applied sciences

Article

E-MQTT: End-to-End Synchronous and Asynchronous Communication Mechanisms in MQTT Protocol

Yerin Im and Mingyu Lim *

Department of Smart ICT Convergence, Konkuk University, 120 Neungdong-ro, Gwangjin-gu, Seoul 05029, Republic of Korea; erimolet@konkuk.ac.kr
* Correspondence: mlim@konkuk.ac.kr; Tel.: +82-2-2049-6270

Abstract: Message Queuing Telemetry Transport (MQTT) enables asynchronous confirmation of message reception by brokers but lacks a way for publishers to know when subscribers receive their messages without adding additional communication overhead. This paper addresses this problem by improving MQTT to establish end-to-end communication between a publisher and subscribers, reducing message exchanges, using what is called End-to-End MQTT (E-MQTT). In E-MQTT, a publisher sets the number of responses that it will wait for when it sends a message. After the broker collects the response messages from subscribers, it sends one aggregated response back to the publisher. The publisher also can receive the response message synchronously or asynchronously. Experimental results consistently show that E-MQTT outperforms traditional MQTT in terms of delay, especially when the publisher needs to monitor when its query message is received by subscribers. Although E-MQTT packets are slightly larger due to additional fields, the difference in packet size compared to MQTT is not significant.

Keywords: asynchronous communication; end-to-end communication; MQTT; publish–subscribe model; synchronous communication

Citation: Im, Y.; Lim, M. E-MQTT: End-to-End Synchronous and Asynchronous Communication Mechanisms in MQTT Protocol. *Appl. Sci.* **2023**, *13*, 12419. https://doi.org/10.3390/app132212419

Academic Editors: Ryan Gibson and Hadi Larijani

Received: 17 October 2023
Revised: 10 November 2023
Accepted: 15 November 2023
Published: 16 November 2023

1. Introduction

Message Queuing Telemetry Transport (MQTT) [1] is a message transmission protocol based on the ISO standard (ISO/IEC PRF 20922) communication model. MQTT operates over the TCP/IP protocol and employs a reliable and asynchronous publish–subscribe communication model. Within this model, after the publisher publishes a message, it does not care which subscriber will receive the message. Instead, the publisher promptly proceeds with subsequent tasks without waiting for feedback from the broker. The versatility of MQTT is evident through its extensive range of applications, which encompass various domains [2–13], including (but not limited to) the Internet of Things (IoT), home and industrial automation, telemetry and remote environment monitoring, energy management, smart agriculture, healthcare, transportation, and social networking systems. As described in Section 2, previous research efforts have explored various aspects of MQTT, including research to enhance MQTT functionalities [2,4,14–22], comparative analyses with other protocols [23–30], support for synchronous communication [31–40], and the facilitation of end-to-end services for security or quality of service (QoS) [41–50].

As the MQTT environments have been growing and becoming more complicated, managing an ever-expanding array of client devices (comprising publishers and subscribers) presents considerable challenges. Therefore, there arises a need for a control device to intermittently assess the status or functionalities of these devices communicating with MQTT. For example, in a healthcare system, a medical professional with a control device might seek to ascertain whether the attached sensing devices on patients can successfully capture specific health information before he/she conducts a monitoring task. In this case, he/she (a publisher) can publish a query message with a request topic and check the status of devices by receiving response messages published with another feedback topic by the devices. This real-world scenario reflects the fact that the publish–subscribe model needs a new communication requirement. That is, publishers need to issue query messages and confirm the moment when subscribers have received these query messages. In other words, the publish–subscribe model could need to be combined with request–response patterns [51] in future communication environments, especially when a publisher wants to check the status of subscribers.

In the existing publish–subscribe models like MQTT, the publisher's awareness is limited to when the broker acknowledges a published message with QoS levels 1 and 2 [52]. As MQTT uses TCP/IP as the underlying transport-layer protocol, a message is guaranteed to be delivered reliably to recipients. However, if the publisher seeks to check the moment when a message is delivered to subscribers, it requires an intricate procedure. Upon reception of a query message from the publisher, the subscriber that holds a subscription for the concerned topic should publish a response message on a new topic to signify reception of the query message. Before that, the publisher should subscribe to this new topic to receive the response message from subscribers. We refer to this specific challenge as the "end-to-end communication problem" within the context of MQTT. Section 3.1 provides an in-depth examination of this problem. Although there have been many previous research efforts regarding MQTT, it is worth noting that none of these prior research endeavors have directly addressed the end-to-end communication problem.

In this paper, we propose a new end-to-end MQTT (E-MQTT) that supports end-to-end communication between the publisher and the subscriber of the MQTT by adding the request–response pattern to the publish–subscribe model. This enhancement empowers publishers with the ability to check synchronously or asynchronously the moment when a message is delivered to subscribers. E-MQTT exhibits minimal divergence from the structure of MQTT packets, incorporating only a handful of additional fields. Despite being an extension of MQTT, E-MQTT still retains the intrinsic characteristics of the publish–subscribe model while augmenting it with the novel feature of the request–response model for end-to-end communication. The publisher can check when the subscriber has received the message without a separate additional process of subscribing to a new topic. To support the E-MQTT functionalities, we introduce a new QoS level 3 to MQTT. Although E-MQTT increases the QoS levels from MQTT, it does not imply that E-MQTT provides more reliable communication than MQTT. Instead, E-MQTT adds the functionality of request–response patterns to the publish–subscribe model for end-to-end communication between a publisher and subscribers. In the new feature of E-MQTT, the publisher specifies the minimum number of response packets that it wants to wait for. Should this count be set to 0, the publisher proceeds without waiting for a response, which is called asynchronous E-MQTT. In this scenario, the broker forwards the first response packet from a subscriber to the publisher, allowing the publisher to asynchronously check the moment of the response from one subscriber. Alternatively, if the minimum number of response packets is more than 0, the publisher can perform the next task only if it receives an aggregated response packet from the broker that has received the predetermined number of response packets from the subscribers. If the publisher receives an aggregated response packet, it knows that at least the predetermined number of subscribers have received its query message, and then continues to conduct the next task. This mode is referred to as synchronous E-MQTT. Additionally, the publisher has an option to specify particular subscribers to wait for, if it has knowledge of subscribers' identifiers (IDs).

As E-MQTT reduces the required number of message exchanges compared to MQTT, experimental results show that E-MQTT consistently outperforms the conventional MQTT primarily in terms of the end-to-end delay when the publisher needs to check the status of subscribers. Instead, E-MQTT increases the size of some packets because it needs to include the additional information of the minimum number of response packets that the publisher wants to wait for. This minor modification causes a marginal increase in packet size compared to MQTT. The difference in packet size becomes notable only if the publisher adds numerous optional subscriber IDs. Despite this slight variation in packet size, the overall advantages offered by E-MQTT, especially in terms of improved delay performance, outweigh any negligible increase in packet size.

In summary, the contributions of our research work are as follows:

- Modified MQTT specifications (E-MQTT) to add the functionality of request–response patterns for end-to-end communication between a publisher and subscribers.
- Enabled checking the moment when its message is received by subscribers.
- Supported end-to-end communication in two modes according to the configuration of the minimum number of response packets: synchronous and asynchronous modes.
- Implemented E-MQTT and compared performance with MQTT.
- Reduced delay of end-to-end request–response communication.

The remainder of this paper is organized as follows. In Section 2, we introduce related works about MQTT. In Section 3, we propose the synchronous and asynchronous end-to-end communication methods of E-MQTT. In Section 4, we describe the main classes, methods, and additional fields required for the implementation of E-MQTT. In Section 5, we analyze the performance of E-MQTT and compare it to the existing MQTT through the measurement of communication delay and message size. Finally, we conclude the paper with future research plans in Section 6.

2. Related Works

In this section, we describe various research efforts regarding MQTT. There have been research approaches to use MQTT as an application, to improve functionalities of MQTT, to compare MQTT with other protocols, to combine synchronous communication with MQTT, and to focus on end-to-end services of MQTT in terms of security or QoS.

2.1. MQTT

MQTT version 3.1.1 [1] encompasses a comprehensive suite of 14 packets, offering support for three distinct QoS levels to ensure reliability. MQTT provides three QoS levels spanning from 0 to 2. These QoS levels effectively dictate the precision with which messages are relayed in communication scenarios involving the publisher and the broker, as well as between the broker and the subscriber. With the ascending QoS level, MQTT extends its support for increasingly precise transmission semantics [52]. At QoS level 0, a published message possesses the potential to be delivered to the receiver (either the broker or the subscriber) at most once. QoS level 1, on the other hand, guarantees that a given message is delivered at least once. Lastly, QoS level 2 is structured to ensure that a message is delivered exactly once via a four-way handshaking process.

The most recent MQTT version 5.0 [53] added twenty items of new features including user properties, shared subscriptions, payload format description, request–response pattern, topic alias, enhanced authentication, and flow control. Among them, the support of request–response patterns reflects the need for another communication model in the publish–subscribe model. This new feature facilitates request–response communication between the publisher and the subscriber by exchanging information such as response topic and correlation data during the connection and publishing stages. However, this feature still requires two separate publish–subscribe sessions, which is the main cause of communication overhead.

2.2. Applications of MQTT

MQTT finds application across diverse domains, encompassing platforms such as Floodnet [7,8], Facebook Messenger [12], and wireless heart rate monitoring systems [5]. MQTT's versatility extends to sectors like healthcare [54], energy, and utilities, as well as social networking systems [13]. Grgić's research [6] innovatively employed MQTT within agricultural drying processes, while MQTT-driven home automation systems empowered remote monitoring of home environments, along with the execution of necessary tasks [3].

2.3. Improvement of MQTT

Numerous research works have tailored MQTT to suit their distinct objectives, appending bespoke functionalities. For example, MQTT-S [2] adapted MQTT to cater to the needs of smaller sensor–actuator (SA) devices within sensor networks, while MQTT-G [4] infused geolocation data into an existing MQTT. Kim's study [14] introduced a priority-QoS flag within the MQTT packet's fixed header to incorporate the priority concept. The research of Ali et al. [15] involved developing a TCP-based MQTT protocol with retransmission rules to improve content delivery probability, ensuring reliable and secure Internet of Everything (IoE) services for smart city applications. Palmese et al. [16] addressed the optimization of a wireless sensor network (WSN) using the MQTT-SN protocol for IoT applications by introducing a dynamic QoS controller.

Various investigations concentrate on enhancing MQTT's security, as exemplified by PICADOR [19], the incorporation of encryption features into the Narada Brokering System, along with separate security nodes [18] and the application of Secure MQTT [17]. Recently, research efforts are more focused on performance improvement. Rocha et al. [21] proposed a QoS dynamic adaptation method (DAM) for WSN to ensure better message delivery. The main idea was to select the proper QoS level based on network latency conditions. Zhang et al. [20] established a unified architecture and proposed a delay-reliability-aware MQTT quality of service selection algorithm to effectively reduce delay and packet loss, meeting the diverse quality of service requirements in electric IoT. Alshammari [22] implemented a real-time remote patient monitoring system utilizing IoT technology to ensure the precision of vital real-time signals. These signals are transmitted from the proposed system to a website through MQTT.

2.4. Comparison between MQTT and Other Protocols

There have been also research works measuring the performance of MQTT compared to other protocols [23–25]. Recent comparisons have also emerged within specific contextual settings. Nguyen et al. [27] assess the efficacy of AMQP and MQTT in the context of an internet radio system. Naik et al. [29] briefly analyze advantages and disadvantages of MQTT, CoAP, AMQP, and HTTP across various aspects, including architecture, message size, cache support, quality of service, and more. Palmese et al. [30] conducted a comparative analysis of two protocols, MQTT-SN and CoAP, in the publish–subscribe version as outlined in a recent IETF draft. The experiments indicate that CoAP offers a viable alternative to MQTT-SN in publish–subscribe scenarios, with CoAP being preferred for highly dynamic networks. Silva et al. [26] undertake a comparison of the performance of MQTT, CoAP, and OPC UA within information-centric networks and industrial IoT environments. Gemirter et al. [28] compare AMQP, MQTT, and HTTP by leveraging real-time public smart city data.

2.5. Synchronous and Asynchronous Communication

While prevailing messaging protocols and platforms such as Paho [31], AMQP [32], RabbitMQ [33], and Apache Kafka [34] are built upon the publish–subscribe model that predominantly features asynchronous communication, they also incorporate provision for synchronous communication. However, it is important to note that the existing protocols and platforms primarily cater to synchronous communication between a publisher and a broker.

To address the scalability problem associated with synchronous communication, Sen et al. [35] proposed Nucleus, which presents a container architecture comprising stateless brokers. Jaloudi [39] proposed a strategy for merging synchronous and asynchronous communication models in industrial IoT environments by integrating MODBUS TCP and MQTT to suit specific scenarios. Bagaskara et al. [36] undertook a performance analysis of RabbitMQ and Apache Kafka within fog computing environments, given that synchronous communication models like HTTP can result in performance degradation in such contexts. Pratama et al. [37] tackled the issue of synchronous communication in the domain of public transportation monitoring systems, achieving this through development of an asynchronous message broker middleware. Shafabakhsh et al. [38] conducted a comparative assessment of the synchronous and asynchronous models of interprocess communication within distributed microservices. Kul et al. [40] introduced a hybrid approach encompassing both synchronous and asynchronous communication mechanisms in event-based microservices, aimed at detecting vehicle information from real-time streamed video sources.

2.6. End-to-End Services of MQTT

Existing research concerning end-to-end support within MQTT is predominantly focused on security investigations [41–46], with some research works [47–50] also touching on end-to-end services. Park et al. [42] introduced a security architecture and protocols designed to establish MQTT security within wireless sensor networks. Spina et al. [43] presented a lightweight security mechanism for managing security levels in MQTT. Bashir et al. [46] proposed a security mechanism that uses lightweight cryptographic operations to provide end-to-end data confidentiality while efficiently managing patient mobility within hospital or home premises. SEEMQTT [41] is a framework that ensures end-to-end data confidentiality, integrity, and authorization in MQTT-based communication for mobile IoT systems. Winarno et al. [44] presented the design of a secure end-to-end encryption MQTT protocol using lightweight cryptographic algorithms and Galantucci secret sharing. Chien et al. [45] discussed the importance of over-the-air (OTA) updates for secure IoT systems and highlighted the widespread use of MQTT as an IoT communication protocol.

Govindan et al. [47] conducted comprehensive analysis of end-to-end service assurance parameters, with a special focus on content delay and probability of content delivery, for MQTT-SN in healthcare IoT applications. D'Ortona et al. [48] presented a comprehensive end-to-end IoT system that monitors information concerning road users, which is generated through wearable sensors. Ali et al. [49] discussed the implementation of an improved TCP-based MQTT network for efficient communication in smart cities, emphasizing quality of service (QoS) considerations for reliable end-to-end service assurance. Jo et al. [50] introduced a self-adaptive resource management framework for large-scale cyber–physical systems (CPS) that maintains low end-to-end monitoring delays while ensuring high monitoring resolution.

3. E-MQTT

Our design approach for E-MQTT prioritizes the preservation of the original MQTT packet structure and protocols to the maximum extent feasible. Guided by this design philosophy, the proposed E-MQTT seamlessly incorporates end-to-end communication functionality into the existing MQTT, integrating with the existing QoS 0, 1, and 2. To distinguish E-MQTT from the conventional MQTT, we introduce a new QoS level 3. In asynchronous E-MQTT, the publisher is not suspended after sending the PUBLISH packet and immediately proceeds to the subsequent task. In this mode, the publisher can asynchronously receive a response message from the broker, which forwards the initial response message originating from subscribers. In contrast, within the synchronous E-MQTT configuration, on transmitting the PUBLISH packet, the publisher enters a suspended state. This suspension persists until the publisher receives a consolidated response message orchestrated by the broker. This aggregate message encapsulates the actual responses generated by individual subscribers.

3.1. End-to-End Communication Problem of MQTT

When a publisher seeks the moment when a query message is received by a subscriber, the MQTT protocol involves the following procedural steps, illustrated in Figure 1. To facilitate the receipt of a response message from a subscriber, the publisher must subscribe to a new topic designated as B (1). The message publishing and subscription to topic B is an additional publish–subscribe session for the exchange of a response message. Subsequently, the publisher sends the original query message with the topic A (2), which the subscriber receives (3). The message publishing and subscription to topic A is the original publish–subscribe session for the exchange of a query message. To reply to the query message, the subscriber needs to publish a new response message with the new topic B (7). Upon reception of the published response message attached to topic B, the publisher confirms that the query message has been received by a subscriber before this response message (8). As a result, MQTT needs two distinct publishing sessions and requires 18 messages exchanged among the publisher, broker, and subscriber.

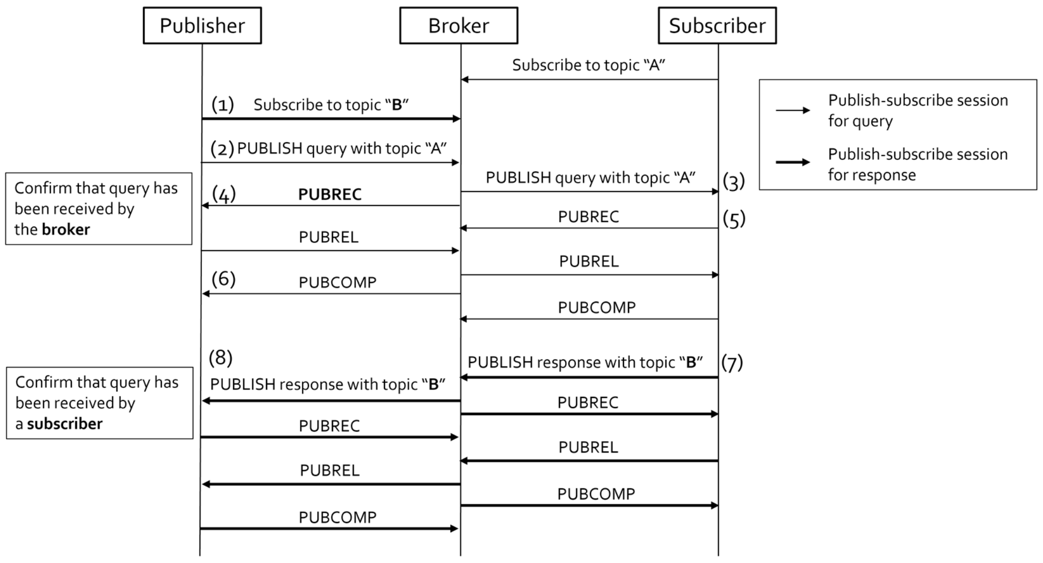

Publisher
Broker
Subscriber
Subscribe to topic "A"
(1) Subscribe to topic "B"
(2) PUBLISH query with topic "A"
PUBLISH query with topic "A" (3)
Confirm that query has been received by the broker
(4) PUBREC
PUBREC (5)
PUBREL
PUBREL
(6) PUBCOMP
PUBCOMP
(8) PUBLISH response with topic "B"
PUBLISH response with topic "B" (7)
Confirm that query has been received by a subscriber
PUBREC
PUBREC
PUBREL
PUBREL
PUBCOMP
PUBCOMP
Publish-subscribe session for query
Publish-subscribe session for response

Figure 1. End-to-end confirmation process of query reception by one subscriber in MQTT (QoS 2).

Figure 2 illustrates how the publisher confirms when n subscribers have received the query message in MQTT. This figure shows only the query and response PUBLISH messages and omits the other PUBREC, PUBREL, PUBCOMP, and subscription messages for simplicity. By receiving n PUBLISH messages with topic B (1), the publisher can check the reception by n subscribers. As the number of subscribers increases, the number of exchanged messages also increases proportionally.

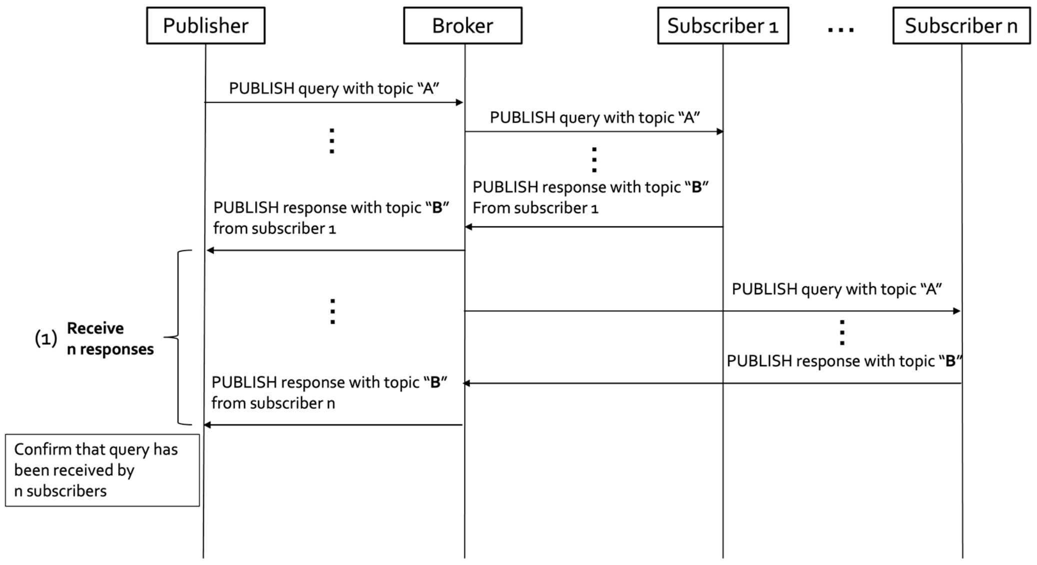

Publisher
Broker
Subscriber 1
...
Subscriber n
PUBLISH query with topic "A"
PUBLISH query with topic "A"
PUBLISH response with topic "B" From subscriber 1
PUBLISH response with topic "B" from subscriber 1
PUBLISH query with topic "A"
(1) Receive n responses
PUBLISH response with topic "B"
PUBLISH response with topic "B" from subscriber n
Confirm that query has been received by n subscribers

Figure 2. End-to-end confirmation process of query reception by n subscribers in MQTT (QoS 2).

Through modifications to MQTT, E-MQTT enables publishers to verify the moment of query message reception by subscribers with only one publishing session, effectively mitigating communication overhead. As shown in Figure 3, E-MQTT introduces a mechanism that permits publishers to check the moment when the query message is received by subscribers upon the receipt of a PUBREC packet (4), thus streamlining the necessary actions. In MQTT, the occurrence of a PUBREC packet ((4) in Figure 1) denotes the broker's reception of the published message, not that of a subscriber. On the other hand, in E-MQTT, the PUBREC packet implies that one or more subscribers have received the published query message. In E-MQTT, the number of exchanged messages is reduced to 9 and the broker forwards the PUBLISH and PUBREC packets to the final target after additional processing for the end-to-end communication.

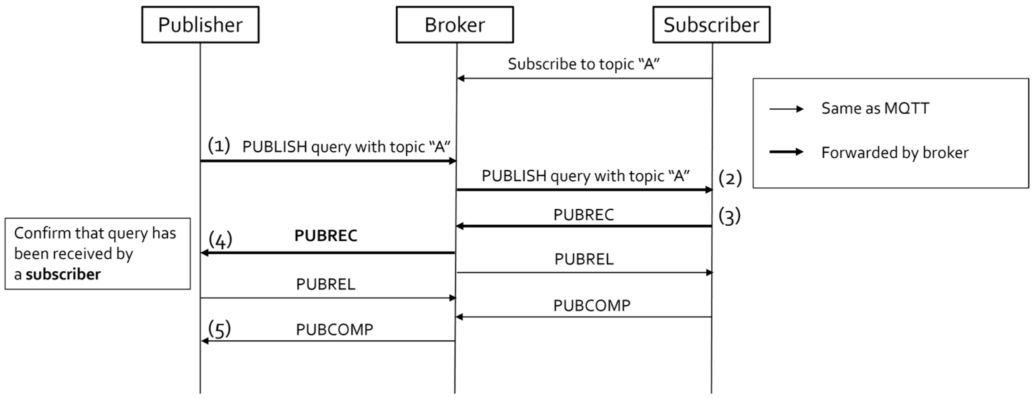


Figure 3. End-to-end confirmation process of query reception by one subscriber in E-MQTT (QoS 3).

Figure 4 illustrates how the publisher confirms when n subscribers have received the query message in E-MQTT. This figure shows only the query PUBLISH and response PUBREC messages and omits the other PUBREL, PUBCOMP, and subscription messages for simplicity. When the publisher sends the query PUBLISH message, it also designates n responses that it will wait for (1). As the broker collects n PUBREC messages from subscribers (2), the publisher can check the reception by n subscribers by receiving one PUBREC message (3). Therefore, it also reduces the number of exchanged messages between the publisher and broker. We describe details of E-MQTT in the following sub-sections.

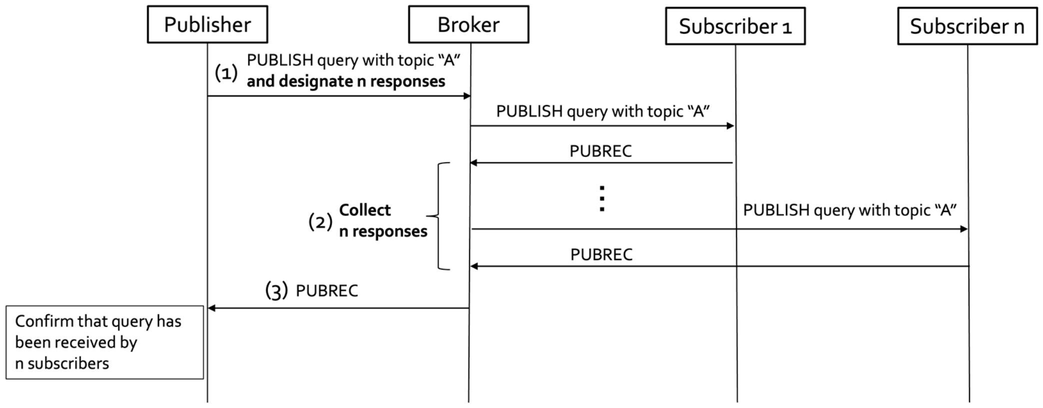

Publisher
Broker
Subscriber 1
Subscriber n
(1) PUBLISH query with topic "A" **and designate n responses**
PUBLISH query with topic "A"
PUBREC
(2) **Collect n responses**
PUBLISH query with topic "A"
PUBREC
(3) PUBREC
Confirm that query has been received by n subscribers

Figure 4. End-to-end confirmation process of query reception by n subscribers in E-MQTT (QoS 3).

3.2. Overview of End-to-End Communication Methods

E-MQTT employs a four-way handshaking process connecting the publisher and subscriber through the broker. Figure 5 shows the end-to-end communication process in E-MQTT. The communication flow within E-MQTT mirrors that of MQTT QoS 2, which implies that all the MQTT packets are guaranteed to be delivered exactly once to a target node. In E-MQTT, the main purpose is not that the publisher wants to confirm whether its message is successfully received by subscribers, but that the publisher can check when its message is received by at least some specified number of subscribers. To this end, E-MQTT modified the message publishing process from MQTT as shown in the gray-colored tasks in Figure 5. Initially, the publisher configures the minimum number of packets to wait for (*waited_packet_num*) and stores the PUBLISH packet to be sent (1). If the number of response packets to be waited for is 1 or more, E-MQTT initiates the synchronous end-to-end communication. If the number is 0, the asynchronous end-to-end communication proceeds. While the *waited_packet_num* field is a mandatory component, the publisher also can optionally set a list of subscriber identifiers (*subscriber_id(s)*). This additional field facilitates the identification of designated subscribers by the publisher. Each subscriber identifier (ID) is a globally unique label. To use the *subscriber_id(s)* field, the publisher must request a list of subscriber IDs from the broker, facilitating the selection of a subset of subscribers. However, it is worth noting that since this additional constraint could potentially infringe upon the principle of the publish–subscribe model's loosely anonymous communication, the inclusion of the *subscriber_id(s)* field remains optional.

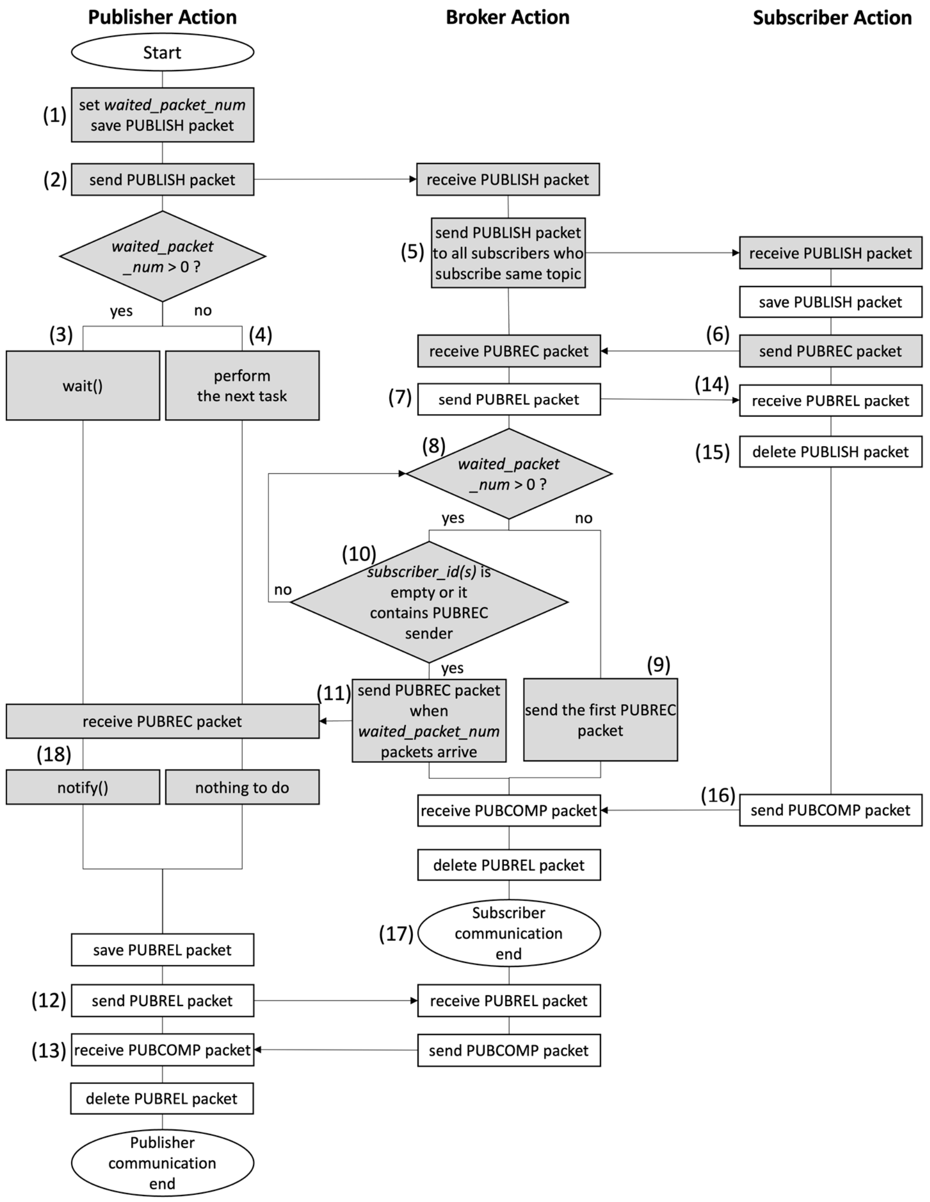


Figure 5. The communication process of E-MQTT.

Subsequently, the publisher proceeds to release the PUBLISH packet with topic *T* (2). This PUBLISH packet is transmitted to the broker. In the synchronous communication scenarios, the publisher enters the suspended state after transmitting the PUBLISH packet (3).

In contrast, during asynchronous communication, the publisher remains unsuspended and seamlessly advances to the subsequent task (4).

Upon receiving the PUBLISH packet, the broker promptly disseminates it to all subscribers of topic *T* like MQTT (5). Once the PUBLISH packet is stored, a subscriber initiates a response via the PUBREC packet (6). When the broker receives the PUBREC packet from a subscriber, the broker answers with the PUBREL packet (7). Following this, the broker evaluates the *waited_packet_num* and *subscriber_id(s)* fields to determine whether it forwards the PUBREC packet to the publisher or not (8). If *waited_packet_num* is less than or equal to 0 (asynchronous E-MQTT), the broker forwards the first PUBREC packet (9). Conversely, if *waited_packet_num* is greater than 0 (synchronous E-MQTT), the broker scrutinizes the optional *subscriber_id(s)* field to determine whether it counts the number of received PUBREC packets or not (10). If *subscriber_id(s)* is empty or it contains the sender of the PUBREC packet, the broker increases the count of received response packets. Subsequently, the broker compares the number of PUBREC packets it has gathered from the subscribers with *waited_packet_num*. If the number of received response packets becomes equal to or greater than *waited_packet_num*, the broker sends one aggregated PUBREC packet to the publisher (11). This action enables the publisher to validate that the number of subscribers, as set by itself, has successfully received the published message.

Upon reception of the PUBREC packet, the publisher proceeds to remove the stored PUBLISH packet. Subsequently, the publisher transmits the PUBREL packet to the broker and stores the PUBREL packet (12). Once the PUBCOMP packet is received from the broker (13), the communication cycle on the publisher's end concludes.

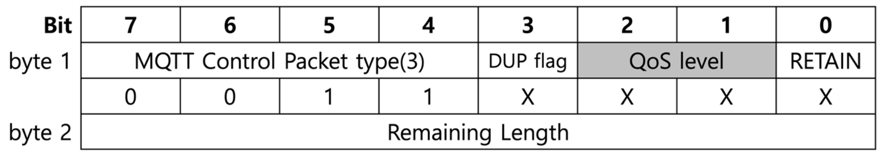
Upon reception of the PUBREL packet (14), the subscriber proceeds to remove the stored PUBLISH packet (15) and promptly generates a response by transmitting the PUBCOMP packet (16). Subsequently, the broker concludes the QoS 3 publishing process upon the receipt of the PUBCOMP packet and deletes the stored PUBREL packet (17).

Within synchronous E-MQTT, the publisher remains in a suspended state starting from the moment the PUBLISH packet is dispatched (3) until the corresponding PUBREC packet is received from the broker (18). However, the publisher can transition out of this suspended state should a timer expire, prompted by exceptional circumstances like packet loss or subscriber failure. Conversely, in the context of asynchronous E-MQTT, the complete communication process unfolds asynchronously, devoid of such suspensions.

In synchronous E-MQTT, the publisher and the subscriber are weakly coupled with the minimum number of waiting packets. However, if the publisher specifies a list of subscriber IDs as well as the number of waiting packets, it becomes strongly coupled to the designated subscribers. Conversely, in asynchronous E-MQTT, as the publisher does not need to wait for the PUBREC packet, the publisher and the subscriber are completely decoupled. Hence, applications employing E-MQTT possess the flexibility to opt for varying degrees of decoupling between publishers and subscribers, tailoring the approach to meet their specific requirements.

3.3. Packet Format

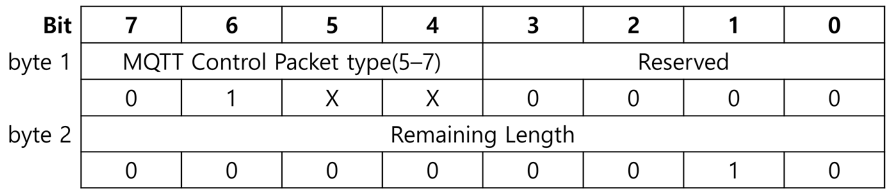
In the existing MQTT, the first byte of the fixed header allots bits 7 to 4 for denoting the packet type. This allocation encompasses a total of 14 packets, ranging numerically from 0001 to 1110. The structure of a fixed header for a conventional MQTT PUBLISH packet is illustrated in Figure 6. Within the first byte, bit 3 is used for indicating DUP flags, whereas bits 2–1 are deployed to signify QoS levels. These QoS levels span from 0 to 2, while the unused value 3 has been repurposed to accommodate the proposed QoS level 3. That is, E-MQTT does not change the format of the PUBLISH packet but assigns the value 3 in the QoS level field to distinguish from the existing QoS levels 0, 1, and 2 of MQTT. This QoS level 3 necessitates the specification of the minimum response packet count (*waited_packet_num*) and, optionally, the inclusion of subscriber IDs (*subscriber_id(s)*) within the variable header of the PUBLISH packet.



Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet type(3)				DUP flag	QoS level		RETAIN
	0	0	1	1	X	X	X	X
byte 2	Remaining Length							

Figure 6. The fixed header of PUBLISH packet of the existing MQTT.

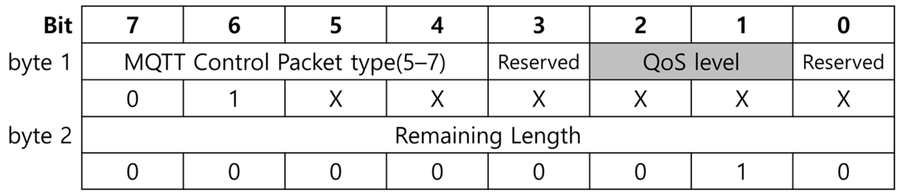
Figure 7 represents the fixed headers of PUBREC, PUBREL, and PUBCOMP, which are relevant packets in the publishing process with QoS 2 within the conventional MQTT. The initial byte of the fixed header reveals that bits 7–4 serve to signify the packet type, while bits 3–0 remain unutilized. The PUBREL and PUBCOMP packets are used by both MQTT and E-MQTT in the same way.



Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet type(5–7)				Reserved			
	0	1	X	X	0	0	0	0
byte 2	Remaining Length							
	0	0	0	0	0	0	1	0

Figure 7. The fixed header of PUBREC, PUBREL, and PUBCOMP packets of the existing MQTT.

Given MQTT's allocation of four bits for denoting packet types, any addition of two or more packets for E-MQTT would necessitate an expansion of the packet type bit count. To circumvent this concern, the solution lies in QoS level 3 adopting the identical packet types as those within QoS level 2. Instead, we leverage the unused bits within the packets to incorporate a QoS field, effectively distinguishing packets meant for QoS level 3 from those of QoS level 2. Figure 8 shows the modified fixed header of the PUBREC packet, reflecting E-MQTT integration. The new QoS field is allocated to bits 2–1, mirroring the QoS bits within the existing PUBLISH packet. Under the QoS level 2, the communication protocol aligns with QoS 2 standard. However, in cases where the QoS value is 3, the communication process adheres to the new QoS 3.



Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet type(5–7)				Reserved	QoS level		Reserved
	0	1	X	X	X	X	X	X
byte 2	Remaining Length							
	0	0	0	0	0	0	1	0

Figure 8. The fixed header of PUBREC, PUBREL, and PUBCOMP packets of E-MQTT.

The PUBLISH packet of E-MQTT is created when a publisher sends a message ((1) in Figure 9). The PUBLISH packet is received by a broker ((1) in Figure 10) that delivers it to subscribers ((1) in Figure 11). The PUBREC packet of E-MQTT is created and sent by a subscriber ((2) in Figure 9) after it receives a PUBLISH packet. The PUBREC packet is received by a broker ((2) in Figure 10) that forwards it to a publisher ((2) in Figure 11).

In summary, we add QoS level 3 in PUBLISH and PUBREC packets by using the existing QoS field or the reserved field to distinguish E-MQTT packets from the other existing MQTT packets. In the PUBLISH packet, we also add the field for the minimum number of response packets and an optional list of subscriber IDs in the variable header.

3.4. Detailed Process of End-to-End Communication

Displayed in Figure 9 are diagrams delineating the execution process of the E-MQTT from the perspective of the publisher. Firstly, the user's publish request, employing QoS 3, is directed to the main thread for processing. Within this main thread are the mandatory *waited_packet_num* field and the optional *subscriber_id* field(s). If the publisher configures the *waited_packet_num* field to 0 (asynchronous mode), it does not stop its execution but proceeds to subsequent tasks after the delivery of the PUBLISH packet to a dedicated sending thread. Conversely, when the *waited_packet_num* is greater than 0 (synchronous mode) and the *subscriber_id* field is not empty, the publisher undertakes a comparison between the *waited_packet_num* field and the *subscriber_id* field(s). If the *waited_packet_num* field exceeds the count of the *subscriber_id* fields, the publisher returns an error and terminates the publish session. However, the opposite case (where the *waited_packet_num* is less than or equal to the number of *subscriber_id* fields) is acceptable because the publisher can wait for a lower number of responses among the designated subscribers.

Following scrutiny of the packet fields, the main thread registers the PUBLISH packet within an unacknowledged (termed unacked) packet list. Subsequently, the main thread dispatches the PUBLISH packet to the sending thread, which then orchestrates transmission to the broker. In scenarios where the publisher specifies the minimum count of response packets to be awaited (indicating synchronous mode), the main thread registers the PUBLISH packet within a synchronizer object. Here, the main thread calls the *wait()* function to enter a suspended state [54]. During this phase, the publisher sets a timer to define the upper limit of waiting time. Upon expiration of this timer, the publisher interprets it as an indication of some subscribers' unavailability, thus prompting termination of the publish session. The main thread resumes execution upon the receipt of a PUBREC packet from the broker, triggered by a processing thread. This resumption is facilitated by the invocation of the *notify()* function, effectively awakening the main thread from its suspended state.

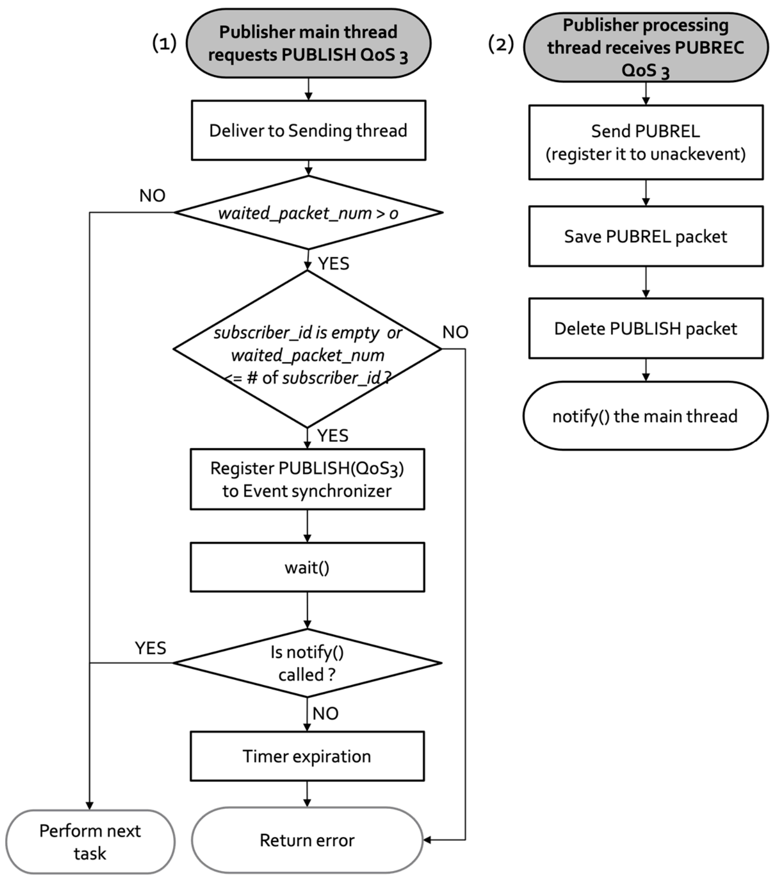

(1)
Publisher main thread requests PUBLISH QoS 3
Deliver to Sending thread
NO
waited_packet_num > 0
YES
subscriber_id is empty or waited_packet_num <= # of subscriber_id ?
NO
YES
Register PUBLISH(QoS3) to Event synchronizer
wait()
YES
Is notify() called ?
NO
Timer expiration
Perform next task
Return error
(2)
Publisher processing thread receives PUBREC QoS 3
Send PUBREL (register it to unackevent)
Save PUBREL packet
Delete PUBLISH packet
notify() the main thread

Figure 9. Flowcharts showing the publisher-side execution flow of E-MQTT.

The publisher's processing thread is responsible for handling incoming packets, which are received from a dedicated receiving thread. During the synchronous communication within E-MQTT (under QoS level 3), when the processing thread receives a PUBREC packet, it initiates the transmission of a corresponding PUBREL packet to the broker. Simultaneously, the received packet is logged within the unacked packet list. Upon the arrival of a PUBREC packet, the processing thread undertakes the removal of the associated PUBLISH packet from the unacked packet list. This action is accompanied by the invocation of the *notify()* function, thereby rousing the main thread from its suspended state. Subsequently, upon receipt of a PUBCOMP packet, the processing thread removes the PUBREL packet from the unacked packet list.

Within the asynchronous E-MQTT, the main thread seamlessly progresses to the subsequent operation after transmitting the PUBLISH packet. Upon the processing thread's reception of the PUBREC packet, it promptly dispatches the corresponding PUBREL packet to the broker, which is the same process of MQTT QoS 2 until it receives the PUBCOMP packet.

Illustrated in Figure 10 is the operational sequence of the broker within the context of E-MQTT. Upon receipt of a QoS level 3 PUBLISH packet from the publisher, the broker stores the packet and then proceeds with its customary procedure of transmitting the packet to all subscribers.

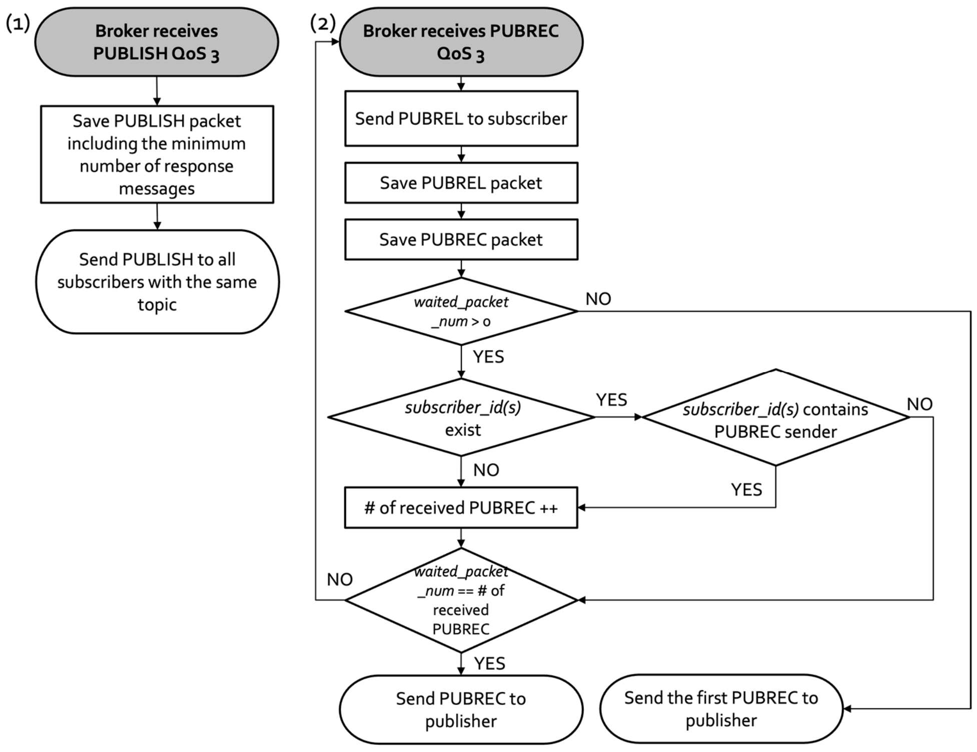

(1)
Broker receives PUBLISH QoS 3
Save PUBLISH packet including the minimum number of response messages
Send PUBLISH to all subscribers with the same topic
(2)
Broker receives PUBREC QoS 3
Send PUBREL to subscriber
Save PUBREL packet
Save PUBREC packet
waited_packet _num > 0
NO
YES
subscriber_id(s) exist
YES
subscriber_id(s) contains PUBREC sender
NO
NO
YES
# of received PUBREC ++
NO
waited_packet _num == # of received PUBREC
YES
Send PUBREC to publisher
Send the first PUBREC to publisher

Figure 10. Flowcharts showing the broker-side execution flow of E-MQTT.

The *waited_packet_num* and *subscriber_id(s)* fields play a pivotal role in the broker's response aggregation process. Upon the broker's reception of a PUBREC packet from a subscriber, it immediately sends a PUBREL packet to the subscriber and stores the PUBREC packet as well as the PUBREL packet. Subsequently, the broker assesses the stored PUBLISH packet. If the *waited_packet_num* field is 1 or more (synchronous E-MQTT), the broker proceeds to examine the *subscriber_id(s)* field. If the field is empty, the broker performs a count of received PUBREC packets. Conversely, if the *subscriber_id(s)* field is not empty, this implies that the publisher has specifically earmarked subscribers whose responses it is awaiting. Consequently, the broker increases the count of received PUBREC packets only if the PUBREC sender is contained in the *subscriber_id(s)* field. The broker then compares the *waited_packet_num* value with the count of PUBREC packets received so far. If the number of received PUBREC packet becomes the same as *waited_packet_num*, the broker transmits an aggregated PUBREC packet to the publisher. However, if the *waited_packet_num* equals 0 (asynchronous E-MQTT), the initial PUBREC packet is promptly transmitted to the publisher upon reception. The subsequent tasks when the broker receives the PUBREL and PUBCOMP packets are the same as that in the existing MQTT.

Figure 11 illustrates the operation sequence of the subscriber within the E-MQTT. The subscriber's functionality is the same as that in MQTT QoS 2 except for the check of the QoS level field. Upon the receipt of the QoS level 3 PUBLISH packet from the broker, the subscriber stores the packet and then sends the PUBREC packet for QoS 3 as a response back to the broker. When the subscriber receives the PUBREL packet, the execution flow is the same as that of MQTT QoS 2.

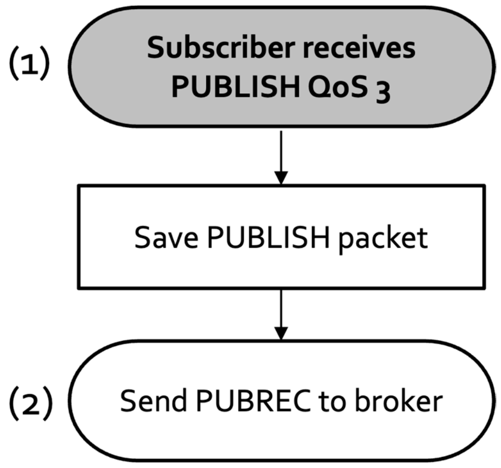


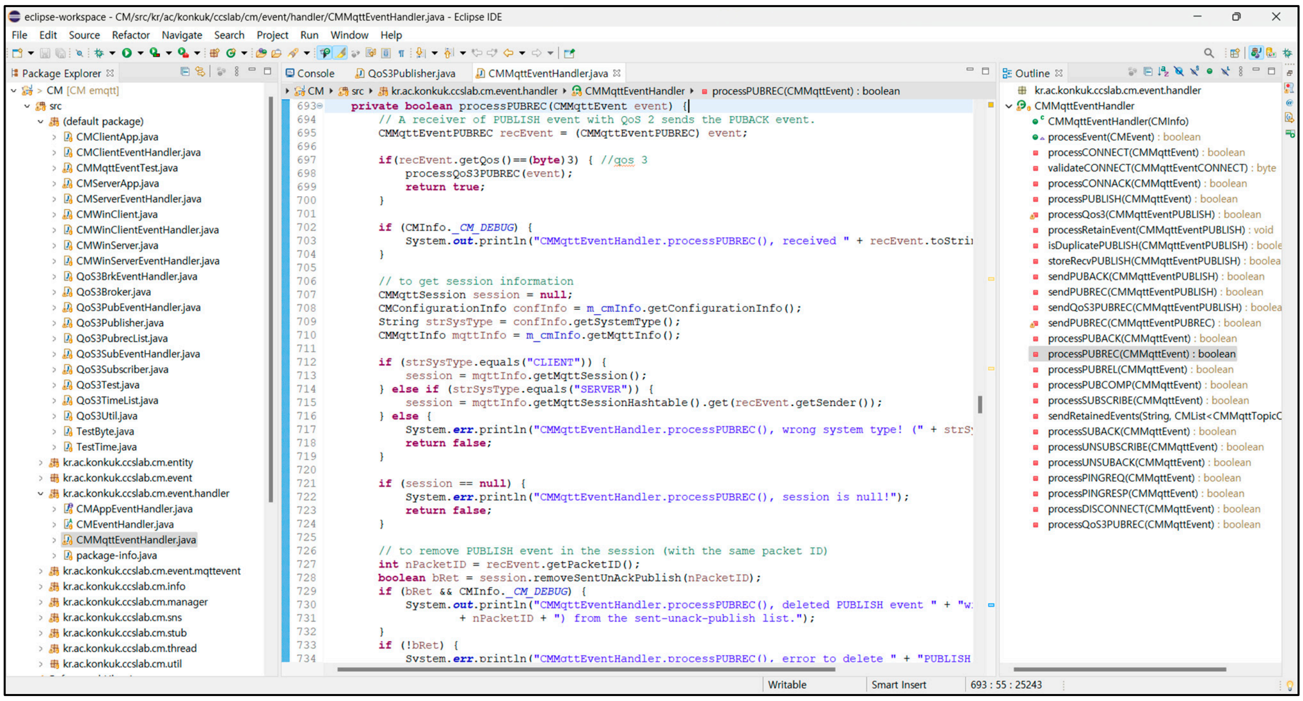
Figure 11. Flowcharts showing the subscriber-side execution flow of E-MQTT.

4. Implementation

The communication framework (CM) [55] serves as an application-level communication framework employed in the development of distributed systems. The CM offers a range of options and communication services that are adaptable to the specific requisites of applications, rendering it more versatile. As a service of the CM, we have implemented E-MQTT, building upon the foundation of MQTT version 3.1.1. E-MQTT has been implemented and assessed utilizing the Java Development Kit (JDK) 20 within the integrated development environment (IDE) Eclipse (2023-03 version). Figure 12 shows a snapshot of the Eclipse IDE housing the E-MQTT implementation. The source codes of the whole CM are available at the Github repository (https://github.com/ccslab/CM.git (accessed on 14 November 2023)) and, specifically, the E-MQTT codes can be found at https://github.com/ccslab/CM/tree/emqtt/CM/src/kr/ac/konkuk/ccslab/cm (accessed on 14 November 2023). To get more information on how to configure and run CM applications that contain E-MQTT functionalities, the quick start guide document can be found at our web page (https://sites.google.com/site/kuccslab/research/cm (accessed on 14 November 2023)).

In both the PUBLISH and SUBSCRIBE packets, a publisher as well as a subscriber possess the capacity to specify QoS 3, thereby establishing a distinction from the existing range of QoS levels spanning 0 through 2. Notably, the QoS field has also been incorporated into the PUBREC, PUBREL, and PUBCOM packets. When a node receives PUBLISH, PUBREC, PUBREL, and PUBCOMP packets, their course of action aligns with the existing MQTT protocol if the QoS level is 2. Conversely, if the QoS level is 3, the proposed E-MQTT is followed.

Figure 13 portrays a class diagram delineating the structure of E-MQTT within the CM. Within this diagram, the principal classes encompass *CMMqttManager* and *CMMqttEventHandler*. *CMMqttManager* is responsible for processing MQTT-related requests originating from a CM client (publisher or subscriber) application. In cases where the publisher requests the PUBLISH service, *CMMqttManager* verifies all the required parameters. The mandatory parameters encompass the topic and the application message, mandating input values from the publisher client. Optional parameters encompass QoS, DUP, and retain flags, along with the minimum number of responses. Should these optional parameters be omitted by the publisher, the default value of 0 is automatically assigned. If the minimum number of response messages is set as equal to or greater than 1, E-MQTT starts the synchronous mode by registering the response (PUBREC) event type in an event synchronization object, of which the class is *CMEventSynchronizer*. The synchronization object is shared by both the *CMMqttManager* and *CMMqttEventHandler* classes to synchronize them. The *CMMqttManager* thread then gets a lock on the synchronization object and waits for the response message by calling the *wait()* method of the synchronization object.


eclipse-workspace - CM/src/kr/ac/konkuk/ccslab/cm/event/handler/CMMqttEventHandler.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
CM [CM emqtt]
src
(default package)
CMClientApp.java
CMClientEventHandler.java
CMMqttEventTest.java
CMServerApp.java
CMServerEventHandler.java
CMWinClient.java
CMWinClientEventHandler.java
CMWinServer.java
CMWinServerEventHandler.java
QoS3BrkEventHandler.java
QoS3Broker.java
QoS3PubEventHandler.java
QoS3Publisher.java
QoS3PubrecList.java
QoS3SubEventHandler.java
QoS3Subscriber.java
QoS3Test.java
QoS3TimeList.java
QoS3Util.java
TestByte.java
TestTime.java
kr.ac.konkuk.ccslab.cm.entity
kr.ac.konkuk.ccslab.cm.event
kr.ac.konkuk.ccslab.cm.event.handler
CMAppEventHandler.java
CMEventHandler.java
CMMqttEventHandler.java
package-info.java
kr.ac.konkuk.ccslab.cm.event.mqttevent
kr.ac.konkuk.ccslab.cm.info
kr.ac.konkuk.ccslab.cm.manager
kr.ac.konkuk.ccslab.cm.sns
kr.ac.konkuk.ccslab.cm.stub
kr.ac.konkuk.ccslab.cm.thread
kr.ac.konkuk.ccslab.cm.util
Console
QoS3Publisher.java
CMMqttEventHandler.java
CM › src › kr.ac.konkuk.ccslab.cm.event.handler › CMMqttEventHandler › processPUBREC(CMMqttEvent) : boolean
Outline
kr.ac.konkuk.ccslab.cm.event.handler
CMMqttEventHandler
CMMqttEventHandler(CMInfo)
processEvent(CMEvent) : boolean
processCONNECT(CMMqttEvent) : boolean
validateCONNECT(CMMqttEventCONNECT) : byte
processCONNACK(CMMqttEvent) : boolean
processPUBLISH(CMMqttEvent) : boolean
processQos3(CMMqttEventPUBLISH) : boolean
processRetainEvent(CMMqttEventPUBLISH) : void
isDuplicatePUBLISH(CMMqttEventPUBLISH) : boole
storeRecvPUBLISH(CMMqttEventPUBLISH) : boolea
sendPUBACK(CMMqttEventPUBLISH) : boolean
sendPUBREC(CMMqttEventPUBLISH) : boolean
sendQoS3PUBREC(CMMqttEventPUBLISH) : boolea
sendPUBREC(CMMqttEventPUBREC) : boolean
processPUBACK(CMMqttEvent) : boolean
processPUBREC(CMMqttEvent) : boolean
processPUBREL(CMMqttEvent) : boolean
processPUBCOMP(CMMqttEvent) : boolean
processSUBSCRIBE(CMMqttEvent) : boolean
sendRetainedEvents(String, CMList<CMMqttTopicC
processSUBACK(CMMqttEvent) : boolean
processUNSUBSCRIBE(CMMqttEvent) : boolean
processUNSUBACK(CMMqttEvent) : boolean
processPINGREQ(CMMqttEvent) : boolean
processPINGRESP(CMMqttEvent) : boolean
processDISCONNECT(CMMqttEvent) : boolean
processQoS3PUBREC(CMMqttEvent) : boolean
Writable
Smart Insert
693 : 55 : 25243

```java
    private boolean processPUBREC(CMMqttEvent event) {
        // A receiver of PUBLISH event with QoS 2 sends the PUBACK event.
        CMMqttEventPUBREC recEvent = (CMMqttEventPUBREC) event;

        if(recEvent.getQos()==(byte)3) { //qos 3
            processQoS3PUBREC(event);
            return true;
        }

        if (CMInfo._CM_DEBUG) {
            System.out.println("CMMqttEventHandler.processPUBREC(), received " + recEvent.toStri

        }

        // to get session information
        CMMqttSession session = null;
        CMConfigurationInfo confInfo = m_cmInfo.getConfigurationInfo();
        String strSysType = confInfo.getSystemType();
        CMMqttInfo mqttInfo = m_cmInfo.getMqttInfo();

        if (strSysType.equals("CLIENT")) {
            session = mqttInfo.getMqttSession();
        } else if (strSysType.equals("SERVER")) {
            session = mqttInfo.getMqttSessionHashtable().get(recEvent.getSender());
        } else {
            System.err.println("CMMqttEventHandler.processPUBREC(), wrong system type! (" + strS
            return false;
        }

        if (session == null) {
            System.err.println("CMMqttEventHandler.processPUBREC(), session is null!");
            return false;
        }

        // to remove PUBLISH event in the session (with the same packet ID)
        int nPacketID = recEvent.getPacketID();
        boolean bRet = session.removeSentUnAckPublish(nPacketID);
        if (bRet && CMInfo._CM_DEBUG) {
            System.out.println("CMMqttEventHandler.processPUBREC(), deleted PUBLISH event " + "w
                    + nPacketID + ") from the sent-unack-publish list.");
        }
        if (!bRet) {
            System.err.println("CMMqttEventHandler.processPUBREC(), error to delete " + "PUBLISH
```

Figure 12. E-MQTT implementation environment.

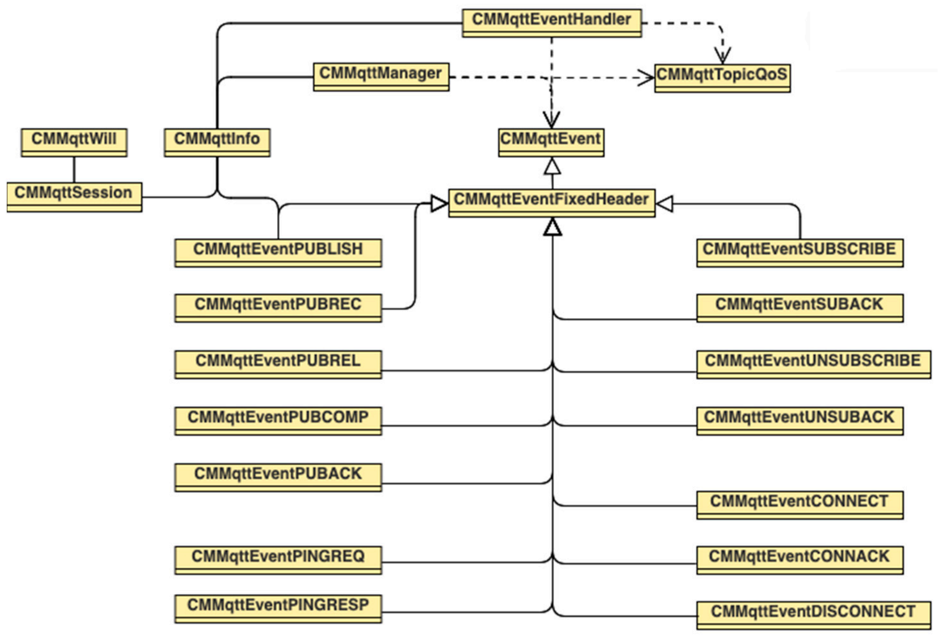


Figure 13. Class diagram of E-MQTT.

The *CMMqttEventHandler* class has a crucial role in the confirmation and processing of incoming MQTT packets. Because the CM uses different threads for the *CMMqttManager* and *CMMqttEventHandler* classes, the *CMMqttEventHandler* class receives messages asynchronously and independently from the task of the *CMMqttManager* class. As this class uses the TCP socket to receive any MQTT packet, the underlying TCP protocol guarantees the confirmation of packet reception. When the *CMMqttEventHandler* class receives an MQTT or E-MQTT packet, it calls the *processEvent()* method, which manages subsequent tasks. This class method distinguishes the packet types, prompting the invocation of the other internal processing methods that correspond to each packet type. In the specific case of PUBLISH packets, the processing method evaluates the associated QoS level. Subsequently, based on the QoS level, an appropriate response packet is transmitted. To elucidate, for QoS 0, no response is issued; for QoS 1, a PUBACK packet is issued; and for QoS 2 and 3, a PUBREC packet is transmitted. When receiving PUBREC, PUBREL, and PUBCOMP packets, the *CMMqttEventHandler* class handles different procedures according to the QoS level, specifically QoS levels 2 and 3. Especially in E-MQTT, when the *CMMqttEventHandler* of the publisher receives a PUBREC packet that is registered in the synchronization object, it recognizes that E-MQTT is operating in the synchronous mode. The *CMMqttEventHandler* then wakes up the *CMMqttManager* class by calling the *notify()* method of the synchronization object so that the *CMMqttManager* class can conduct the next task.

The *CMMqttInfo* class fulfills the role of maintaining all pertinent information necessary for the operation of E-MQTT. Acting as a vital intermediary, the *CMMqttEvent* class serves as a wrapper encompassing an MQTT packet, facilitating its exchange between various CM nodes. As it is an abstract class, *CMMqttEvent* is inherited by the wrapper classes of 14 MQTT packets such as *CMMqttEventPUBLISH*, *CMMqttEventPUBREC*, and so on.

5. Experimental Results

We conducted a series of experiments to analyze the performance of E-MQTT. To facilitate a comprehensive evaluation, we drew a comparative analysis between the proposed E-MQTT QoS 3 and the existing MQTT QoS 2, given their closely aligned transmission processes. We implemented test clients (a publisher and subscribers) alongside broker applications that used both MQTT and E-MQTT because other benchmark frameworks or performance evaluation methods [56–59] do not support the features of E-MQTT. The source codes of the test applications (the broker, publisher, and subscriber) are available at https://github.com/ccslab/CM/tree/emqtt/CM/src (accessed on 14 November 2023). Through the experiments, we measured five performance criteria: end-to-end delay, publish–completion delay, broker–subscriber delay, packet sizes, and energy consumption. They are normal metrics to evaluate the performance of MQTT-related research works [56–59]. Although we analyzed the number of exchanged messages in Section 3.1, the measurement of delay was necessary to figure out the overall overhead incurred by not only the number of messages but also other internal processing costs. Especially for delay, we measured delay values in different periods to analyze the overall cost from different point of views, as shown in Figures 1 and 3. The main purpose of measuring end-to-end delay ((1)~(8) in Figure 1 and (1)~(4) in Figure 3) was to measure total overhead in the end-to-end communication scenario that we addressed, and this metric was most important. We also measured publish–completion delay ((1)~(6) in Figure 1 and (1)~(5) in Figure 3) to compare the overhead of the normal message publication procedure because the E-MQTT broker has additional tasks when it receives PUBLISH and PUBREC packets. The aim of measuring broker–subscriber delay ((3)~(5) in Figure 1 and (2)~(3) in Figure 3) was to compare the response performance of the subscriber, looking for any side effects due to the procedure of E-MQTT. As well as the delay measurement, we also conducted the comparison of packet sizes because E-MQTT surely increases the size of some packets. Furthermore, we measured battery usage of the publisher to compare how much energy is consumed, out of regard for energy-constrained environments like IoT devices.

Our test applications contained all the required codes to measure all the delay metrics and battery usage. To conduct the experiments, we configured the broker address in the publisher and subscriber applications. After starting the broker, we also started the publisher and subscriber. Then, they started MQTT connection with the broker and subscribed to topics required for the experiments. In the publisher application, we could select and start one of experiment menus (end-to-end delay, publish–completion delay, broker–subscriber delay, and battery usage). We could also select one of measurement methods among MQTT, asynchronous E-MQTT, and synchronous E-MQTT for comparison. For the delay measurement, the publisher sent a PUBLISH packet and measured the end-to-end and publish–completion delay values. The broker also measured the broker–subscriber delay value.

5.1. Experimental Environments

In this subsection, we describe environment information of the experiments that we conducted with our test clients and broker applications. The experiments of delay measurement (described in Sections 5.2–5.4) were conducted utilizing a pair of distinct devices. Specifically, the test broker and subscriber applications were executed on a desktop PC, whereas the test publisher application was carried out on a laptop PC. These devices are respectively designated as PC1 (for the desktop) and PC2 (for the laptop). We assumed no network congestion or failure during the experiments. Table 1 shows the detailed device specifications and network configurations of PC1 and PC2.

Table 1. Device specifications for delay measurement.

	PC 1	**PC 2**
Processor	AMD Ryzen 5 3500 6-Core Processor 3.60 GHz	Intel® Core™ i5-8250U CPU @ 1.60 GHz
RAM	16 GB RAM	4 GB RAM
OS	Windows 10	Windows 10
Network	Ethernet (1 Gbps link speed)	Wi-Fi

For the experiments of delay measurement, we started one publisher, one broker, and ten subscriber applications. To measure one delay value, the publisher sent a query message and received a delay value. In the PUBLISH packet of the query message, we set the topic field with a one-byte string and the payload message with six bytes. The other headers of the packet remained empty or default values. In each experiment, we measured delay with sixty tests and obtained an average value. We conducted the same experiment while increasing the number of subscribers by ten until it reached eighty. In synchronous E-MQTT, the number of subscribers corresponded to the minimum number of response packets to be awaited, and the list of subscriber identifiers remained empty.

We set another environment of experiment (described in Section 5.6) because this supplementary experiment was conducted later for the approximate measurement of energy consumption. As a specific IoT environment was not available, we executed the publisher solely on a laptop operating on battery power, while the broker and subscribers were run on a desktop PC. To measure battery usage, the publisher regularly sent a PUBLISH packet every second for 1200 times per experiment. When each experiment started and ended, we recorded the battery level provided by the operating system and obtained the amount of battery usage using the difference between the two values. Table 2 presents the specifications of the devices utilized in this experiment. Apart from the devices, the remaining experimental conditions remained consistent with the previous experiments.

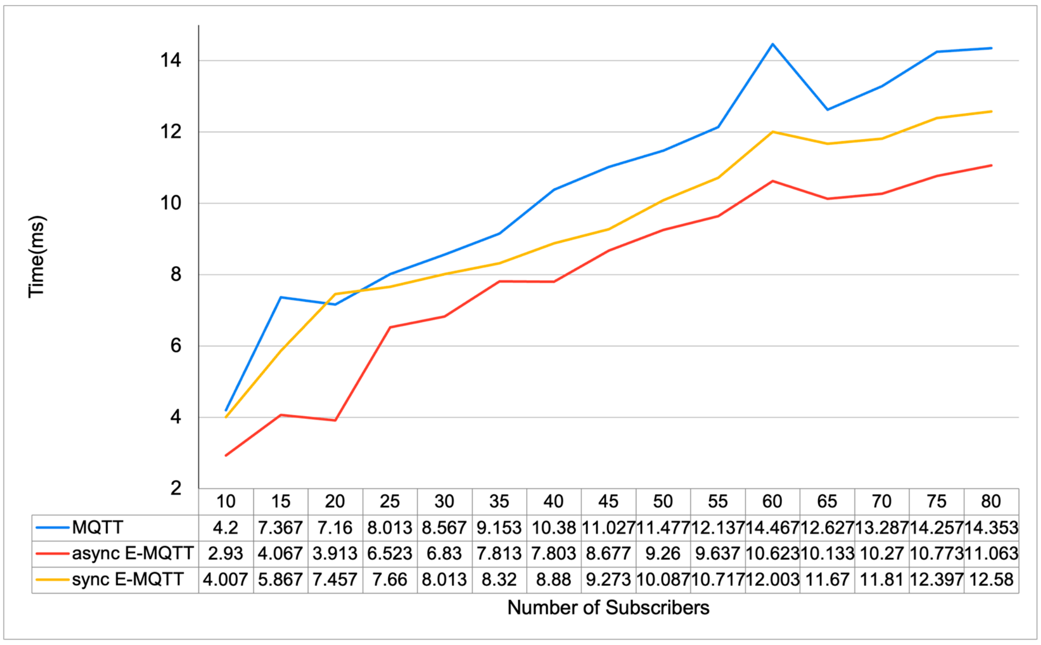
5.2. End-to-End Delay

In the first experiment, we measured response latency within the context of situations wherein the publisher's awareness of message reception by subscribers is essential. In E-MQTT, the subscriber does not need to create a separate PUBLISH packet for a response. As a response packet to PUBLISH, the subscriber transmits the PUBREC packet. However, the existing MQTT requires an additional process to respond to the publisher. The subscriber must respond to the publisher by creating and sending a new PUBLISH packet. To facilitate this, the publisher is mandated to pre-configure subscription to an additional topic (referred to as topic B) enabling the reception of subscriber responses. Subsequently, the subscriber conveys its acknowledgement by publishing a PUBLISH packet with topic B. Upon the receipt of the aforementioned PUBLISH packet with topic B, the publisher effectively confirms successful reception of the original PUBLISH packet by the subscriber.

Table 2. Device specifications for battery usage experiment.

	Publisher	**Broker and Subscribers**
Processor	Apple M2	3.8 GHz Intel i5
RAM	32 GB RAM	16 GB RAM
Power	69.6 Wh Battery	AC

Within the existing MQTT, we defined the end-to-end delay as the elapsed time from the moment the publisher sends a PUBLISH packet to the point when it receives the final response PUBLISH packet sent by the subscribers. In contrast, within the context of the proposed E-MQTT, this end-to-end delay was the elapsed time from the moment the publisher sends a PUBLISH packet to the point when it receives an aggregated PUBREC packet sent by the broker. The first experiment was conducted by increasing the number of subscribers, ranging from 10 to 80. Figure 14 shows the results of the experiment.



	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80
MQTT	4.2	7.367	7.16	8.013	8.567	9.153	10.38	11.027	11.477	12.137	14.467	12.627	13.287	14.257	14.353
async E-MQTT	2.93	4.067	3.913	6.523	6.83	7.813	7.803	8.677	9.26	9.637	10.623	10.133	10.27	10.773	11.063
sync E-MQTT	4.007	5.867	7.457	7.66	8.013	8.32	8.88	9.273	10.087	10.717	12.003	11.67	11.81	12.397	12.58

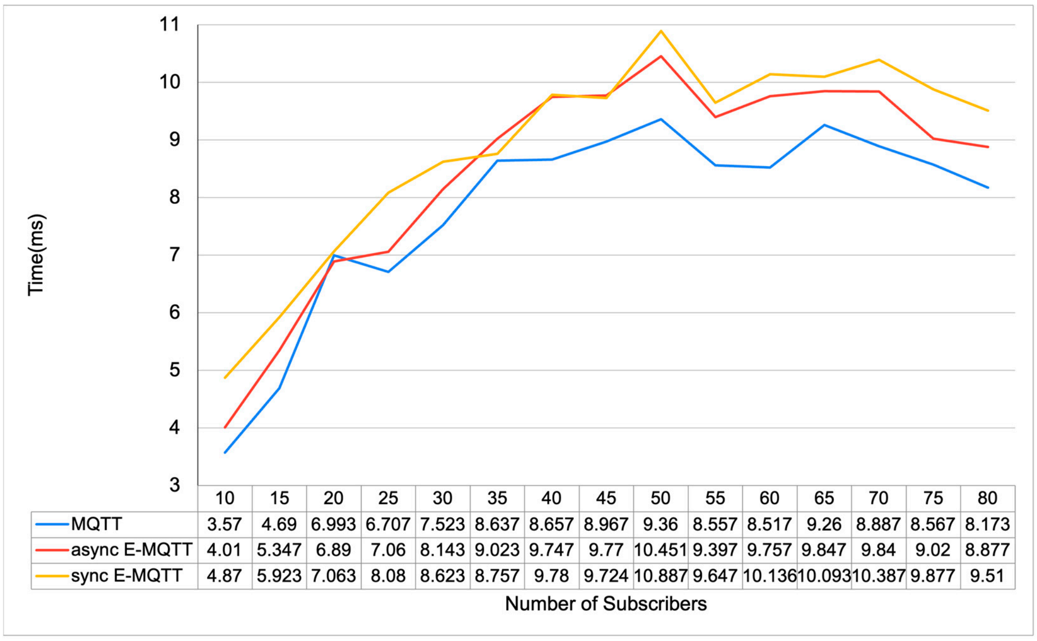
Figure 14. Comparison of end-to-end delay results.

The average end-to-end delay values of MQTT (QoS 2), asynchronous E-MQTT, and synchronous E-MQTT were 9.13, 6.84, and 8.49 milliseconds, respectively. Both synchronous and asynchronous E-MQTT demonstrated notably shorter delays when compared to the conventional MQTT. It is noteworthy that asynchronous E-MQTT consistently outperformed other scenarios across all cases. The synchronous E-MQTT, in contrast, requires an additional synchronization process in the publisher application unlike the asynchronous E-MQTT and conventional MQTT. In terms of message delivery and processing overhead, E-MQTT optimizes the response mechanism within the CM layer that is under the application layer. This differs from MQTT, which carries out the response procedure within the application layer by transmitting an independent PUBLISH packet as the response, which results in supplementary overhead. The broker, in turn, needs to ascertain the topic of the response PUBLISH packet, locate the relevant subscribers (which include the publisher), and send them the packet, which is also an additional overhead of the response mechanism of MQTT. Notably, the quantity of packets traversing between the broker and publisher equates to the number of subscribers. While the proposed E-MQTT involves the supplementary task of collecting response packets from subscribers, the broker can efficiently transmit a single response packet to the publisher.

In summary, the asynchronous E-MQTT showed the shortest end-to-end delay compared to the other two cases, the synchronous E-MQTT and MQTT. This supremacy can be attributed to its omission of a synchronization process and any supplementary response procedures within the application layer. The synchronous E-MQTT has no additional response procedure like the MQTT but needs a certain degree of additional overhead for synchronous communication. With E-MQTT introducing the mechanism of collecting subscriber responses at the broker, the net outcome is a reduction in the number of packets exchanged between the broker and the publisher. Since the time taken for the additional response process of MQTT consistently outweighs other delay factors, the conventional MQTT emerges as the slowest performer across all cases.

5.3. Publish–Completion Delay

In the second experiment, we focused on quantifying the publish–completion delay experienced by the publisher when no response from the subscriber was necessary. This delay is characterized by the duration between the moment the publisher sends a PUBLISH packet and the point at which it receives a corresponding PUBCOMP packet. By progressively increasing the number of subscribers from 10 to 80, we measured the publish–completion delay experienced by the publisher. The experiment's outcomes are visually depicted in Figure 15. The average publish–completion delay values of MQTT, asynchronous E-MQTT, and synchronous E-MQTT were 6.87, 7.53, and 8.20, respectively. Upon reviewing the graph, it is evident that the difference between MQTT and E-MQTT remains minor, manifesting as a mere difference of a few milliseconds. Despite E-MQTT incorporating supplementary logic at the broker's end, it maintains its efficiency and remains on par with MQTT even in cases where no interaction is required.



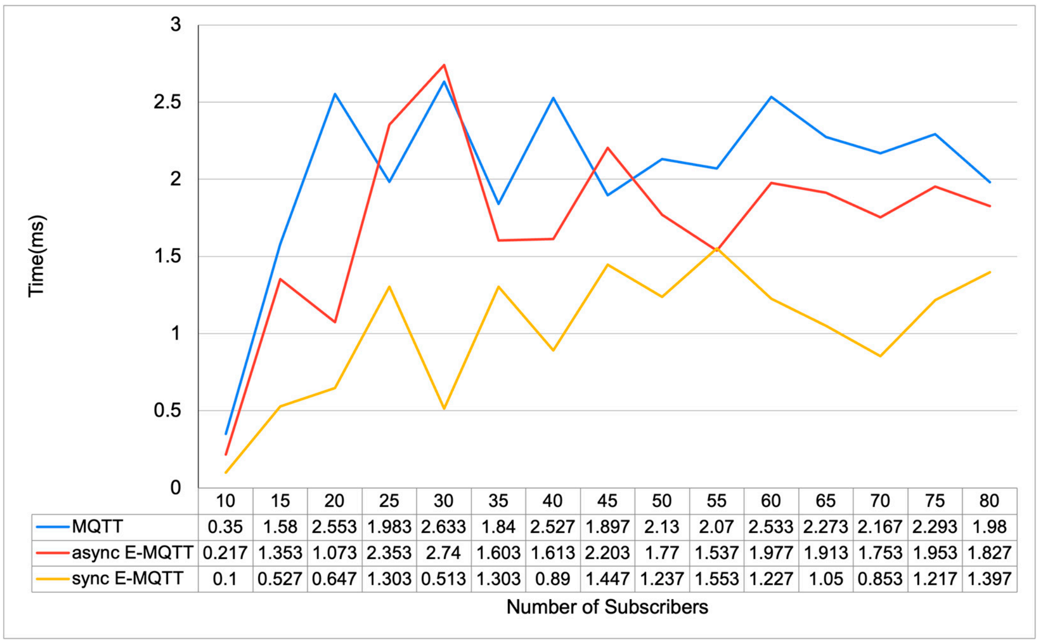
	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80
MQTT	3.57	4.69	6.993	6.707	7.523	8.637	8.657	8.967	9.36	8.557	8.517	9.26	8.887	8.567	8.173
async E-MQTT	4.01	5.347	6.89	7.06	8.143	9.023	9.747	9.77	10.451	9.397	9.757	9.847	9.84	9.02	8.877
sync E-MQTT	4.87	5.923	7.063	8.08	8.623	8.757	9.78	9.724	10.887	9.647	10.136	10.093	10.387	9.877	9.51

Figure 15. Comparison of publish–completion delay results.

5.4. Broker–Subscriber Delay

In the third experiment, we measured the time lapse extending from the moment the broker sends the PUBLISH packet to the point at which it receives the PUBREC packet from the subscriber. This temporal duration was referred to as the broker–subscriber delay. During this investigation, both MQTT and E-MQTT transmitted two packets (PUBLISH and PUBREC) between the broker and the subscriber. However, it is important to note that the broker of E-MQTT incurs an additional computational overhead, attributed to its verification process involving the designated subscriber fields within the PUBLISH packet before sending it to the intended subscribers.

As the number of subscribers increased, we measured the broker–subscriber delay and calculated the average value according to the number of subscribers. As demonstrated in Figure 16, the delay duration experienced by both MQTT and E-MQTT exhibited a comparable pattern. The average broker–subscriber delay values of MQTT, asynchronous E-MQTT, and synchronous E-MQTT were 1.72, 1.48, and 0.85, respectively. Given that both E-MQTT and MQTT transmitted an equal number of packets, discernible dissimilarities in the delay period were not observed. In essence, the message transmission rates of MQTT and E-MQTT were not significantly different.



	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80
MQTT	0.35	1.58	2.553	1.983	2.633	1.84	2.527	1.897	2.13	2.07	2.533	2.273	2.167	2.293	1.98
async E-MQTT	0.217	1.353	1.073	2.353	2.74	1.603	1.613	2.203	1.77	1.537	1.977	1.913	1.753	1.953	1.827
sync E-MQTT	0.1	0.527	0.647	1.303	0.513	1.303	0.89	1.447	1.237	1.553	1.227	1.05	0.853	1.217	1.397

Figure 16. Comparison of broker–subscriber delay results.

5.5. Packet Size

We also measured the size of the transmitted packets (PUBLISH, PUBREC, PUBREL, and PUBCOMP) within the publishing process. As an MQTT packet comprises a fixed header, variable header, and payload, we conducted separate measurements for these three components. Given that our previous experiments involved the encapsulation of an MQTT packet within a CM event, the overall packet size encompassed the CM event header as well. The size of the CM event header remained constant for all types of packets including PUBLISH, PUBREC, PUBREL, and PUBCOMP. When comparing E-MQTT to MQTT, it was observed that E-MQTT led to an increase in the PUBLISH packet size due to the inclusion of a mandatory minimum number of response packets. Moreover, E-MQTT introduces additional fields such as an optional list of designated subscriber IDs. In the case of E-MQTT, the synchronous and asynchronous modes share the same packet format, resulting in uniform packet sizes. Figure 17 illustrates the outcome of our measurement results. Specifically, in the context of the PUBLISH packet, E-MQTT entailed an augmentation of 2 bytes in the variable header to accommodate the minimum number of waiting packets.

However, for the other packet types, including PUBREC, PUBREL, and PUBCOMP, the sizes remained consistent between E-MQTT and MQTT.

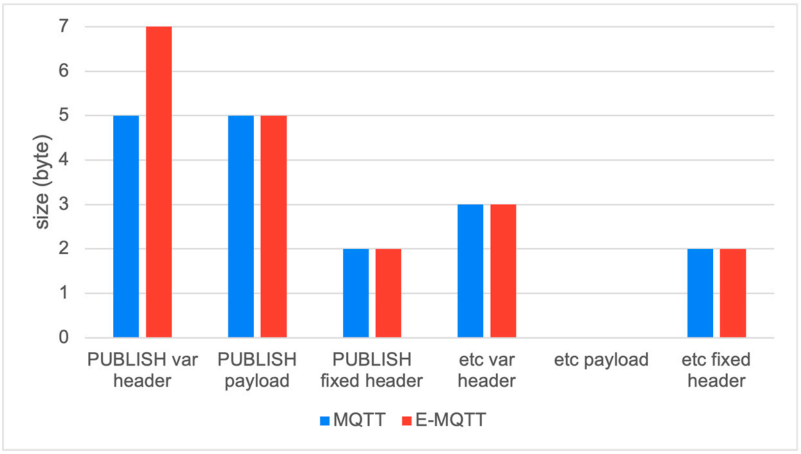


Figure 17. Comparison of packet size results.

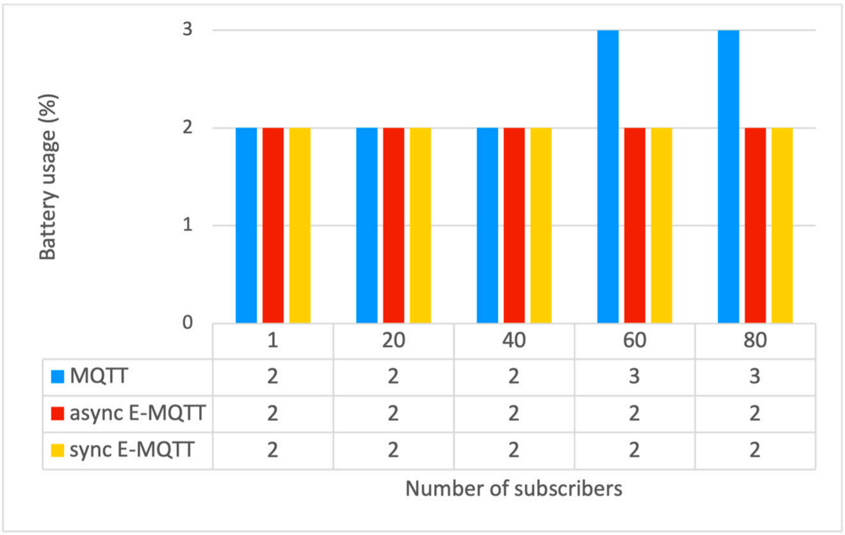
Table 3 provides insight into the packet sizes of the PUBLISH packet in scenarios where the E-MQTT publisher incorporates optional subscriber IDs. Notably, each subscriber ID was represented by a single UTF-16 digit. The table's second column displays the count of designated subscribers. As is inherent to E-MQTT, the size of its PUBLISH packet consistently exceeded that of MQTT by 2 bytes. This discrepancy arises from the inclusion of the number of pre-specified waiting packets (*waited_packet_num*) within E-MQTT's PUBLISH packet. As the PUBLISH packet of E-MQTT stores a list of subscribers, the packet size increases whenever the publisher adds a subscriber ID. As such, it is prudent for the E-MQTT publisher to exercise caution with regard to an excessive number of subscribers, as this could lead to a significant enlargement in packet size.

Table 3. PUBLISH packet sizes according to the number of subscriber IDs.

QoS	Number of Subscriber IDs	PUBLISH (Bytes)
2 (MQTT)	N/A	12
3 (E-MQTT)	0	14
	1	16
	2	18
	3	20
	4	22
	5	24
	10	34
	20	54
	40	94
	80	174

5.6. *Energy Consumption*

Finally, we conducted measurements on the battery usage of the publisher to estimate the energy consumption of MQTT and E-MQTT. Figure 18 illustrates the battery usage results, indicating that the overall battery usage was nearly identical for MQTT and E-MQTT. However, when the number of subscribers exceeded 60, the MQTT publisher experienced a slight increase in battery consumption (3%) due to the higher number of packet exchanges, as discussed in Section 3.1. As the battery level information provided by the operating system was in units of 1%, we could not measure more accurate values in floating point numbers. However, we could estimate that a large number of packets also affected energy consumption. On the other hand, the battery usage of the E-MQTT publisher remained consistent (2%) since it exchanges the same number of packets regardless of the number of subscribers. Notably, both synchronous and asynchronous E-MQTT demonstrated equivalent battery usage. This implies that the additional overhead involved in the synchronization management of the synchronous E-MQTT has minimal impact on energy consumption.



	1	20	40	60	80
🟦 MQTT	2	2	2	3	3
🟥 async E-MQTT	2	2	2	2	2
🟨 sync E-MQTT	2	2	2	2	2

Figure 18. Comparison of battery usage of publisher results.

6. Conclusions

In this paper, we proposed E-MQTT, a novel mechanism addressing the end-to-end communication problem inherent in MQTT. From the experimental results, although E-MQTT has a slightly larger packet size than MQTT, its distinctive response simplification strategy and reduced number of exchanged packets contributed to shorter end-to-end delay and less energy consumption. The advantages gained from these performance improvements far outweigh the minor difference in packet size.

The current E-MQTT still has limitations and needs to improve further. Firstly, we developed E-MQTT based on MQTT version 3.1.1, which is not the most recent one. Secondly, although we combined the publish–subscribe model with the request–response pattern, the response message is just a simple acknowledgement without carrying additional information. Lastly, E-MQTT is an improvement of MQTT QoS 2 but not the other QoS levels (0 and 1). As future research, we plan to resolve the current limitations of E-MQTT. We will redesign E-MQTT according to the specifications of MQTT version 5.0, which is the latest one to accommodate new features. Using the new features of MQTT version 5.0, we will extend the combination of the publish–subscribe model and request–response patterns so that the response message can contain additional information. Furthermore, we will improve E-MQTT such that it can improve the other QoS levels 0 and 1 of MQTT in the request–response pattern communication.

Author Contributions: Conceptualization, M.L.; methodology, M.L. and Y.I.; software, Y.I. and M.L.; validation, Y.I. and M.L.; formal analysis, Y.I.; investigation, Y.I. and M.L.; resources, M.L.; data curation, Y.I.; writing—original draft preparation, Y.I.; writing—review and editing, M.L.; visualization, Y.I.; supervision, M.L.; project administration, M.L.; funding acquisition, M.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by a National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1F1A1047032).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to private experimental environment.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. *ISO/IEC 20922: 2016*; MQTT Version 3.1.1. ISO: Geneva, Switzerland, 2016. Available online: https://www.iso.org/standard/69466.html (accessed on 31 July 2023).
2. Hunkeler, U.; Truong, H.L. MQTT-S—A Publish/Subscribe Protocol for Wireless Sensor Networks. In Proceedings of the International Conference on Communication Systems Software and Middleware and Workshops, Bangalore, India, 6–10 January 2008. [CrossRef]
3. Kodali, R.K.; Soratkal, S. MQTT Based Home Automation System Using ESP8266. In Proceedings of the IEEE Region 10 Humanitarian Technology Conference (R10-HTC), Agra, India, 21–23 December 2016. [CrossRef]
4. Bryce, R.; Shaw, T.; Srivastava, G. MQTT-G: A Publish/Subscribe Protocol with Geolocation. In Proceedings of the International Conference on Telecommunications and Signal Processing, Athens, Greece, 4–6 July 2018. [CrossRef]
5. Chooruang, K.; Mangkalakeeree, P. Wireless Heart Rate Monitoring System Using MQTT. *Procedia Comput. Sci.* **2016**, *86*, 160–163. [CrossRef]
6. Grgić, K.; Špeh, I.; Heđi, I. A Web-Based IoT Solution for Monitoring Data Using MQTT Protocol. In Proceedings of the International Conference on Smart Systems and Technologies (SST), Osijek, Croatia, 12–14 October 2016.
7. Zhou, J.; De Roure, D. Floodnet: Coupling Adaptive Sampling with Energy Aware Routing in a Flood Warning System. *J. Comput. Sci. Technol.* **2007**, *22*, 121–130. [CrossRef]
8. De Roure, D.; Hutton, C.; Cruickshank, D.; Kuan, E.L.; Neal, J.; Roddis, R.; Stanford-Clark, A.; Vivekanandan, S.; Zhou, J. Floodnet–Improving Flood Warning Times Using Pervasive and Grid Computing. 2005. Available online: https://www.researchgate.net/publication/238669079_FloodNet_-_Improving_Flood_Warning_Times_using_Pervasive_and_Grid_Computing (accessed on 14 November 2023).
9. Mishra, T.; Garg, D.; Madhav, G. A Publish/Subscribe Communication Infrastructure for VANET Applications. In Proceedings of the IEEE Workshop of International Conference on Advanced Information Networking and Applications, Biopolis, Singapore, 22–25 March 2011. [CrossRef]
10. Nunes, P.; Nicolau, C.; Santos, J.P.; Completo, A. From a Traditional Bicycle to a Mobile Sensor in the Cities. In Proceedings of the 6th International Conference on Vehicle Technology and Intelligent Transport Systems, Online, 2–4 May 2020. [CrossRef]
11. Chodorek, A.; Chodorek, R.R.; Sitek, P. UAV-Based and WebRTC-Based Open Universal Framework to Monitor Urban and Industrial Areas. *Sensors* **2021**, *21*, 4061. [CrossRef] [PubMed]
12. Shinde, S.A.; Nimkar, P.A.; Singh, S.P.; Salpe, V.D.; Jadhav, Y.R. MQTT—Message Queuing Telemetry Transport Protocol. *Int. J. Res.* **2016**, *3*, 240–244.
13. Soni, D.; Makwana, A. A Survey on MQTT: A Protocol of Internet of Things (IoT). In Proceedings of the International Conference on Telecommunication, Power Analysis and Computing Techniques, Chennai, India, 6–8 April 2017.
14. Kim, S.J.; Oh, C.H. Method for Message Processing According to Priority in MQTT Broker. *J. Korea Inst. Inf. Commun. Eng.* **2017**, *21*, 1320–1326. [CrossRef]
15. Ali, J.; Zafar, M.H.; Hewage, C.; Hassan, R.; Asif, R. Mathematical Modeling and Validation of Retransmission-Based Mutant MQTT for Improving Quality of Service in Developing Smart Cities. *Sensors* **2022**, *22*, 9751. [CrossRef]
16. Palmese, F.; Redondi, A.E.C.; Cesana, M. Adaptive Quality of Service Control for MQTT-SN. *Sensors* **2022**, *22*, 8852. [CrossRef]
17. Singh, M.; Rajan, M.A.; Shivraj, V.L.; Balamuralidhar, P. Secure MQTT for Internet of Things (IoT). In Proceedings of the Fifth International Conference on Communication Systems and Network Technologies, Gwalior, India, 4–6 April 2015.
18. Chandramouli, B.; Yang, J. End-to-End Support for Joins in Large-Scale Publish/Subscribe Systems. *Proc. VLDB Endow.* **2008**, *1*, 434–450. [CrossRef]
19. Pallickara, S.; Pierce, M.; Gadgil, H.; Fox, G.; Yan, Y.; Huang, Y. A Framework for Secure End-to-End Delivery of Messages in Publish/Subscribe Systems. In Proceedings of the 7th IEEE/ACM International Conference on Grid Computing, Barcelona, Spain, 28–29 September 2006. [CrossRef]
20. Zhang, H.; Zhang, H.; Wang, Z.; Zhou, Z.; Wang, Q.; Xu, G.; Yang, J.; Gan, Z. Delay-Reliability-Aware Protocol Adaption and Quality of Service Guarantee for Message Queuing Telemetry Transport-Empowered Electric Internet of Things. *Int. J. Distrib. Sens. Netw.* **2022**, *18*, 1–11. [CrossRef]
21. Rocha, H.d.; Monteiro, T.L.; Pellenz, M.E.; Penna, M.C.; Alves Junior, J. An MQTT-SN-based QoS Dynamic Adaptation Method for Wireless Sensor Networks. In Proceedings of the International Conference on Advanced Information Networking and Applications, Matsue, Japan, 27–29 March 2019. [CrossRef]
22. Alshammari, H.H. The Internet of Things Healthcare Monitoring System Based on MQTT Protocol. *Alex. Eng. J.* **2023**, *69*, 275–287. [CrossRef]

23. Thangavel, D.; Ma, X.; Valera, A.; Tan, H.X.; Tan, C.K.Y. Performance Evaluation of MQTT and CoAP via a Common Middleware. In Proceedings of the IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing, Singapore, 21–24 April 2014. [CrossRef]
24. Naik, N. Choice of Effective Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP and HTTP. In Proceedings of the IEEE International Systems Engineering Symposium (ISSE), Vienna, Austria, 11–13 October 2017. [CrossRef]
25. Collina, M.; Corazza, G.E.; Vanelli-Coralli, A. Introducing the QEST Broker: Scaling the IoT by Bridging MQTT and REST. In Proceedings of the IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC), Sydney, NSW, Australia, 9–12 September 2012.
26. Silva, D.; Carvalho, L.I.; Soares, J.; Sofia, R.C. A Performance Analysis of Internet of Things Networking Protocols: Evaluating MQTT, CoAP, OPC UA. *Appl. Sci.* **2021**, *11*, 4879. [CrossRef]
27. Uy, N.Q.; Nam, V.H. A Comparison of AMQP and MQTT Protocols for Internet of Things. In Proceedings of the 2019 6th NAFOSTED Conference on Information and Computer Science (NICS), Hanoi, Vietnam, 12–13 December 2019. [CrossRef]
28. Gemirter, C.B.; Senturca, Ç.; Baydere, Ş. A Comparative Evaluation of AMQP, MQTT and HTTP Protocols Using Real-Time Public Smart City Data. In Proceedings of the 2021 6th International Conference on Computer Science and Engineering (UBMK), Ankara, Turkey, 15–17 September 2021. [CrossRef]
29. Naik, G.; Bapat, A. A Brief Comparative Analysis on Application Layer Protocols of Internet of Things: MQTT, CoAP, AMQP and HTTP. *Int. J. Comput. Sci. Mob. Computing.* **2020**, *9*, 135–141. [CrossRef]
30. Palmese, F.; Longo, E.; Redondi, A.E.; Cesana, M. CoAP vs. MQTT-SN: Comparison and Performance Evaluation in Publish-Subscribe Environments. In Proceedings of the 2021 IEEE 7th World Forum on Internet of Things (WF-IoT), New Orleans, LA, USA, 14 June–31 July 2021. [CrossRef]
31. Eclipse Paho Project. Available online: https://www.eclipse.org/paho/ (accessed on 31 July 2023).
32. Advanced Message Queuing Protocol (AMQP). Available online: https://www.amqp.org (accessed on 31 July 2023).
33. RabbitMQ. Available online: https://www.rabbitmq.com (accessed on 31 July 2023).
34. Apache Kafka. Available online: https://kafka.apache.org (accessed on 31 July 2023).
35. Sen, S.; Balasubramanian, A. A Highly Resilient and Scalable Broker Architecture for IoT Applications. In Proceedings of the 10th International Conference on Communication Systems & Networks (COMSNETS), Bengaluru, India, 3–7 January 2018. [CrossRef]
36. Bagaskara, A.E.; Setyorini, S.; Wardana, A.A. Performance Analysis of Message Broker for Communication in Fog Computing. In Proceedings of the 12th International Conference on Information Technology and Electrical Engineering (ICITEE), Yogyakarta, Indonesia, 6–8 October 2020. [CrossRef]
37. Pratama, H.P.; Prihatmanto, A.S.; Sukoco, A. Implementation Messaging Broker Middleware for Architecture of Public Transportation Monitoring System. In Proceedings of the 6th International Conference on Interactive Digital Media (ICIDM), Bandung, Indonesia, 14–15 December 2020. [CrossRef]
38. Shafabakhsh, B.; Lagerström, R.; Hacks, S. Evaluating the Impact of Inter Process Communication in Microservice Architectures. In Proceedings of the 8th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2020), Singapore, 1 December 2020.
39. Jaloudi, S. Communication Protocols of an Industrial Internet of Things Environment: A Comparative Study. *Future Internet* **2019**, *11*, 66. [CrossRef]
40. Kul, S.; Tashiev, I.; Şentaş, A.; Sayar, A. Event-Based Microservices with Apache Kafka Streams: A Real-Time Vehicle Detection System Based on Type, Color, and Speed Attributes. *IEEE Access* **2021**, *9*, 83137–83148. [CrossRef]
41. Hamad, M.; Finkenzeller, A.; Liu, H.; Lauinger, J.; Prevelakis, V.; Steinhorst, S. SEEMQTT: Secure End-to-End MQTT-Based Communication for Mobile IoT Systems Using Secret Sharing and Trust Delegation. *IEEE Internet Things J.* **2022**, *10*, 3384–3406. [CrossRef]
42. Park, C.; Nam, H. Security Architecture and Protocols for Secure MQTT-SN. *IEEE Access* **2020**, *8*, 226422–226436. [CrossRef]
43. Spina, M.G.; De Rango, F.; Marotta, G.M. Lightweight Dynamic Topic-Centric End-to-End Security Mechanism for MQTT. In Proceedings of the IEEE/ACM 25th International Symposium on Distributed Simulation and Real Time Applications, Valencia, Spain, 27–29 September 2021. [CrossRef]
44. Winarno, A.; Sari, R.F. A Novel Secure End-to-End IoT Communication Scheme Using Lightweight Cryptography Based on Block Cipher. *Appl. Sci.* **2022**, *12*, 8817. [CrossRef]
45. Chien, H.Y.; Wang, N.Z. A Novel MQTT 5.0-Based Over-the-Air Updating Architecture Facilitating Stronger Security. *Electronics* **2022**, *11*, 3899. [CrossRef]
46. Bashir, A.; Mir, A.H. Lightweight Secure MQTT for Mobility Enabled e-health Internet of Things. *Int. Arab. J. Inf. Technol.* **2021**, *18*, 773–781. [CrossRef]
47. Govindan, K.; Azad, A.P. End-to-End Service Assurance in IoT MQTT-SN. In Proceedings of the 12th Annual IEEE Consumer Communications and Networking Conference, Las Vegas, NV, USA, 9–12 January 2015. [CrossRef]
48. D'Ortona, C.; Tarchi, D.; Raffaelli, C. Open-Source MQTT-Based End-to-End IoT System for Smart City Scenarios. *Future Internet* **2022**, *14*, 57. [CrossRef]
49. Ali, J.; Zafar, M.H. Improved End-to-End Service Assurance and Mathematical Modeling of Message Queuing Telemetry Transport Protocol Based Massively Deployed Fully Functional Devices in Smart Cities. *Alex. Eng. J.* **2023**, *72*, 657–672. [CrossRef]

50. Jo, H.C.; Jin, H.W.; Kim, J. Self-Adaptive End-to-End Resource Management for Real-Time Monitoring in Cyber-Physical Systems. *Comput. Netw.* **2023**, *225*, 109669. [CrossRef]
51. Tanenbaum, A.S.; Steen, M.V. *Distributed Systems: Principles and Paradigms*, 2nd ed.; Pearson Prentice Hall: Old Bridge, NJ, USA, 2007; pp. 35–37+303.
52. Lee, S.; Kim, H.; Hong, D.K.; Ju, H. Correlation Analysis of MQTT Loss and Delay According to QoS Level. In Proceedings of the International Conference on Information Networking, Bangkok, Thailand, 28–30 January 2013. [CrossRef]
53. Banks, A.; Briggs, E.; Borgendale, K.; Gupta, R. MQTT Version 5.0. OASIS Standard. Available online: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html (accessed on 7 November 2023).
54. Gomes, Y.F.; Santos, D.F.; Almeida, H.O.; Perkusich, A. Integrating MQTT and ISO/IEEE 11073 for Health Information Sharing in the Internet of Things. In Proceedings of the IEEE International Conference on Consumer Electronics, Las Vegas, NV, USA, 9–12 January 2015. [CrossRef]
55. Lim, M. Directly and Indirectly Synchronous Communication Mechanisms for Client-Server Systems Using Event Based Asynchronous Communication Framework. *IEEE Access* **2019**, *7*, 81969–81982. [CrossRef]
56. Roy, D.G.; Mahato, B.; De, D.; Buyya, R. Application-Aware End-to-End Delay and Message Loss Estimation in Internet of Things (IoT)—MQTT-SN protocols. *Future Gener. Comput. Syst.* **2018**, *89*, 300–316. [CrossRef]
57. Longo, E.; Redondi, A.E.C.; Cesana, M.; Manzoni, P. BORDER: A Benchmarking Framework for Distributed MQTT Brokers. *IEEE Internet Things J.* **2022**, *9*, 17728–17740. [CrossRef]
58. Ferraz, N.; Silva, A.A.A.; Guelfi, A.E.; Kofuji, S.T. Performance Evaluation of Publish-Subscribe Systems in IoT Using Energy-Efficient and Context-Aware Secure Messages. *J. Cloud Comput.* **2022**, *11*, 6. [CrossRef] [PubMed]
59. Mishra, B.; Mishra, B.; Kertesz, A. Stress-Testing MQTT Brokers: A Comparative Analysis of Performance Measurements. *Energies* **2021**, *14*, 5817. [CrossRef]