Designing a broker extension for seamless CoAP and MQTT interoperability

Corrado Innamorati
DEIB
Politecnico di Milano
Milan, Italy
corrado.innamorati@polimi.it

Alessandro Enrico Cesare Redondi
DEIB
Politecnico di Milano
Milan, Italy
alessandroenrico.redondi@polimi.it

Matteo Cesana
DEIB
Politecnico di Milano
Milan, Italy
matteo.cesana@polimi.it

***Abstract*—The Internet of Things (IoT) is a transformative paradigm facilitating connectivity among everyday objects and devices via the internet, simplifying industrial processes, and enhancing quality of life for individuals, while also fostering the development of innovative applications. However, the creation of an interconnected network of heterogeneous "things" presents unique adaptation challenges in hardware and software characteristics, notably communication protocols, leading to major interoperability issues. This work addresses these challenges by proposing a solution for seamless integration of two widely adopted protocols, MQTT and CoAP, through the development of an interoperability broker extension. By effectively mapping REST and publish/subscribe mechanisms, this broker extension enables secure and seamless interaction between heterogeneous clients, utilizing a single broker structure and eliminating the need for additional middleware. Comprehensive testing demonstrates comparable performance in latency and memory resource consumption, validating the system as a promising solution to resolve IoT interoperability issues.**

***Index Terms*—Internet of Things, IoT, Interoperability, MQTT, CoAP, Broker extension**

I. Introduction

The Internet of Things (IoT) represents a revolutionary paradigm in the world of technology and connectivity, allowing for ubiquitous connection of devices. At its core, IoT empowers "things" with internet connectivity, enabling them to gather, exchange, and act upon real-time data. These "smart objects" [1] span over a vast spectrum, ranging from household appliances and vehicles to industrial machinery and healthcare sensors, thereby expanding the capabilities of IoT into a series of sub-sections, such as Industrial IoT (IIOT) or Smart Cities. The impact of this technology is profound: Huawei forecasts 100 billion devices to be connected by 2025 [2], with a potential economic impact ranging from 3.9 to 11 trillion annually in the same year [3]. The rapid proliferation of the IoT market and its diverse applications have led to a scenario where each device requires different characteristics to best adapt to its working environment. This diversity has resulted in a wide heterogeneity of designs and communication capabilities, creating a fragmented landscape where the seamless connectivity promised by IoT paradigms often loses its efficacy [4]. This heterogeneity is mirrored at the application layer, where a multitude of protocols compose the picture (e.g., HTTP, CoAP, MQTT, AMP) [5]. While these protocols are crafted to navigate the constrained IoT environment, they operate on different and non-standardized principles, complicating communication between devices and exacerbating interoperability challenges. As a result, interoperability emerges as one of the most limiting and weakening aspects of IoT development [6]. Ensuring smooth connectivity between IoT devices is not only crucial for enhancing their efficiency, but also for unlocking the full potential of IoT applications across various domains. Interoperability goes beyond simple inter-device connection; it involves empowering them to collaborate seamlessly, regardless of their inherent protocols or communication standards. This is precisely where the suggested bridging solution, centered on MQTT and CoAP protocols, plays a pivotal role.

The proposed solution offers a practical means to overcome interoperability challenges and streamline communication between the two protocols by seamlessly integrating MQTT and CoAP protocols within a unified system, facilitated by a MQTT broker. Without relying on middleware or complex intermediary layers, the presented architecture extends MQTT broker functions into the CoAP environment, ensuring a cohesive function between the two protocols.

This paper is structured as follows: Section II reviews existing work on interoperability solutions for CoAP and MQTT. In Section III, we provide a technical background to introduce the concepts necessary for understanding our approach. Section IV details our extended broker, including descriptions of the mapping implementations and overall functionalities offered. Next, in Section V, we present and evaluate the experimental results obtained in our testing scenario. Finally, we conclude the work in Section VI with reflections on our findings and directions for future research.

II. Related works

In the landscape of IoT interoperability solutions, various approaches have been explored to tackle the challenges posed by diverse protocols and fragmented communication standards. An accurate overview of the main works in the literature is reported in [7], where authors present a comprehensive survey on Industrial Internet of Things (IIoT) application layer protocols interoperability works conducted between 2014-2020. Specifically, different integration approaches were

identified, categorized as Gateway, Middleware, Broker, and more specific approaches such as SDN [11]. In [8], Collina et al. introduce the QEST broker, a solution to connect MQTT and REST and improve scalability and interoperability in IoT. This work entails building an ad-hoc broker with MQTT and REST server components, utilizing a Redis database for data storage. The project later evolved into the Eclipse Ponte project [10], which further refined broker capabilities by expanding support for the CoAP protocol, introducing pub/sub logic, and enhancing support for multiple data storage units. However, the project is currently archived and has not received recent updates. In [9], Dave et al. propose a middleware MQTT-CoAP interconnector (MCI) that actively operates between a CoAP server and MQTT broker, providing data parsing and fast translation operations. Palavras et al. introduced the Secure Multi-protocol Integration Bridge for IoT (SeMIBIoT) [12], a gateway that enables secure communications between heterogeneous clients, featuring real-time translation of messages across different protocols including HTTP, XMPP, MQTT, and CoAP, albeit limited to specific actions or requests. In [13], the IOTM^{2}B introduces a multi-protocol strategy where protocols such as MQTT, CoAP, and HTTP communicate with an external target system (e.g., Cloud) through REST messages, facilitated by the system's multi-protocol functionalities. While existing works have made significant strides in addressing interoperability issues, each solution may have its limitations or trade-offs, ranging from system adaptability to ease of implementation or additional infrastructure requirements.

III. Technical Background

A. MQTT

MQTT is a lightweight communication protocol designed for constrained environments [14]. Initially developed by IBM, it has now become an open source OASIS standard protocol. Operating on a broker-based architecture, MQTT employs a publish/subscribe logic. Clients connect to a central MQTT broker to publish new messages on topics or subscribe to them for real-time updates. This decoupled architecture enables scalable and asynchronous communication between devices. The MQTT broker plays a fundamental role in managing, storing, and routing messages, relieving devices of heavy computational and energy use. Messages can be stored on topics by using the retain flag: a publish message with the retain flag set will be maintained in the broker's memory and possibly transmitted to clients who subscribe after publications have been performed. MQTT typically operates over TCP/IP, supporting its security mechanisms, and introduces Quality of Service (QoS) levels to allow clients to specify the reliability of messages. MQTT's widespread usage is supported by a broad range of programming languages like Java, Python and C. For this work, the HiveMQ CE broker [15] was selected as the basis, offering a Java-based open source MQTT broker with the flexibility to implement custom extensions.

B. CoAP

CoAP, or Constrained Application Protocol, is a standardized communication protocol designed specifically for the Internet of Things (IoT) ecosystem [16]. Developed by the Internet Engineering Task Force (IETF), CoAP follows a RESTful architecture, mirroring HTTP principles. Operating on a request-response model, CoAP messages are typically carried over UDP for efficiency. It distinguishes between Confirmable and Non-Confirmable requests for reliability. CoAP messages support various operations, including GET, POST, PUT, and DELETE, allowing for CRUD (Create, Read, Update, Delete) operations on IoT resources. CoAP introduces support for asynchronous communication through the observe mechanism, allowing devices to subscribe to resource changes and receive notifications of modifications without continuous polling. In this work, CoAP implementations are based on the Eclipse Californiun framework [17], a lightweight Java implementation of the CoAP protocol. Leveraging this framework enables the utilization of CoAP's basic functions in a simple and efficient manner. Additionally, supplementary sub-modules such as Scandium are incorporated to enhance the functionality of CoAP implementations, providing additional features and capabilities.

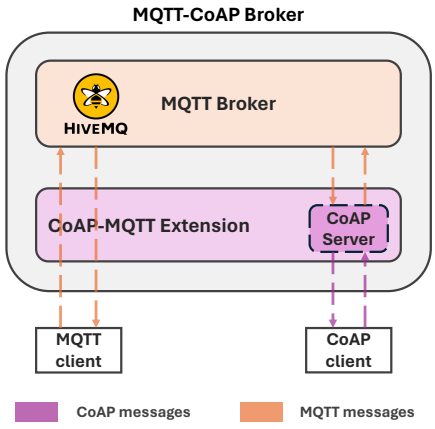

Fig. 1. General Broker structure.

IV. The MQTT-CoAP Broker extension

The solution discussed in this paper extends the functionality of the HiveMQ MQTT broker, bridging between the differing communication models adopted by MQTT and CoAP: pub/sub and requests/response. In the proposed architecture, summarized in Figure 1, the native pub/sub logic of MQTT is maintained, while a specific CoAP-MQTT Interoperability extension is designed to integrate CoAP requests into the system, facilitating two-way seamless connectivity between CoAP and MQTT clients.

A. Mapping into topics

To maintain the fundamental logic of the MQTT broker, topics are leveraged as the primary mechanism for communication. In contrast, CoAP relies on resources and URIs for addressing. To bridge this gap, CoAP resources are transformed into MQTT topics by mapping URI path schemes into topic names. A singular CoAP resource, called "**mqtt**", serves as the entry point for CoAP requests, effectively acting as a gateway to the MQTT broker. CoAP clients access the broker by specifying the "$mqtt$" resource in their URI, along with the Broker IP address. Subsequent URI path levels are treated as part of the MQTT topic hierarchy. To adhere to CoAP logic, the broker makes CoAP ports 5683/5684 available for communication.
For instance, as illustrated in Figure 2, a CoAP client can send a POST request to the "$mqtt/room/temp$" resource path, thereby creating a "$room/temp$" topic accessible to MQTT clients as well. Conversely, if an MQTT client publishes on the topic "$room/light$", the CoAP client can access the "$mqtt/room/light$" URI path to retrieve the content.

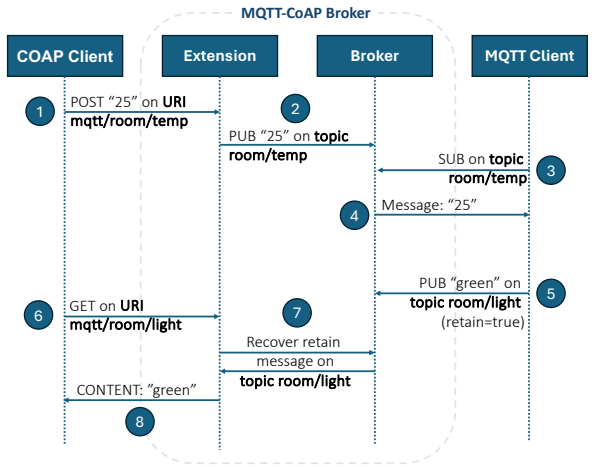

Fig. 2. URI-topic mapping example.

B. Resource Discovery

In contrast with MQTT, which is inherently stateless, CoAP supports resource discovery mechanisms through GET requests to the **"/.well-known/core"** URI path. Any client request to such a URI triggers the server to reply with a list of active resources that can be queried subsequently. To bridge CoAP and MQTT, it is necessary to create a mechanism to inform a client about the currently active MQTT topics. For this reason, the proposed extension maintains a *Topic Map* which contains MQTT topics considered active resources, that can be directly accessed with COAP methods. Such topics are (i) topics which have an associated retained message or (ii) topics for which an update event (either MQTT PUB or COAP POST/PUT) is observed within a specific time threshold T. Therefore, topics for which no messages are observed for more than T seconds are considered inactive and are removed from the *Topic Map*. Any PUT/GET/DELETE request made to these inactive topics will trigger a 4.04 NOT FOUND error. When a CoAP client performs a GET request to the broker address and the well-known URI, the *Topic Map* is returned, with the "mqtt" gateway resource always included, emphasizing the option to interact with MQTT topics.

C. CoAP request methods mapping

While MQTT clients can publish or subscribe to topics, CoAP offers four types of requests, GET, POST, PUT, and DELETE. These CoAP requests are adapted into MQTT logic as follows:

1) **POST Request**: Used to create a new resource with new data. Upon receiving a POST request, the extension extracts the topic name and payload, then creating a Publish message on the identified topic. The response code is either CREATED or CHANGED based on the existence of the topic. The retain flag of the Publish message is typically set to true to save the message for future clients.
2) **PUT Request**: Similar to the POST request, it updates the content of a resource, but it cannot create a new one. Upon receiving a PUT request, the extension extracts the URI and payload fields and checks for the topic's existence. If the topic doesn't exist, it responds with the code METHOD NOT ALLOWED. Otherwise:
   - It sends a CoAP response with the code CHANGED.
   - Updates the timestamp in the *Topic Map*.
   - Publishes an MQTT message with the extracted field, similar to the POST case.
3) **DELETE Request**: Used to delete a resource. Upon receiving a DELETE request, the extension checks for the requested topic's existence and, if positive:
   - Cancels the topic from the *Topic Map*.
   - Publishes an MQTT message with an empty payload and retain flag, effectively clearing the topic from the MQTT perspective.
   - Sends a CoAP response with the DELETED code.
4) **GET Request**: Used to retrieve the state of a resource. The extension extracts the topic name from the URI and retrieves the last retained message from the topic. The response can be negative (NOT FOUND) if the requested topic does not exist or if the topic exists but no saved message is found, or positive (CONTENT) if both the topic and the saved message are found.

Regarding QoS levels, all CoAP requests sent as NON-confirmable requests are set to QoS=0, while confirmable requests are typically set with QoS=1, ensuring confirmation mechanisms.

D. GET Observe

The GET Observe method, introduced in RFC 7641 [19], extends CoAP architecture into a synchronous structure. It

allows a CoAP client to emulate SUBSCRIBE behavior, receiving real-time notifications without constant polling.
To implement this method, our extension introduces the *Observe Map*, which stores reference data for all clients engaged in an observe session. This map uses URI/topics as keys and stores three elements for each client session:

- The reference to the CoAP Exchange used for the request/response
- A sequence number, updated with each notification sent
- A timestamp, marking the session as active or inactive

When a client initiates an observe session by sending a GET request with the observe flag set to 0, the system creates a new entry in the *Observe Map* with the provided information. An MQTT **Publish Interceptor** tracks notification updates, searching for matches between published topics and those stored in the *Observe Map*. If a match is found, a GET response is promptly sent to all observing clients through the CoAP Exchange reference. The same mechanism is applied to incoming POST and PUT requests from CoAP clients, ensuring the *Observe Map* is consistently updated. In the event of a DELETE request on an observed topic, observers receive a closing notification about the topic deletion.
Clients can perform a GET Observe on all existing topics stored in the *Topic Map*. If the topic is storing a retained message at the time of the request, it is sent as the initial response (CONTENT). Otherwise, if the topic has been active but lacks a retain message, the CoAP client receives a "VALID" response code, indicating the topic is active and awaiting notifications.
To terminate a relation, the CoAP client has several options:

- Sending a GET Observe cancel message, with the observe flag set to 1, removing its info from the *Observe Map*, closing the relation and receiving the last observed message as a final response.
- Sending a RST message with the associated token, deleting all related info.
- Taking no action, allowing the broker to delete the relation if the associated timestamp surpasses a certain threshold value.

E. Security Implementations

The system also supports security and authentication mechanisms. As the modified broker operates as a unified system, security and encryption between MQTT and CoAP messages within the broker are unnecessary. Instead, security is managed independently for each protocol.
On the MQTT side, TLS encryption is enabled, with certificate-based authentication implemented on the broker's secure port 8883. Similarly, CoAP encryption is achieved through DTLS (Datagram TLS), utilizing PSK (Pre-Shared Key) methods for authentication. PSK was chosen for its lightweight implementation, although alternative methods can be integrated into our broker. The security implementation is facilitated by the Scandium submodule of the Californium framework.

V. Experimental Results

A testing scenario was devised to evaluate the performance of the broker system under various conditions. Using a container-based network with Docker, multiple CoAP and MQTT clients communicated through the extended broker over a simulation period of one hour.

Four types of clients were designed as follows:

- **MQTT Subscriber**: Sends subscribe requests to multiple topics and switches subscriptions every 5 minutes.
- **MQTT Publisher**: Sends publish messages at variable frequencies, with a maximum waiting time of 5 seconds between consecutive messages.
- **CoAP Observer**: Establishes observe relations to different topics, switching subscriptions every 5 minutes.
- **CoAP Publisher**: Sends POST, PUT, GET, or DELETE requests at a similar rate to the MQTT Publisher client.

CoAP clients were built using the Eclipse Californium framework (Java), while MQTT clients were developed with the Eclipse Paho library (Python). We deployed five clients for each type, resulting in a total of 20 active clients in the scenario. Additional characteristics considered for the testing phase include:

- Message payloads with short (timestamp) or longer (more than 1024 bytes) sizes.
- Use of security (TLS for MQTT and DTLS for CoAP).
- Packet Loss conditions, at increasing percentages of [5%, 10%, 20%]

For a uniform analysis, the Quality of Service (QoS) level was set to 1, while in CoAP, Confirmable (CON) requests were employed. To simulate packet loss within Docker containers and assess its impact on our extension, Pumba, an open-source tool for testing Docker applications under adverse network conditions, was utilized [18].
A total of four simulation scenarios were considered, with an increasing packet loss, each collecting data for one hour. These are summarized in Table 1. To evaluate the performance of our extension, latency was chosen as a primary metric. Latency refers to the time required between the creation of a packet and its processing by the receiving client. Latencies were saved based on the type of request to compare the performance of interactions between different protocols and within the same protocol. Additionally, CPU and memory performance of the broker during the simulation were monitored. These values were sampled every few seconds from the container running the MQTT broker equipped with our extension to obtain a time trend.
Figure 3 depicts the average latency measurements under ideal network conditions, where packet loss is absent. Each bar shows the average and standard deviation of the end-to-end latency (from message transmission to arrival to the final receiver), computed over approximately 20,000 messages sent. Additionally, it includes benchmark round-trip time (RTT) measurements for MQTT communication conducted on the standard HiveMQ broker, provided for reference. Although the latency measurements reflect the influence of extension

TABLE I
TESTING SCENARIOS

Scenario	Security	Payload	Packet Loss
1	No	Short	[0, 5, 10, 20]
2	No	Long	[0, 5, 10, 20]
3	Yes	Short	[0, 5, 10, 20]
4	Yes	Long	[0, 5, 10, 20]

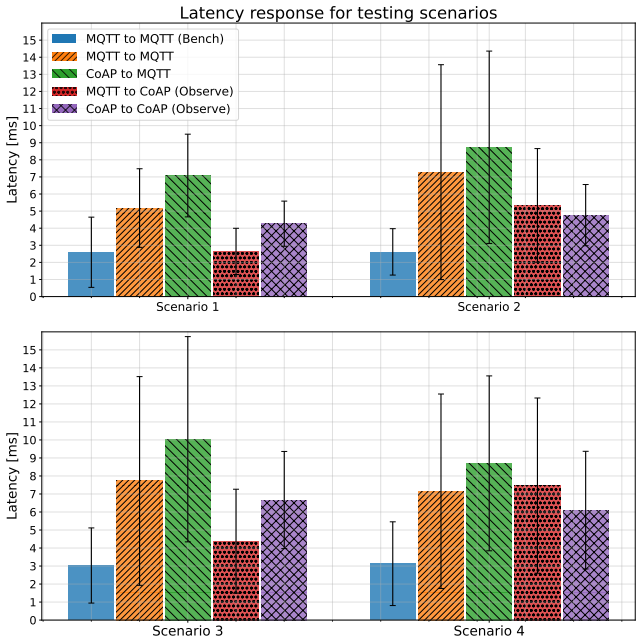

Fig. 3. Average latencies for different testing scenario in ideal network conditions.

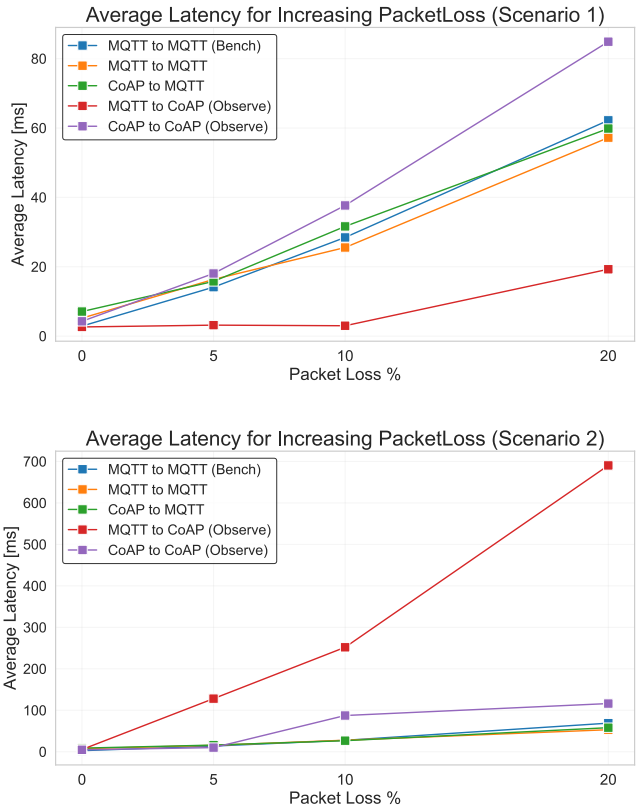

Fig. 4. Average latencies with increasing packet loss in case of short (above) and long (below) payload, W/O security.

broker processing, it is noteworthy that the performance remains comparable to the benchmark levels. This observation suggests that, despite the additional processing overhead, the extension broker's performance is considered satisfactory. This evaluation is particularly pertinent when taking into account the pivotal role the extension broker plays in enabling CoAP-MQTT interactions that would otherwise be unattainable.

A more detailed examination reveals several specific observations:

- Apart from the benchmarking MQTT scenario (in blue), the second best performing is the MQTT to COAP (Observe) path (in red). This is due to our implementation which directly forwards MQTT messages to the Californium module if the publication topic matches any topic in the *Observe Map*.
- An increase in latency is noticeable when longer payloads are sent, particularly in scenarios involving CoAP. This latency surge is primarily due to fragmentation issues, resulting in a higher volume of packets being transmitted within the CoAP framework.
- Despite the implementation of security mechanisms, such as encryption and authentication, the latency remains within acceptable bounds. Therefore, security measures do not significantly worsen latency.

When packet loss increases, as anticipated, latency performance tends to deteriorate, consistent with our previous expectations. However, the overall behavior observed previously remains confirmed. For reference, Figure 4 displays reported latencies in the absence of security measures. It is notably evident how the communication flow between MQTT and CoAP is significantly disrupted, especially with longer payloads and escalating packet loss, underscoring the impact of fragmentation. Results with security measures are not reported here as they do not exhibit significant variation.

The evaluation of resource consumption was conducted by monitoring memory and CPU utilization data of the Docker container hosting the MQTT broker throughout the entire simulation duration. Figure 5 illustrates the average results obtained under ideal network conditions. An initial observation reveals that memory consumption remains limited, ranging between 2-2.6% of the total dedicated memory (15.61 GB). This value slightly increases over time due to the rising traffic and increased information handling demands. The average CPU usage remains below 20%, although occasional peaks with higher consumption are observed. These peaks typically coincide with intensive operations performed by the broker, such as the automatic lookup of *Topic Map* or

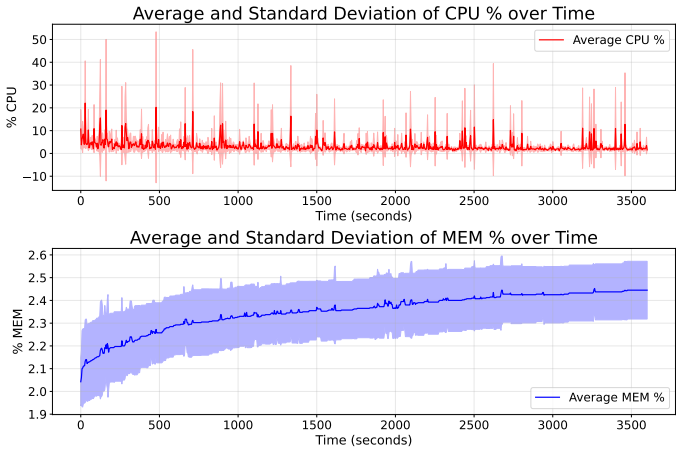

Fig. 5. CPU and MEM measurements.

Observe Map, or subscription/observe switching, which occur simultaneously for most subscribing (SUB) clients after predefined intervals.

VI. Conclusions and Future Work

In the vast landscape of Internet of Things (IoT) protocols, interoperability emerges as a crucial component, facilitating seamless connections and efficient management of devices. Our focus on two widely used, standardized IoT protocols, MQTT and CoAP, revealed a lack of a standardized and efficient solution for two-way communications.

The CoAP-MQTT extended broker presented in this study addresses this gap by enhancing interoperability, offering real-time intercommunication and seamless connectivity. Moreover, it integrates security mechanisms and fundamental functionalities of both protocols. Our experiments demonstrate that the system's performance is comparable to benchmark MQTT solutions, still delivering innovative interoperability features and an easy-to-deploy architecture that remains transparent to end-users.

As for future developments, our efforts aim to further enhance the system's capabilities, such as enabling MQTT clients to access external CoAP resources through the broker, expanding interoperability possibilities. Additionally, we plan to optimize the architecture, explore load-balancing in multi-broker scenarios, and investigate edge/cloud interactions. Conducting stress testing will provide a more comprehensive analysis of our broker's performance compared to other existing solutions, facilitating ongoing advancements in IoT interoperability.

Acknowledgment

This study was carried out within the MICS (Made in Italy Circular and Sustainable) Extended Partnership and received funding from Next-Generation EU (Italian PNRR M4 C2, Invest 1.3 – D.D. 1551.11-10-2022, PE00000004). CUP MICS D43C22003120001

References

[1] C. G. García, D. Meana-Llorián, J. M. C. Lovelle, et al. A review about smart objects, sensors, and actuators. International Journal of Interactive Multimedia & Artificial Intelligence, 4(3), 2017.

[2] Global Connectivity Index. Huawei Technologies Co., Ltd., 2015. Web. 6 Sept. 2015. http://www.huawei.com/minisite/gci/en/index.html

[3] Manyika, et al. (2015) The Internet of Things: Mapping the Value Beyond the Hype. Mckinsey Global Institute, San Francisco.

[4] J. Sidna, B. Amine, N. Abdallah, and H. El Alami. Analysis and evaluation of communication protocols for iot applications. In Proceedings of the 13th International Conference on Intelligent Systems: Theories and Applications, SITA'20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450377331. doi: 10.1145/3419604.3419754. URL https://doi.org/10.1145/3419604.3419754.

[5] Yuang Chen, Thomas Kunz, " Performance Evaluation of IoT Protocols under a Constrained Wireless Access Network", International Conference on Selected Topics in Mobile Wireless Networking (MoWNeT), IEEE Explore, 978-1-5090-1743-0, June 2016.

[6] Slavko Zitnik, Marko Jankovi c, Klemen Petrov ci c, Marko Bajec," Architecture of Standard-based, Interoperable and Extensible IoT Platform", IEEE Explore, 978-1-5090-4086-5, Jan 2016

[7] A. Amjad, F. Azam, M. W. Anwar, and W. H. Butt. A systematic review on the data interoperability of application layer protocols in industrial iot. IEEE Access, 9: 96528–96545, 2021. doi: 10.1109/ACCESS.2021.3094763.

[8] M. Collina, G. E. Corazza, and A. Vanelli-Coralli. Introducing the qest broker: Scaling the iot by bridging mqtt and rest. In 2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications - (PIMRC), pages 36–41, 2012. doi: 10.1109/PIMRC.2012.6362813.

[9] M. Dave, J. Doshi, and H. Arolkar. Mqtt- coap interconnector: Iot interoperability solution for application layer protocols. In 2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), pages 122–127, 2020. doi: 10.1109/I-SMAC49090.2020.9243377.

[10] M. Dave, M. Patel, J. Doshi, and H. Arolkar. Ponte message broker bridge configuration using mqtt and coap protocol for interoperability of iot. In Computing Science, Communication and Security, pages 184–195, Singapore, 2020. Springer Singapore. ISBN 978-981-15-6648-6.

[11] C.-H. Lee, Y.-W. Chang, C.-C. Chuang, and Y. H. Lai. Interoperability enhancement for internet of things protocols based on software-defined network. In 2016 IEEE 5th Global Conference on Consumer Electronics, pages 1–2, 2016. doi: 10.1109/GCCE. 2016.7800510.

[12] E. Palavras, K. Fysarakis, I. Papaefstathiou and I. Askoxylakis, "SeMIBIoT: Secure Multi-Protocol Integration Bridge for the IoT," 2018 IEEE International Conference on Communications (ICC), Kansas City, MO, USA, 2018, pp. 1-7, doi: 10.1109/ICC.2018.8422486. keywords: Protocols;Wireless sensor networks;Encryption;Bridges;Internet of Things;Wireless communication,

[13] Vinícius A. Barros, Sérgio A. B. Junior, Sarita M. Bruschi, Francisco J. Monaco, and Júlio C. Estrella. 2019. An IoT multi-protocol strategy for the interoperability of distinct communication protocols applied to web of things. In Proceedings of the 25th Brazillian Symposium on Multimedia and the Web (WebMedia '19). Association for Computing Machinery, New York, NY, USA, 81–88. https://doi.org/10.1145/3323503.3349546

[14] Mqtt.org. URL https://mqtt.org. November 2024 [Active]. Official website of the MQTT protocol.

[15] Hivemq community edition. URL https://github.com/hivemq/ hivemq-community-edition. GitHub page for downloading the HiveMQ CE broker.

[16] Zach Shelby, Klaus Hartke, and Carsten Bormann. "The Constrained Application Protocol (CoAP)." Request for Comments, RFC 7252, RFC Editor, June 2014. DOI: 10.17487/RFC7252. URL: https://www.rfc-editor.org/info/rfc7252.

[17] Californium. URL https://eclipse.dev/californium/. Californium page.

[18] A. Ledenev. Pumba: chaos testing tool for docker. URL https://github.com/ alexei-led/pumba. Page for downloading the Pumba, chaos testing command line tool for Docker containers.

[19] Klaus Hartke. ”Observing Resources in the Constrained Application Protocol (CoAP).” RFC 7641, RFC Editor, September 2015. URL: https://www.rfc-editor.org/info/rfc7641