,15	1,488
0,2	1,845

Neglecting *RTT* against *T_sensing* and *ACK_TIMEOUT* and assuming that this sum has infinite terms (in fact, the maximum number of retransmissions is limited) we have:

$$delay = T\_sensing + q^2 \times ACK\_TIMEOUT \times \sum_{n=1}^{\infty} n(1-q^2)^n \tag{2}$$

Taking into account that $\sum_{n=1}^{\infty} nx^n = \frac{x}{(1-x)^2}$ we obtain:

$$delay = T\_sensing + ACK\_TIMEOUT \times \frac{1-q^2}{q^2} \tag{3}$$

For $T\_sensing = 720$ ms and $ACK\_TIMEOUT = 2000$ ms, we show in Table 11 the value of the delay for different values of probability of packet loss p.

This analytical study is difficult to extend to the case when using a security mechanism with CoAP. In this case, the losses are recovered in different ways depending on whether they affect one of the 6 packets that contain DTLS handshake messages, or if they affect one of the 2 or 4 messages (depending on the scenario) that carry CoAP protocol messages. In the first case, DTLS implements a retransmission timer of 1 s, which doubles in case of repeated loss of the message. In the second case, the CoAP recovery mechanism that we discussed earlier comes into play.

Table 12
MQTT messages length.

Message	MQTT (bytes)	Total (bytes)
CONNECT	14 + Client_ID	79 + Client_ID
CONNACK	4	70
PUBLISH	4 + Topic_Length + Data_Length	70 + Topic_Length + Data_Length
PUBACK	4	70
PUBRECEIVED	4	70
PUBRELEASE	4	70
PUBCOMPLETE	4	70

5.2. Analytical model of the MQTT protocol

The MQTT case is more difficult to model analytically when using TCP as the transport protocol. TCP manages a series of timers, such as the delayed ACKs or the constant estimation of the retransmission timeout (RTO), which means that the number of transmitted messages is not always the same (depending on whether the delayed ACKS are sent or piggybacked in a data segment).

We will now make an estimate of the number of exchanged packets and the size of the messages for the different qualities of service supported by MQTT, for the case in which we do not use encryption and there are no losses.

Table 12 shows the size of the MQTT messages that we are going to observe during the publication of a reading in a broker and while it is sent to a subscriber. Some of the messages have a variable length, which depends on the length of the client identifier (*Client_ID*), the topic name (*Topic_Length*), and the size of the published data (*Data_Length*). Note that in this table we show the size of the messages both at the application layer (MQTT) and once transmitted over the network, assuming that an IEEE 802.3, IPv4 and TCP transport is used (the result will vary depending on the options implemented by the TCP stack, we are considering a TCP stack that implements the timestamp option).

In our work we assume that publishers when they want to publish a new value in the broker, open a TCP connection with the broker, connect using a client ID, publish the data in a certain topic with a quality of service (QoS 0, QoS 1 or QoS 2), and close the connection. We will also assume that there is a single subscriber. In this case, the total number of packets theoretically required to make this publication is shown in Table 13.

For this calculation we have taken into account the TCP segments that do not contain any data (such as SYN and FIN), as well as the acknowledgements (ACK), which we have assumed are not delayed. In practice, the delayed ACKs that are sent together with another data segment imply that the number of packets sent can be reduced up to the number that we also indicate in the table.

Regarding the delay until a subscriber receives a reading from the publisher, assuming that the propagation time from the publisher and

Table 13
Number of MQTT packets involved when publishing a new value.

QoS level	Number packets (without delayed ACKs)	Number of packets (with delayed ACKs)
QoS 0	17	14
QoS 1	21	16
QoS 2	29	21

the subscriber to the broker is approximately the same, and taking into account that the TCP connection establishment plus the connection to the broker (CONNECT and CONNACK) takes 2 RTTs, from Fig. 7 we see that the delay with QoS 0 and QoS 1 is 3 RTT, while with QoS 2 is 4 RTT.

We must take into account that in this case we are not considering the delay in obtaining the temperature reading from the sensor (*T_sensing*), since we consider that this is done immediately before the sensor connects to the broker to publish the obtained value.

Due to the greater complexity of TCP mechanisms, we do not address for MQTT the analytical study of the number of messages and delay with encryption or loss in the network. We will study these cases through our experiment in the next section.

6. Experimental results

This section contains the results of the processing of the measurements obtained from the experiments, focusing on bandwidth efficiency, CPU usage and latency of the communication. First, the results for CoAP protocol and MQTT protocols will be shown independently and, subsequently, the comparison between them.

The performance of the protocols has been evaluated using the previously explained experiments according to three indicators:

- The total amount of bytes transferred per message exchange as an indicator of the **bandwidth usage**.
- The average number of *CPU cycles* and the *amount of packets* employed per message exchange. *CPU cycles* are an important indicator of the power consumption as a higher number of CPU cycles directly implies a higher power consumption and a bigger need of processing capabilities [26]. *Number of packets transferred* is also important, because the network interface power consumption has a fixed part per transmitted packet and a variable one that depends on the number of bytes of the packet.
- The time delay between the requests and the responses as an indicator of the **latency** of the communication.

In the figures, we show the average values over the experiments, together with the confidence intervals with a 95% confidence level.

6.1. CoAP results

6.1.1. CoAP bandwidth

Fig. 12 shows the results in terms of bandwidth for the scenarios A, B and C. The results coincide with those obtained analytically. It is proved that ciphering methods drastically increase the bandwidth use in all the scenarios:

- The results in the scenarios A and C are practically identical. In these, the amount of bytes transferred with a 0% loss rate increases from 134 bytes when no securing the communication to 1769 bytes when securing the communication in PSK mode and 2299 when using PKI mode. Both PSK and PKI modes add the overhead of the DTLS negotiation but PKI mode is more greedy in terms of bandwidth use as it includes the server's certificate exchange.
- In the Scenario B, with a 0% loss rate the bytes transferred increases from 258 bytes in NOSEC mode to 1854 and 2453 bytes in PSK and PKI modes respectively. There exists a minor difference with respect to Scenario A caused by a bigger number of CoAP packets needed for a single transfer (from 2 in scenario A to 4 in scenario B). However, in PSK and PKI modes this increase of number of packets becomes insignificant when comparing modes A and B.

In both scenarios, as the packet loss rate increases, the number of bytes per message that are transmitted in PSK and PKI grows up significantly as well, because of the needed packets per transmission is greater and also their length.

6.1.2. CoAP CPU usage

Figs. 13 and 14 show the evolution of the CPU usage with the different ciphering modes under different values of packet loss rates.

The results in terms of CPU cycles (Fig. 13) prove that ciphering significantly increases the CPU usage, being PKI mode the most demanding one. Also, the no-reliability offered by Scenario C means a decrease of the CPU cycles as the loss rate increases. The reason of this effect is that some requests do not even reach the server and they are not retransmitted by the client, so less requests are processed on the server and this is translated into a lower CPU usage. Although this reduction is achieved at the expense of losing some requests and worsening the overall performance, some system would prefer this mode when messages are not so important because they are periodically repeated.

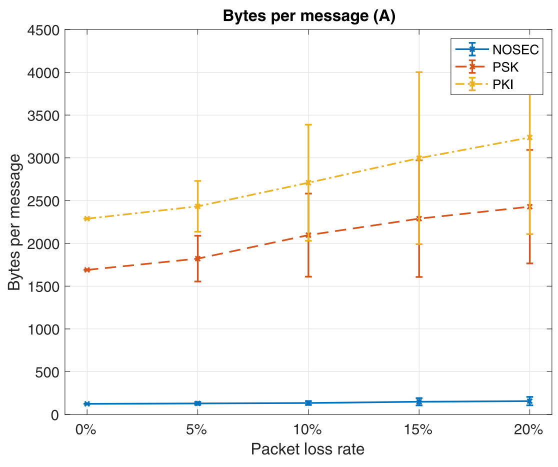
In addition to these conclusions, it can be observed that mode A consumes more power than mode B and much more than mode C. This difference with mode B is due to the need of an immediate response that requires an additional effort on the server side. On the other hand, the difference with mode C is due to the lack of reliability of the latter one, avoiding the use of retransmission timers and the processing of the lost requests.

Attending to the number of packets (Fig. 14), it can be observed that the number of packets when comparing to NOSEC is significantly increased by PSK and PKI modes but, in this case, in a similar fashion (both follows the DTLS message exchange pattern). This increase of the number of transmitted packets will increase the power consumption of the network interface and then, it will also have an impact on the overall power consumption.

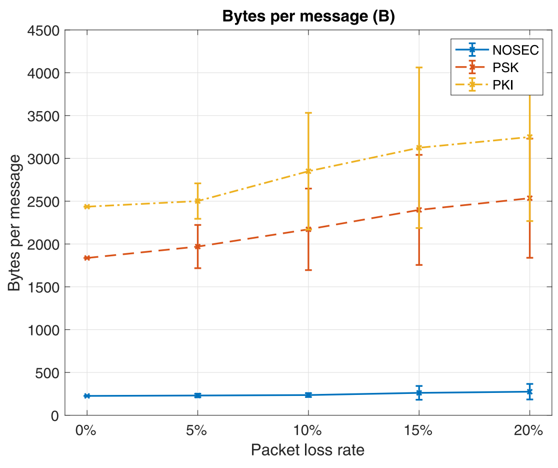
6.1.3. CoAP latency

The time delay between the first and the last packets exchanged for each transfer were measured during the experiments to evaluate the latency offered. It is important to remark that these time delays includes the time required by the Raspberry Pi to measure the temperature ($\approx$ 720 ms). The results for the three scenarios under different packet loss rates using the three proposed ciphering modes are shown in Fig. 15. This results agree with those obtained analytically for NOSEC in Table 11. The following conclusions can be drawn:

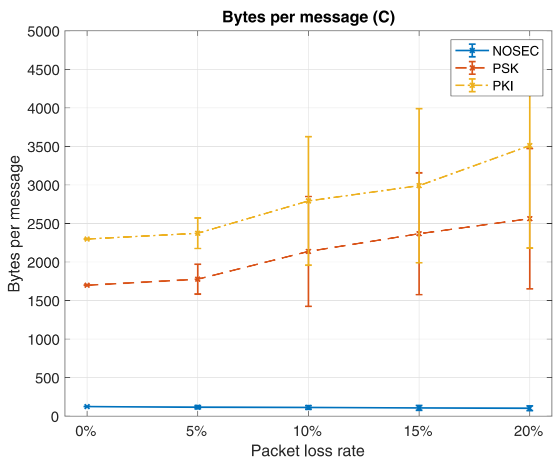
- In lossless networks, the effect of the ciphering modes on the latency is almost despicable in the three scenarios. For example, the average time delays in a lossless scenario A are 720.04 ms, 723.1876 ms and 728.3494 ms for NOSEC, PSK and PKI modes, respectively.
- With higher loss rates secured communications are more adversely affected than no-secured ones, observing a similar fashion for PSK and PKI modes. This is due to the retransmission timers of DTLS protocol and the exchange of a higher number of packets. Thus, there is higher probability of having at least 1 packet lost in a single message exchange when CoAP is run over DTLS.



(a) Bytes per message (A)



(b) Bytes per message (B)



(c) Bytes per message (C)

Fig. 12. Bandwidth usage by scenario (CoAP).

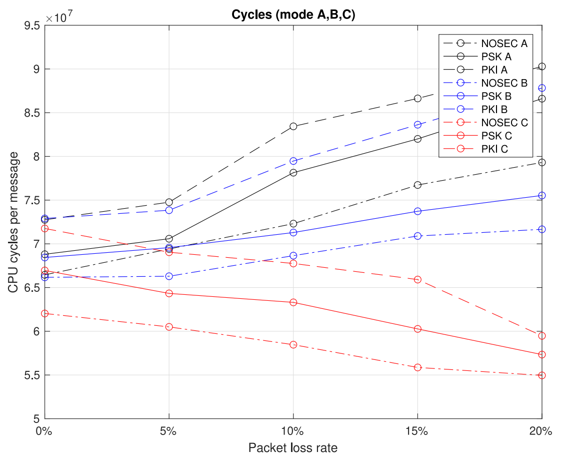


Fig. 13. CoAP CPU usage.

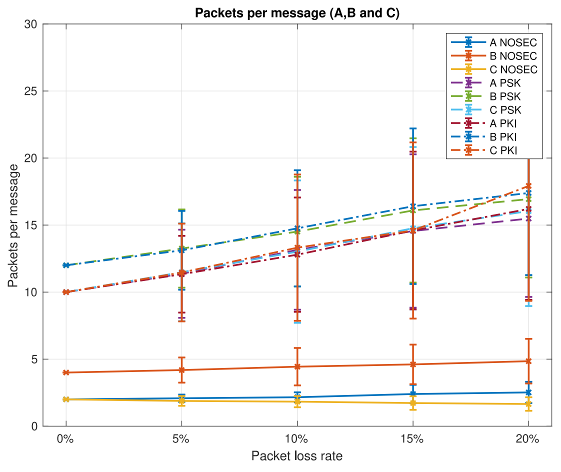


Fig. 14. CoAP number of packets transferred per message.

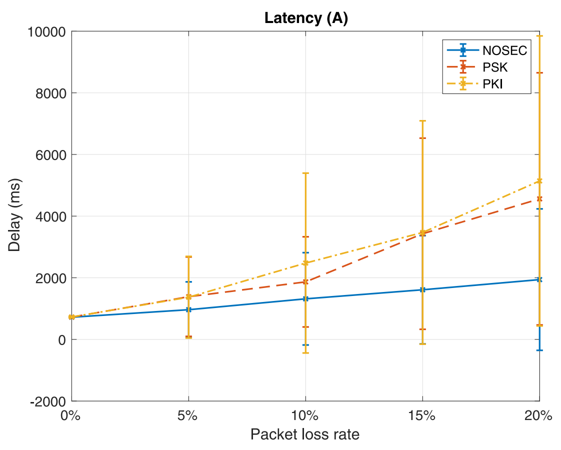
- In the Scenario C, the loss of a CoAP application packet means an infinite wait for a response. For this reason, only complete transfers are considered and the packet loss rate has no effect on the latency of the NOSEC mode and the effect on PSK and PKI modes is lower than the observed effect in scenarios A and B.

Summarizing, effect of ciphering is negligible ($\approx$ 1%) in lossless network, but it is more adversely affected by losses.

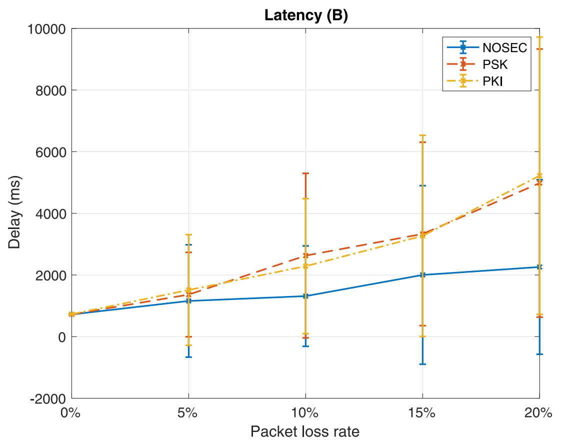
In addition to these graphics representing the average time delay, the cumulative distribution functions of the time delays were obtained for the different packet loss rates. They are shown in Fig. 16.

Considering these cumulative distribution functions, similar conclusions can be stated:

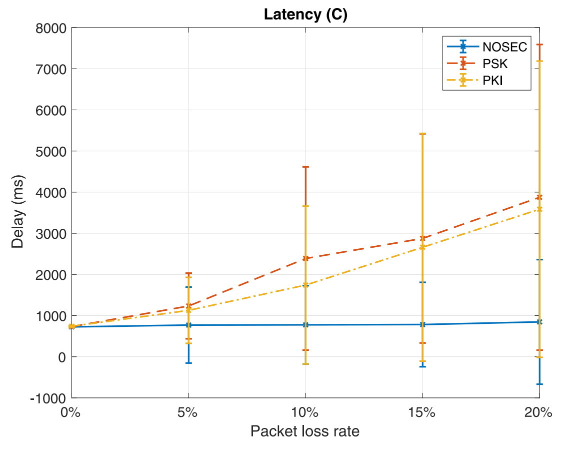
- In lossless networks (0% loss) the distribution of the time delays are almost identical, being the PKI ciphering mode the one with highest delays followed by PSK and finally NOSEC modes.
- In networks with loss rates greater than 0, the non-secure communications using scenario C are not affected in terms of time delay since a loss does not mean a delay but the end of the transfer.
- The non-secured scenarios A and B behave similarly as the loss rate increases. However, the time delay of scenario B is a little higher that the one of scenario A due to a higher number of packets per message.
- PSK and PKI ciphering modes in scenarios A and B show a similar fashion as the loss rate increases, being more affected by the losses than the non-secured scenarios.
- When securing the scenario C either with PSK or PKI, there exist also cases in which the CoAP request is lost and the data transfer



(a) Time delay (Scenario A)



(b) Time delay (Scenario B)



(c) Time delay (Scenario C)

Fig. 15. Time delay per message by scenario (CoAP).

is not completed. For this reason, the PSK and PKI cumulative distribution function of scenario C may seem better than the ones of scenarios A and B but it is important to remark that this is achieved at the expense of losing some messages.

6.2. MQTT results

The performance measurements of the MQTT protocol were done analogously to the performance measurements of CoAP.

6.2.1. MQTT bandwidth

Fig. 17 shows the bandwidth usage for the different values of quality of service (QoS 0, QoS 1 and QoS 2), measured through the amount of bytes transmitted per message:

- As it may be expected, the bytes transmitted per message are higher for higher quality of service values since more packets are needed for every exchange. In lossless networks (0% loss rate), 1057 bytes are exchanged for QoS 0, 1204 for QoS 1 and 1560 for QoS 2.
- As in the case of CoAP, it can be observed that securing the MQTT communications significantly increases the bandwidth use in all cases. With a 0% loss rate the bytes transferred increase from 1057, 1204 and 1560 (for QoS 0, QoS 1 and QoS 2 respectively) to 2009, 2280 and 2713 with PSK ciphering and to 3978, 4281 and 4685 for PKI ciphering. Analogous to the DTLS support in CoAP, PSK and PKI modes increase the bandwidth usage by the introduction of the TLS handshake. Moreover, PKI ciphering mode includes the exchange of the server's certificate, becoming again the more demanding mode in terms of bandwidth usage.
- With respect to the effect of the different packet loss rates, the amount of bytes transferred grows up as the loss rate increases, showing NOSEC, PSK and PKI similar slopes.

6.2.2. CPU usage

Figs. 18 and 19 show the evolution of these two parameters for the different quality of services values, packet loss rates and ciphering modes.

Focusing on Fig. 18, it can be stated, as in the case of CoAP, that securing MQTT communications means a major increase of the consumed power, being PSK CPU usage more moderate and the PKI more intense. However, no difference seems to exist in terms of CPU usage between the different qualities of service and a minor difference exist between the different loss rate values.

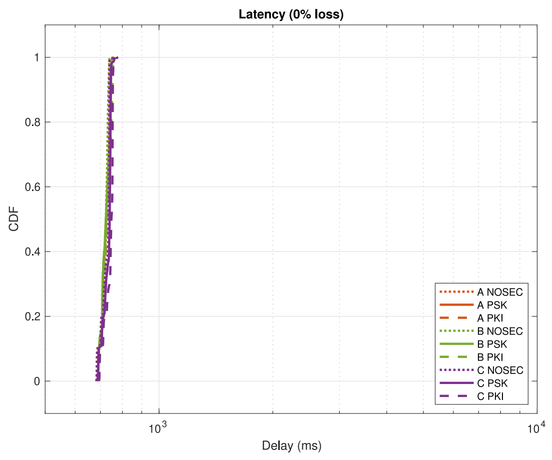
This could be explained through two reasons:

- CPU usage by TCP may not being monitored by PERF tool. Although this tool counts the CPU cycles both at user and kernel levels, the minor effect of the increase of the packet loss rate on the CPU usage may be indicating that TCP processing is not being considered.
- Mosquitto may be a very efficient implementation of MQTT, reducing the CPU charge of the distinct quality of service modes.

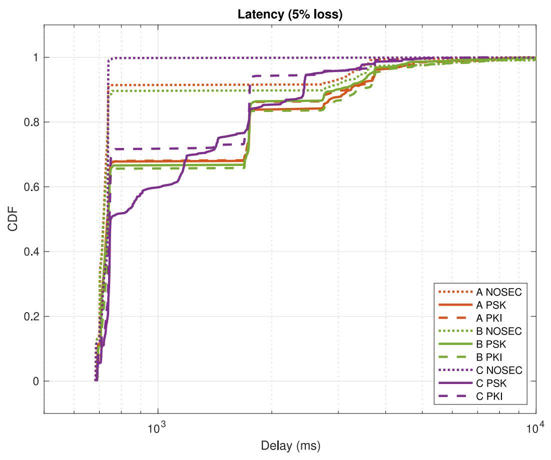
With respect to the number of packets transmitted, more coherent results are obtained. In this case, the number of packets clearly increases as the packet loss rate grows up and there exists an important different in the number of packets among the diverse quality of service modes: in lossless networks QoS 2 requires 21, 26 and 27 packets in average with NOSEC, PSK and PKI modes respectively, QoS 1 requires 16, 21 and 23 packets for the same modes and QoS 0 14, 18 and 19 packets. This values agree with those obtained for NOSEC in Section 5.

6.2.3. Latency

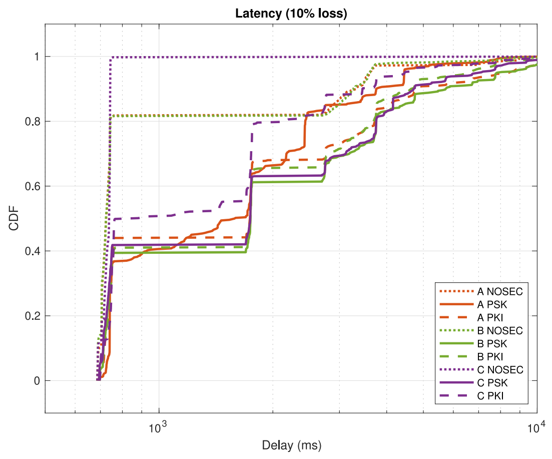
After deploying the scenario shown in Fig. 11, the time delay between the transmission and the reception was measured for all the 45 experiments. There exist a very important difference with respect to CoAP latency analysis: the receiver does not have to wait for the Raspberry Pi to measure the temperature. When available, the Raspberry Pi initiates the communication, contrary to what happens in CoAP



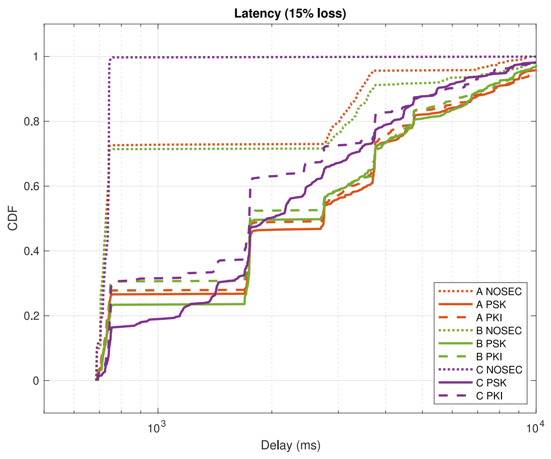
(a) 0% packet loss rate



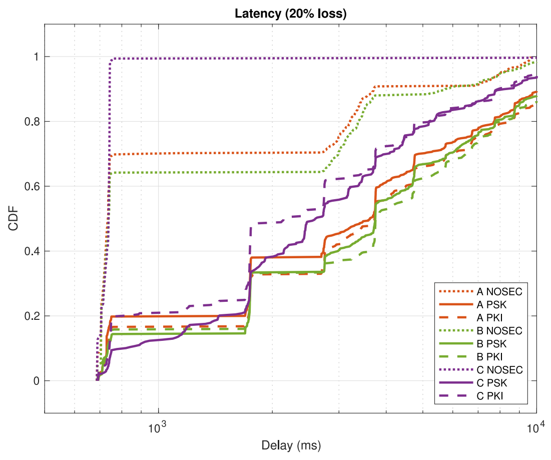
(b) 5% packet loss rate



(c) 10% packet loss rate



(d) 15% packet loss rate



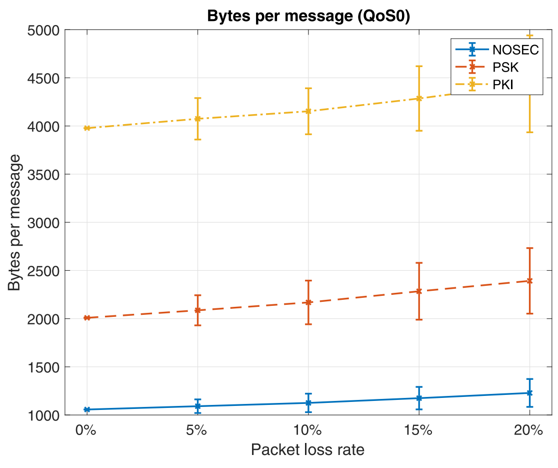
(e) 20% packet loss rate

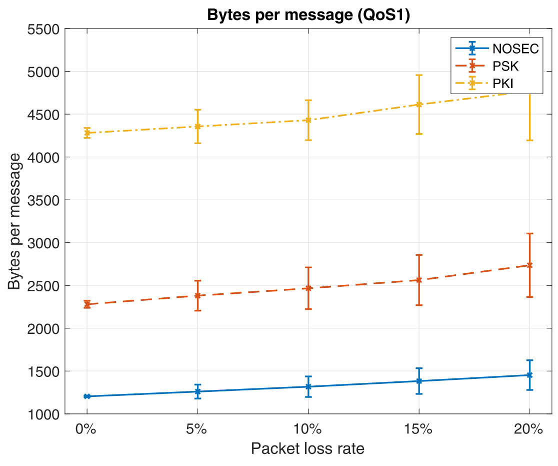
Fig. 16. Cumulative distribution function of time delays per packet loss rate (CoAP).

protocol. For this reason, the 720 ms needed by the Raspberry Pi to measure the temperature are not included in the latency of the protocol.

Fig. 20 shows the latency estimates for the three quality of service values under different packet loss rates and the distinct ciphering modes. It can be stated that:

- As in the case of CoAP, the effect of ciphering on the latency does not make a substantial difference in lossless networks for the three quality of service values. For example, with QoS 2 the latency values are 10.2 ms for non-secure communications, 21.4 ms for PSK ciphered communications and 21.9 ms for PKI ones.


(a) Bytes per message (QoS 0)


(b) Bytes per message (QoS 1)

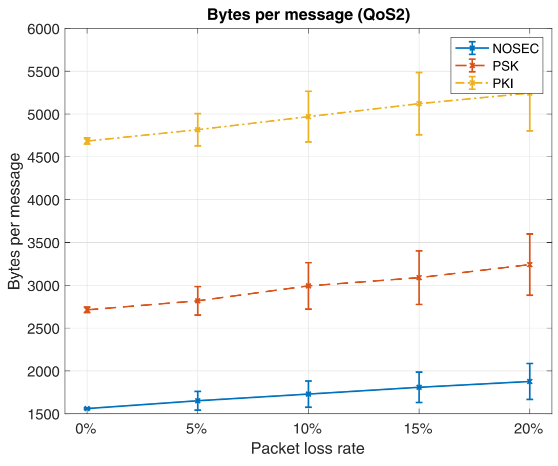

(c) Bytes per message (QoS 2)

Fig. 17. Bandwidth usage by quality of service (MQTT).

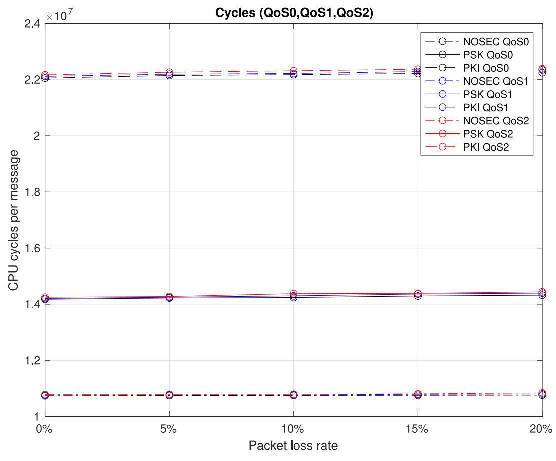


Fig. 18. MQTT CPU usage.

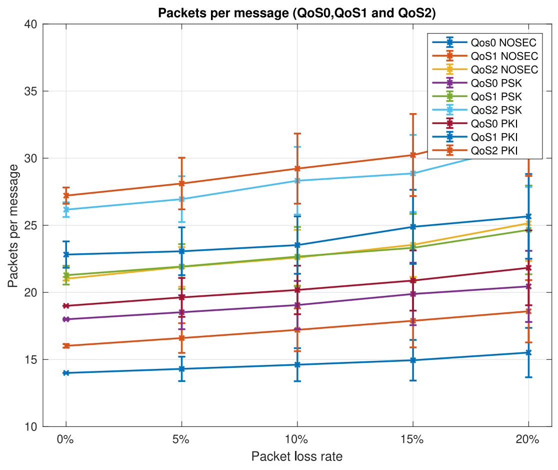


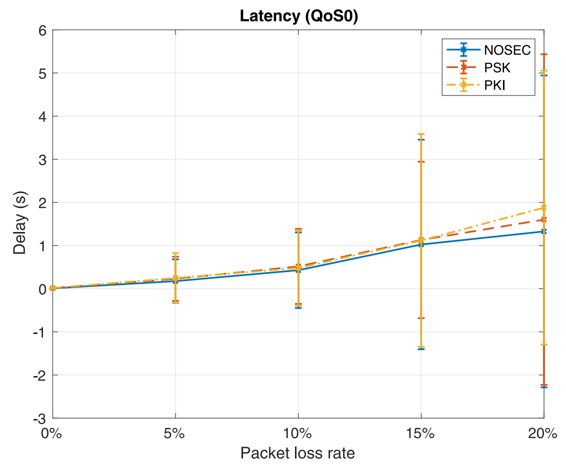
Fig. 19. MQTT number of packets transferred per message.

- As expected, a higher value of quality of service means a higher value of latency for the communication. However, it is not observed a substantial difference between the modes in terms of latency, as it does exist in terms of bandwidth and CPU. For example, for non-secured lossless networks, the latency values are 8.6 ms for QoS 0, 9.5 ms for QoS 1 and 10.2 ms for QoS 2. As the packet loss rate increases this difference becomes greater but it is still not as pronounced as the existing one in terms of bandwidth and CPU.

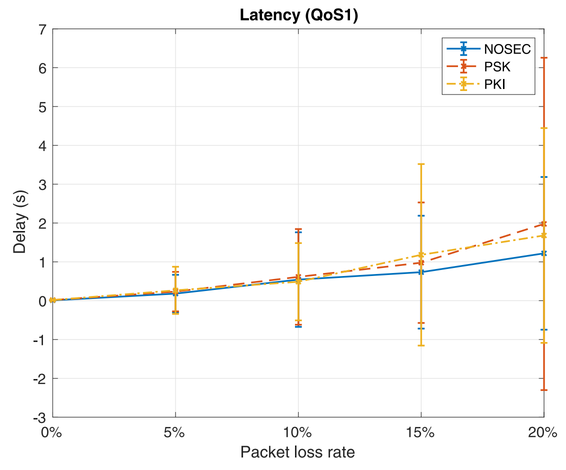
As done during CoAP analysis, cumulative distribution functions of the latency for the diverse packet loss rates are shown in Fig. 21.

Considering these cumulative distribution functions, similar conclusions can be stated:

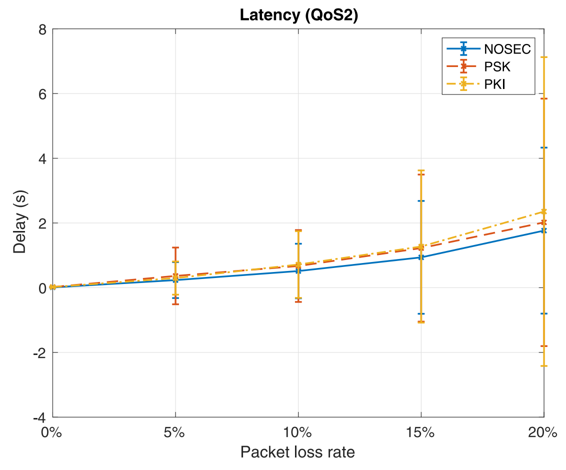
- In lossless networks (0% loss) the time delays of all the ciphering and quality of services modes follow similar distribution curves. The PKI and PSK modes for the highest quality of service (QoS 2) produce the highest delays followed by the PKI modes for the other two values of quality of service (QoS 1 and QoS 0). Finally, the PSK modes for QoS 1 and QoS 0 and the three NOSEC ones complete the graph.
- PSK and PKI ciphering modes follow similar cumulative distribution functions as the loss rate increases, being more adversely affected by the losses than the non-secured scenarios. However, PKI ciphering produces a higher delay due to the need of exchanging the server's certificate and the transmission of a higher number of packets.
- A higher value of quality of service means a higher time delay due to the need of exchanging a higher number of packets. For this reason, in the five graphs for a certain ciphering mode, the



(a) Time delay (QoS 0)



(b) Time delay (QoS 1)



(c) Time delay (QoS 2)

Fig. 20. Time delay per message by scenario (MQTT).

curves of QoS 2 are always more shifted to the right that the corresponding QoS 1 ones, and these in turn are more to the right than the corresponding QoS 0 ones. As stated before, the time difference between the modes does exist but it is not as critical as the difference in terms of bandwidth usage or CPU. It is not even comparable to the time difference between the secured and the non-secured scenarios, being this one a more crucial issue.

6.3. CoAP and MQTT comparison

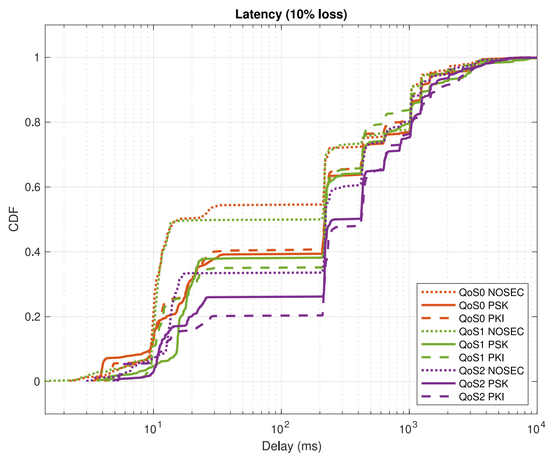
Attending to the results shown in the two previous sections, a comparison between CoAP and MQTT is offered in terms of bandwidth usage, CPU and latency:

- **Bandwidth**. The bytes per message exchange needed by both protocols are represented in Fig. 22 classified attending to the ciphering mode under test (NOSEC, PSK or PKI). As it can be observed, the use of MQTT supposes an important increase of the bandwidth used due to the deployment over TCP instead of over UDP for all the ciphering modes. In lossless non-secured networks the difference is substantial: the most reliable CoAP scenario (B) employs just 226 bytes in average while the MQTT scenario with the highest quality of services employs 1560 bytes.

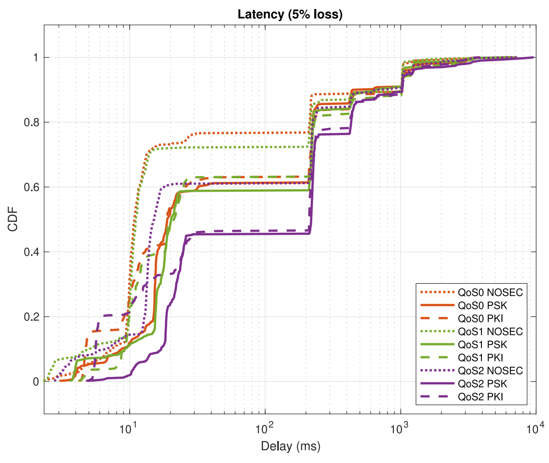
  When securing the communications, CoAP adds the overhead of the DTLS handshake and MQTT adds the one introduced by the TLS handshake. Even under this equal conditions, TLS still uses TCP and DTLS uses UDP instead, making MQTT again heavier than CoAP. Just in the case of PSK ciphering mode, the graphs of CoAP and MQTT cross each other: CoAP can be heavier than the MQTT scenario with the lowest quality of service for high packet loss rates. The reason why this crossing does not happen for PKI mode is that, due to specific needs of the implementations, the server's certificate needed by mosquitto is heavier than the one needed by libcoap.

  In conclusion, MQTT is more demanding than CoAP in terms of bandwidth use due to the employment of TCP as transport protocol. On the other hand, the usage of this protocol makes the MQTT messages transmission reliable, what is not achieved by CoAP just using UDP: CoAP protocol has to add and handle its own reliability mechanism (modes A, B and C). However, this substantial difference with respect to the bandwidth use can constrain the usability of MQTT on very limited networks and devices.
- **CPU usage.** The packets transferred per message (PPM) exchanged are shown in Fig. 23 classified according to the ciphering mode. Assuming that the network interface power consumption has a fixed part per packet, a higher number of packets may mean a higher power consumption in combination with a higher number of total bytes transferred. Attending to this criterion, MQTT turns to be also more demanding in terms of power consumption: the use of TCP requires an additional overhead to establish, control and finish the connections. This remains true for all the ciphering modes as it can be clearly seen. Moreover, the difference in terms of packets number is high: in lossless non-secured networks most reliable CoAP mode (B) needs only 4 packets, while MQTT highest quality of service mode needs 21 packets.

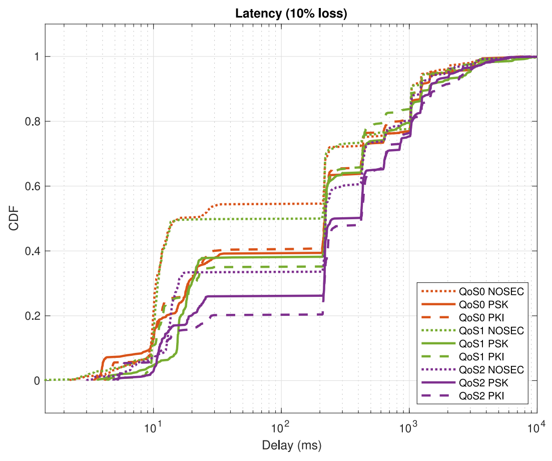
  In conclusion, the use of MQTT in constrained devices could be limited due to a higher power consumption that is linked to the need of a higher number of packets to maintain the TCP workflow.
- **Latency:** The average time delays measured for both protocols using the diverse modes of operation are shown in Fig. 25. There exists a substantial time difference between the two protocols: MQTT seems to offer a smaller latency than CoAP. However, it must be taken into account that when using CoAP the Raspberry Pi has to measure the temperature, adding the delay associated to this operation to the total time delay. When using MQTT, the information is served when available, without adding the delay



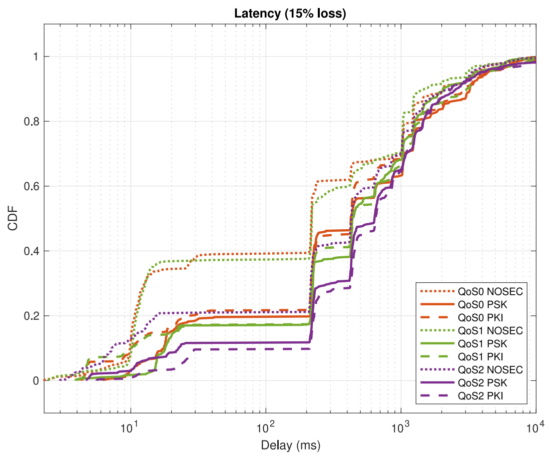
(a) 0% packet loss rate



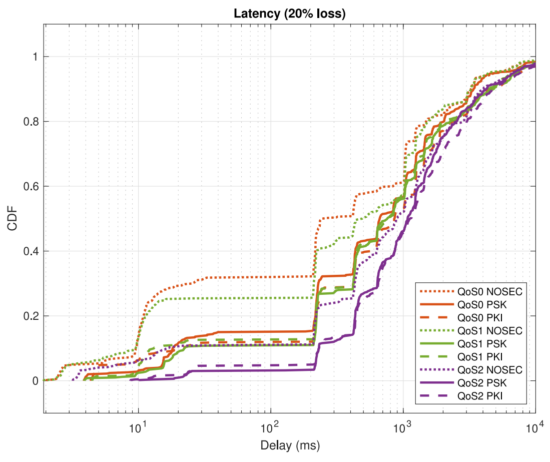
(b) 5% packet loss rate



(c) 10% packet loss rate



(d) 15% packet loss rate

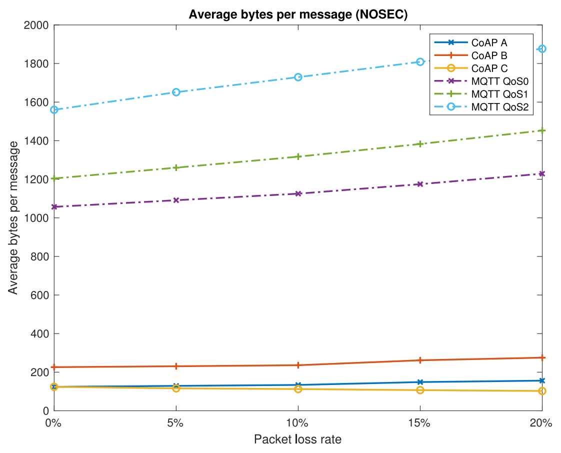


(e) 20% packet loss rate

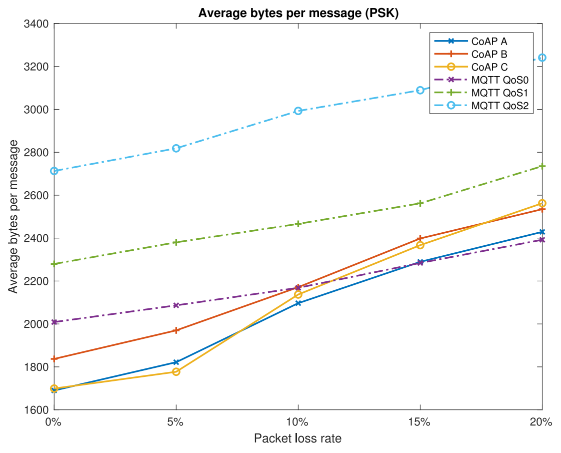
Fig. 21. Cumulative distribution function of time delays per packet loss rate (MQTT).

of the measurement. When subtracting the time needed for the measurement, both protocols offer a similar behaviour in terms of latency.

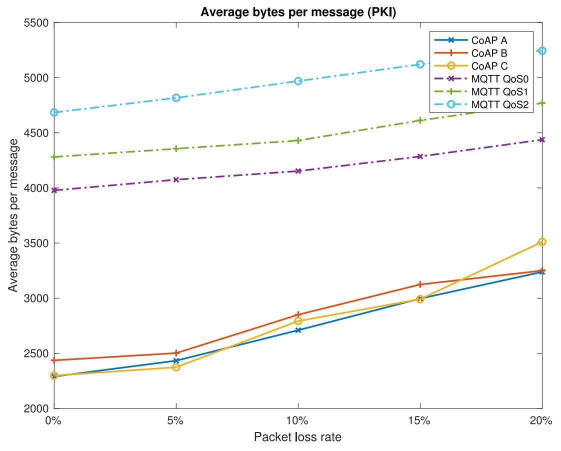
However, MQTT retransmissions follow the TCP scheme for retransmissions, what can lead to delays of several seconds. The retransmission timers of CoAP can be adjusted to avoid such big delays, even improving the performance of MQTT when the packet loss rate is high. However, by default the retransmission timers values of libcoap are high as it is shown in Fig. 24, leading to similar results to the ones offered by MQTT.



(a) Bytes per message (NOSEC)

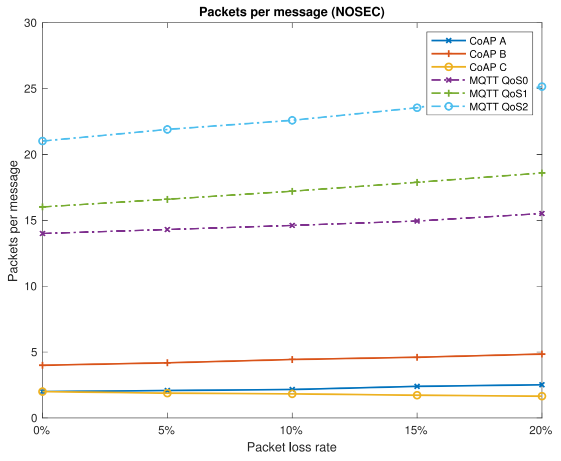


(b) Bytes per message (PSK)

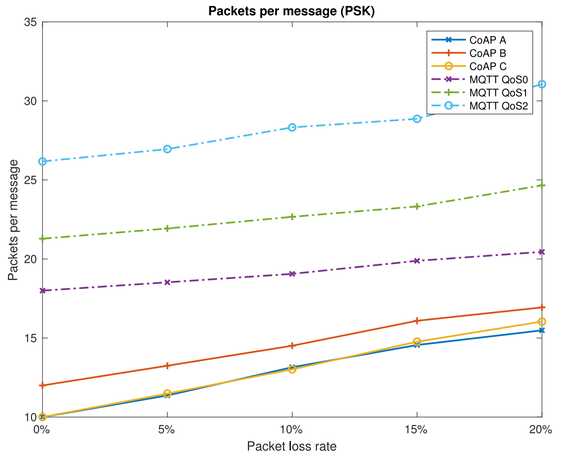


(c) Bytes per message (PKI)

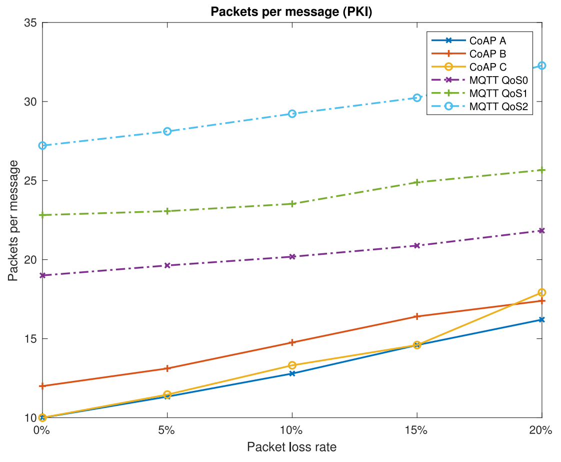
Fig. 22. Bandwidth usage (CoAP vs MQTT).



(a) PPM (NOSEC)



(b) PPM (PSK)

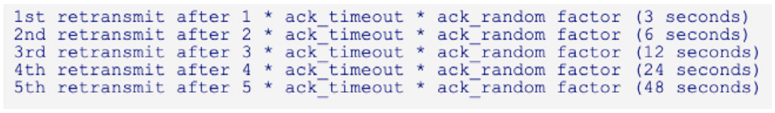


(c) PPM (PKI)

Fig. 23. Number of packets transferred per message (CoAP vs MQTT).

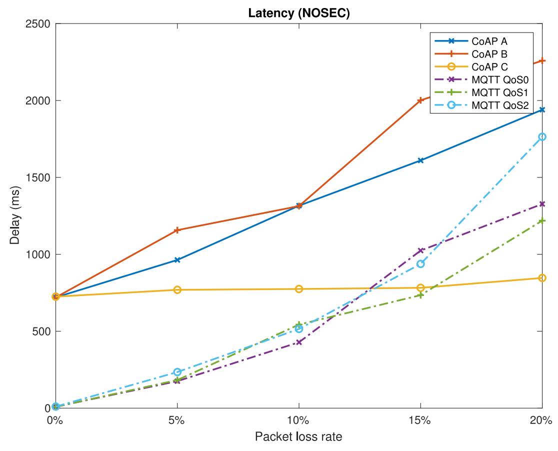
7. Conclusions and future work

In this paper we have studied the performance of CoAP and MQTT, both the unsecure and the secure versions, under different network conditions using a simple network scenario, contrasted in some simple situations with data obtained from an analytical model. The main conclusions of our study are that the MQTT protocol is more bandwidth demanding as it adds the TCP overhead. On the other hand, it offers different QoS and implicitly offers reliability (for running over TCP), while CoAP just offers a simple QoS and reliability mechanism through confirmable or non-confirmable messages. as expected, securing the

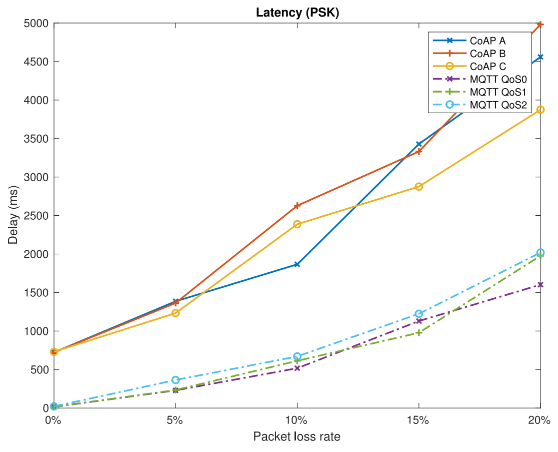


```
1st retransmit after 1 * ack_timeout * ack_random factor (3 seconds)
2nd retransmit after 2 * ack_timeout * ack_random factor (6 seconds)
3rd retransmit after 3 * ack_timeout * ack_random factor (12 seconds)
4th retransmit after 4 * ack_timeout * ack_random factor (24 seconds)
5th retransmit after 5 * ack_timeout * ack_random factor (48 seconds)
```

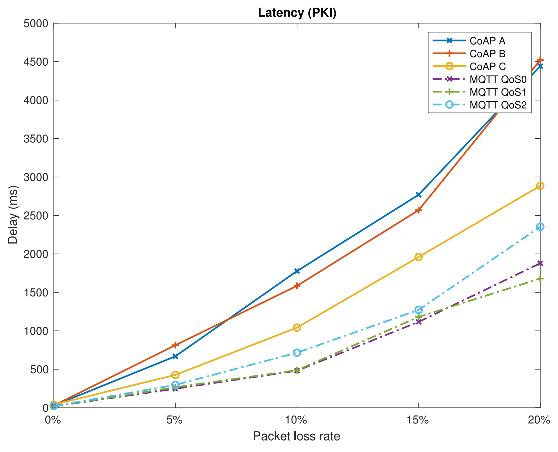
Fig. 24. CoAP message retry rules (https://libCoAP.net/).



(a) Time delay (NOSEC)



(b) Time delay (PSK)



(c) Time delay (PKI)

Fig. 25. Average time delay (CoAP vs MQTT).

communications through DTLS (CoAP) or TLS (MQTT) adds an important increase of the bandwidth usage – more than 1000% in CoAP and between 74 and just over 200% in MQTT – and the CPU usage – about 3.5% for PSK and 11.5% for PKI in CoAP and about 27% for PSK and 36% for PKI in MQTT – taking into account the modes of operation and QoS. Also, secure communications are more adversely affected by losses in terms of latency, than in lossless networks.

As future work, we are going to include direct energy measurements to our study beyond CPU cycles and transmitted packets. We will also use other network interfaces, such us WiFi or ZigBee, and consider more realistic loss models in wireless environments, to study the performance of the protocols in one-hop or multi-hop wireless networks. In addition, we will analyse alternative cipher suites and cached information extension [27], which is not supported by libcoap, because these could improve the overhead and CPU usage. Finally, we will evaluate the OSCORE performance for CoAP and a more lightweight version of MQTT, MQTT-SN.

CRediT authorship contribution statement

Victor Seoane: Software, Investigation, Writing – original draft, Visualization. **Carlos Garcia-Rubio:** Conceptualization, Methodology, Formal analysis, Writing – review & editing, Funding acquisition. **Florina Almenares:** Conceptualization, Methodology, Validation, Resources, Writing – review & editing, Supervision. **Celeste Campo:** Conceptualization, Methodology, Validation, Resources, writing – review & editing, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was supported in part by the Ministry of Economy and Competitiveness (Spain) under the project MAGOS (TEC2017-84197-C4-1-R) and by the Comunidad de Madrid (Spain) under the projects: CYNAMON (P2018/TCS-4566), co-financed by European Structural Funds (ESF and FEDER), and the Multiannual Agreement with UC3M in the line of Excellence of University Professors (EPUC3M21), in the context of the V PRICIT (Regional Programme of Research and Technological Innovation).

References

[1] IETF (Internet Engineering Task Force), The Internet of Things, 2021, https://www.ietf.org/topics/iot/.
[2] AIOTI (Alliance for Internet of Things Innovation, Research and innovation priorities for IoT: Industrial, business and consumer solutions, Tech. rep., European Commission, 2018, https://aioti.eu/wp-content/uploads/2018/09/AIOTI_IoT-Research_Innovation_Priorities_2018_for_publishing.pdf.
[3] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, M. Ayyash, Internet of Things: A survey on enabling technologies, protocols, and applications, IEEE Commun. Surv. Tutor. 17 (4) (2015) 2347–2376, http://dx.doi.org/10.1109/COMST.2015.2444095.
[4] Z. Shelby, K. Hartke, C. Bormann, The constrained application protocol (CoAP), RFC 7252, RFC Editor, 2014, https://tools.ietf.org/html/rfc7252.
[5] A. Banks, R. Gupta, MQTT Version 3.1.1, OASIS, 2014, http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[6] A. Banks, E. Briggs, K. Borgendale, R. Gupta, MQTT Version 5.0, OASIS, 2019, https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

[7] H. Tschofenig, T. Fossati, Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things, RFC 7925, RFC Editor, 2016, https://tools.ietf.org/html/rfc7925.

[8] K. Hartke, Observing resources in the constrained application protocol (CoAP), RFC 7641, IETF, 2015, https://tools.ietf.org/html/rfc7641.

[9] J. Krishnamurthy, M. Maheswaran, Chapter 5 - Programming frameworks for Internet of Things, in: R. Buyya, A.V. Dastjerdi (Eds.), Internet of Things, Morgan Kaufmann, 2016, pp. 79–102, http://dx.doi.org/10.1016/B978-0-12-805395-9.00005-8.

[10] L.L. Silva, Internet of Things: Pros and Cons of CoAP Protocol Solution for Small Devices, 2016.

[11] C. Bormann, CoAP-RFC 7252 Constrained Application Protocol, 2016, https://coap.technology/.

[12] D. Thangavel, X. Ma, A. Valera, H. Tan, C.K. Tan, Performance evaluation of MQTT and CoAP via a common middleware, in: IEEE 9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing, ISSNIP, 2014, pp. 1–6, http://dx.doi.org/10.1109/ISSNIP.2014.6827678.

[13] G. Selander, J. Mattsson, F. Palombini, Ephemeral Diffie-Hellman over COSE (EDHOC), Tech. rep., IETF Network Working Group, 2021, https://datatracker.ietf.org/doc/html/draft-ietf-lake-edhoc.

[14] M. Gunnarsson, J. Brorsson, F. Palombini, L. Seitz, M. Tiloca, Evaluating the performance of the OSCORE security protocol in constrained IoT environments, Internet Things 13 (100333) (2021) 2–16, http://dx.doi.org/10.1016/j.iot.2020.100333, https://www.sciencedirect.com/science/article/pii/S2542660520301645.

[15] J. Mattsson, F. Palombini, M. Vucinic, Comparison of CoAP Security Protocols, Tech. rep., IETF LWIG Working Group, 2020, https://tools.ietf.org/id/draft-ietf-lwig-security-protocol-comparison-05.html.

[16] W. Gao, J.H. Nguyen, C. Lu, D. Ku, Assessing performance of constrained application protocol (CoAP) in MANET using emulation, in: RACS '16: Proceedings of the International Conference on Research in Adaptive and Convergent Systems, 2016, pp. 103–108, http://dx.doi.org/10.1145/2987386.2987400.

[17] N. Naik, Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP, in: IEEE International Systems Engineering Symposium, ISSE, 2017, pp. 1–7, http://dx.doi.org/10.1109/SysEng.2017.8088251.

[18] R. Morabito, Z. Laaroussi, J. Jiménez, Evaluating the performance of CoAP, MQTT, and HTTP in vehicular scenarios, in: IEEE Conference on Standards for Communications and Networking, CSCN, 2018.

[19] A. Larmo, A. Ratilainen, J. Saarinen, Impact of CoAP and MQTT on NB-IoT system performance, Sensors 19 (1) (2019) http://dx.doi.org/10.3390/s19010007, https://www.mdpi.com/1424-8220/19/1/7.

[20] P. Thota, Y. Kim, Implementation and comparison of M2M protocols for Internet of Things, in: 4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science Engineering, ACIT-CSII-BCD, 2016, pp. 43–48.

[21] U. Tandale, B. Momin, D.P. Seetharam, An empirical study of application layer protocols for IoT, in: International Conference on Energy, Communication, Data Analytics and Soft Computing, ICECDS, 2017, pp. 2447–2451.

[22] E. Liri, P.K. Singh, A.B. Rabiah, K. Kar, K. Makhijani, K.K. Ramakrishnan, Robustness of IoT application protocols to network impairments, in: 2018 IEEE International Symposium on Local and Metropolitan Area Networks, LANMAN, 2018, pp. 97–103.

[23] T. Moraes, B. Nogueira, V. Lira, E. Tavares, Performance comparison of IoT communication protocols, in: IEEE International Conference on Systems, Man and Cybernetics, SMC, 2019, pp. 3249–3254, http://dx.doi.org/10.1109/SMC.2019.8914552.

[24] M. Martí, C. Garcia-Rubio, C. Campo, Performance evaluation of CoAP and MQTT-SN in an IoT environment, in: Proceedings 13th International Conference on Ubiquitous Computing and Ambient Intelligence, UCAmI, vol. 31(1), 2019, pp. 1–12, http://dx.doi.org/10.3390/proceedings2019031049, https://www.mdpi.com/2504-3900/31/1/49/pdf.

[25] V. Seoane, F. Almenares, C. Campo, C. Garcia-Rubio, Performance evaluation of the CoAP protocol with security support for IoT environments, in: 17th ACM Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, & Ubiquitous Networks, 2020, pp. 41–48.

[26] F. Kaup, P. Gottschling, D. Hausheer, PowerPi: Measuring and modeling the power consumption of the Raspberry Pi, in: 39th Annual IEEE Conference on Local Computer Networks, 2014, pp. 236–243.

[27] S. Santesson, H. Tschofenig, Transport layer security (TLS) cached information extension, RFC 7924, IETF, 2016, https://tools.ietf.org/html/rfc7924.

Víctor Seoane is a network planning and optimization engineer in Huawei. He received the M. Sc. Degree in Telecommunications Engineering and in Advanced Communications Technologies, both in 2020 from University Carlos III of Madrid. Contact him at vseoane@it.uc3m.es.

Carlos Garcia-Rubio is an associate professor at the Department of Telematic Engineering of the University Carlos III of Madrid. His research focus is centred in mobile and wireless networked computing systems, and in the design and performance evaluation of communication protocols, mainly at the transport and application layers. He received his Ph.D. degree from the Technical University of Madrid in 2000. Contact him at cgr@it.uc3m.es.

Florina Almenares Mendoza received the M.Sc. degree in telematics and the Ph.D. degree from the University Carlos III of Madrid, in 2003 and 2006, respectively. From 2004 to 2005, she was a Researcher and, then, became an Assistant Professor. Since 2008, she has been an Associate Professor with the Department of Telematics Engineering, University Carlos III of Madrid. Her research interests include trust and reputation management models, identity management, secure architectures and cyber security, and risk assessment. This research is applied to cloud computing, social networks, ubiquitous computing and the IoT, smart grids, and smart cities. Contact her at florina@it.uc3m.es.

Celeste Campo is an associate professor at the Department of Telematic Engineering of the University Carlos III of Madrid. Her research interests include design and performance evaluation of communication protocols for ad hoc networks, energy aware communications, and middleware technologies for pervasive computing. She received her Ph.D. degree from the University Carlos III of Madrid in 2004. Contact her at celeste@it.uc3m.es.