Enhancing IoT Edge Platforms: Selecting an MQTT-Compatible Broker for Kubernetes Environments

Bahareh Aghajanpour*, Alessio Bucaioni*, Gabriele Capannini*
* Mälardalen University (Sweden)
† *name.surname@mdu.se*

***Abstract*—This paper aims to enhance IoT communication in edge platforms by identifying suitable MQTT-compatible solutions. The primary focus is on investigating two critical quality attributes of message brokers: data throughput and latency. To guide this research, we address two questions: (1) How do protocols like MQTT and their architecture contribute to messaging in IoT communications? and (2) How can MQTT architecture be optimized to meet the requirements of the ABB IoT Edge Platform? A mixed-methods research approach was applied, encompassing both quantitative and qualitative phases. The quantitative phase involves experimental evaluation of data throughput and latency among various message brokers, while the qualitative phase includes stakeholder interviews and literature review. Results indicate that event-driven architecture, publisher-subscriber design patterns, and the NATS message broker provide effective support for MQTT-based IoT edge platforms. Notably, NATS demonstrated high throughput and low latency compared to alternative brokers, making it a strong candidate for use in IoT edge environments. These findings offer valuable insights into best practices for MQTT-compatible message brokers in IoT platforms, specifically benefiting software architects developing for the ABB IoT edge platform.**

***Index Terms*—IoT, Edge, MQTT, Kubernetes, NATS**

I. Introduction

The Internet of Things (IoT) refers to a network of interconnected, intelligent objects capable of autonomously transferring data [1]. At its core, IoT involves the seamless integration of physical objects with the internet, creating vast networks that serve applications ranging from household devices to industrial systems [2]. Regardless of scale, efficient data transfer is essential for IoT devices. Typically, these devices transmit telemetry data to edge gateways and cloud platforms, often relying on the Message Queuing Telemetry Transport (MQTT) protocol. Known for its publisher-subscriber design pattern, MQTT facilitates machine-to-machine communication, making it particularly suited for IoT applications [3]. Consequently, MQTT has emerged as a key protocol in supporting reliable communication and data exchange within IoT platforms.

Message brokers act as intermediaries in these systems, translating and routing messages to meet the requirements of different software components, services, or systems. They leverage the MQTT protocol to bridge applications and middleware, creating seamless connections between disparate components [4]. However, the abundance of broker implementations presents software architects with the challenge of selecting a suitable broker, especially in enterprise IoT contexts where data throughput and latency are critical considerations [5]. In clustered environments, factors such as capacity planning, configuration, and system availability add further complexity [6].

Initial research by ABB has highlighted limitations with certain brokers, such as Mosquitto, especially concerning latency and data throughput. For instance, on ABB's IoT edge platform, Mosquitto has exhibited persistence issues and struggles to handle the high data throughput required by enterprise-level IoT deployments. As IoT platforms expand, the need for scalable brokers with robust performance becomes increasingly important.

In this paper, we aim to enhance IoT communication in edge platforms by identifying an MQTT-compatible broker that aligns with the requirements of an event-driven, publisher-subscriber architecture. Experimental evaluation of various MQTT-compatible brokers, including NATS, indicates that NATS is a promising candidate to replace Mosquitto in Kubernetes-orchestrated edge environments, particularly for ABB's IoT edge platform.

The remainder of this paper is structured as follows. Section II outlines the research process. Section III details the experimental setup and results. Section IV presents the findings derived from experimental and research data. Section V discuss the findings of this research, while Section VI compares this study with relevant works. Finally, Section VII concludes the paper and suggests avenues for future research.

II. Research process

The primary Research Goal (RG) was to identify an architectural solution, including a message broker, that would function effectively for an IoT application on an edge-based platform with Kubernetes orchestration. We broke down the overarching RG into the following Research Questions (RQs):

RQ1: How do protocols like MQTT and their architecture contribute to messaging in IoT communications?

RQ2: How can the MQTT architecture be enhanced to meet the requirements of the ABB IoT edge platform?

To provide answers to these questions, we employed a mixed-methods research approach, inspired by [7], [8], combining experimental analysis and literature review to gain insights into MQTT broker limitations and to identify best practices for IoT edge platforms on Kubernetes clusters. The research process began with a quantitative phase, where we conducted experiments to evaluate the data throughput and latency of different message brokers under various conditions. This phase helped us in answering to RQ1. The qualitative phase followed, involving an in-depth literature review to contextualize these findings within broader architectural considerations and industry best practices. This phase helped us in answering

to RQ2 and, partially, RQ1. In the final interpretation phase, we integrated results from both quantitative and qualitative methods, creating a comprehensive perspective that addresses existing limitations and offers informed recommendations for enhancing IoT edge platforms.

A. Literature review

To address our research questions, particularly RQ2, we conducted comprehensive literature reviews as the qualitative phase of our study. Striving to maintain a balance between rigor and agility, we adopted a streamlined protocol for our literature review, structured as follows:

Search strategy: our primary academic databases were IEEE Xplore and ScienceDirect, with the majority of literature sourced from IEEE. We conducted searches using keywords such as "MQTT," "Kubernetes clustering," "IoT architecture," and "message brokers." Additionally, we employed Boolean operators to combine these keywords (e.g., "MQTT AND IoT architecture," "MQTT OR IoT architecture," "MQTT AND message broker"). To ensure comprehensive coverage, we also included relevant conference proceedings, official website documentation, and books.

Inclusion and exclusion criteria: given that IoT is a rapidly evolving field, we focused on articles and conference papers published within the last decade. We prioritized publications that concentrated on MQTT architecture, IoT edge computing, and message brokers, selecting only those written in or translated into English. Publications not directly relevant to our main focus were excluded.

Paper selection: an initial screening of titles and abstracts led to the selection of 25 relevant papers for full-text review. Following a detailed examination, three papers were excluded due to lack of relevance.

Data extraction: to directly address RQ2, we systematically reviewed nine of these papers, while the remaining studies provided a broader understanding of the topic. Key findings and insights were extracted and synthesized to answer the research questions comprehensively throughout the report.

B. Experiment

Our selection of message brokers was guided by specific criteria aligned with the requirements of this study, informed by relevant literature [3], [9]. The key features considered were as follows:

- Latency: time taken for each broker to deliver a message from publisher to subscriber.
- Throughput: rate of messages sent per second.
- Ease of setup.
- Compatibility with MQTT protocol standards.

These criteria were chosen to ensure that the selected message broker could meet the real-time communication needs of our IoT system while adhering to industry standards and minimizing implementation complexity. Initially, five message brokers were considered—HiveMQ, VerneMQ, RabbitMQ, NATS, and Apache Kafka—which are discussed further in Section III. Due to time constraints, only one of these brokers was benchmarked in this study. Although Apache Kafka was considered initially, challenges in setup led to an unsuccessful attempt at benchmarking. Kafka's complex deployment requirements were deemed unsuitable, given the priority for ease of setup in this study. As a result, we relied on existing research data for Apache Kafka [10], [11]. Similarly, we used benchmarking data from existing studies for HiveMQ and VerneMQ [12], which are also discussed in **Section** III. After further consideration, NATS was selected over RabbitMQ for benchmarking due to its simple deployment, efficient performance characteristics, and its event-driven architecture [13].

Experimental environment: The test environment consisted of a Linux Ubuntu 20.04.6 LTS Virtual Machine with a 16-core CPU, 128GB RAM, and 250GB HDD. Docker version 26.0.0 was installed to facilitate deployment. To benchmark the performance of NATS, we used JetStream [14], a built-in distributed system within the NATS ecosystem that enhances functionality by operating atop the core NATS services and enabling accessibility from all client applications. JetStream was used to conduct both benchmark and stress tests, simulating high loads and evaluating data throughput and latency within the NATS messaging system. The tests focused on both publishers and subscribers under various load scenarios, enabling a detailed examination of system performance under different conditions.

Experimental setup: to work around security restrictions limiting internet access, we pre-downloaded the NATS image for deployment. The NATS server was started using Docker, which allowed for simplified deployment. Although Docker can support clustered configurations with multiple NATS servers, our focus was on benchmarking a single broker instance to isolate and measure performance in a controlled environment. JetStream was activated on top of the NATS server via the command `nats-server -m 8222 -js`, providing confirmation of server readiness through a "Server is ready" status message. Due to network restrictions, the Docker command for running JetStream included network proxy credentials to facilitate connectivity.

Test scenarios for throughput and latency: we structured the benchmarking tests around six throughput scenarios and eight latency scenarios to assess NATS under different conditions. For the throughput Scenarios, configurations varied by message size, number of publishers and subscribers, and message volume. Each scenario was designed to test the broker's performance under progressively increased load and message size variations. For the latency scenarios, we tested NATS in request-reply mode using various combinations of publishers, subscribers, and message counts. Each latency scenario simulated different usage patterns to evaluate system responsiveness under varying loads. Latency was measured by averaging the time taken for each request-response cycle, allowing for comparative analysis across scenarios. This procedural setup allowed for a structured assessment of NATS under diverse workload scenarios, providing insights into the broker's capacity to handle the demands of an IoT edge environment. A summary of the specific experimental results is presented in Section III.

III. How do protocols like MQTT and their architecture contribute to messaging in IoT communications? (RQ1)

As outlined earlier, to address RQ1, we evaluated five message brokers: HiveMQ, VerneMQ, RabbitMQ, NATS, and Apache Kafka. Due to time constraints, direct benchmarking

was conducted only for NATS. For the remaining brokers, we relied on benchmarking data from prior studies, as detailed below.

A. Experimental results for NTS industrial benchmarking

Due to security restrictions limiting internet connectivity, the NATS image was pre-downloaded for deployment. The remainder of the setup was conducted via Docker. A NATS server was initiated using the `docker run` command, though it is possible to create clusters of multiple NATS servers via Docker [13]. However, for this study, we aimed to benchmark NATS in a controlled, single-server environment to accurately measure performance for individual publisher and subscriber pairs. JetStream, an integral feature of NATS for benchmarking, was activated using the command `nats-server -m 8222 -js`, with the server readiness confirmed by the `Server is ready` status.

For the experimental work, network proxy credentials were incorporated into the Docker command to overcome connection restrictions. The performance of NATS was then tested across six throughput scenarios and eight latency scenarios to evaluate its capacity in handling varied workloads.

A set of tests was conducted by varying the number of publishers in the set $\{1, 4, 16\}$ and the number of subscribers in the set $\{1, 4, 16, 64\}$, across different scenarios with message sizes of 2 and 20 kilobytes. The results in Table I indicate that as the number of publishers and subscribers increases, the overall bandwidth also increases until the load on the system exceeds its capacity, then it becomes congested and the bandwidth degrades.

# Pubs	# Subs	Msg/s (GB/s)	
		2kB	20kB
1	1	568820 (1.08)	86408 (1.65)
1	4	804433 (1.53)	137822 (2.63)
1	16	1026244 (1.96)	162124 (3.09)
1	64	903633 (1.72)	156847 (2.99)
4	1	579655 (1.11)	73812 (1.41)
4	4	963657 (1.84)	112364 (2.14)
4	16	1391364 (2.65)	218907 (4.18)
4	64	1233010 (2.35)	182787 (3.49)
16	1	562086 (1.07)	76624 (1.46)
16	4	1056885 (2.02)	125987 (2.40)
16	16	1306175 (2.49)	213806 (4.08)
16	64	1152619 (2.20)	118377 (2.26)

TABLE I: Bandwidth results for messages of 2 kB and 20 kB in Messages per Second and GB/s.

To evaluate latency, NATS was tested in eight request-reply scenarios with varied publisher, subscriber, and message configurations, simulating different system loads. In each scenario, average latency was calculated as the inverse of the publishing rate.

- Scenario 1: configured with 1 publisher, 1 subscriber, and 1,000 messages, this setup resulted in an average latency of approximately 0.4325 ms.
- Scenario 2: with 20 publishers and 20 subscribers for 1,000 messages, latency decreased significantly to 0.0321 ms, highlighting how increasing publishers and subscribers can reduce latency.
- Scenario 3: with 50 publishers, 50 subscribers, and 1,000 messages, latency dropped further to 0.0260 ms.
- Subsequent scenarios: increasing the message count to 10,000 across different publisher-subscriber configurations resulted in latency values as low as 0.0173 ms in high-load conditions, showing an inverse relationship between message volume and latency.

These latency results, shown in Table II, indicate that higher numbers of publishers, subscribers, and messages generally lead to reduce the latency, showing NATS efficiency under high-load conditions. The explaination could be that¸ when more publishers and subscribers are present in the system, multiple messages can be processed concurrently. Applying batching and pipelining techniques, NATS is able to optimize the network usage more effectively by reducing the average latency per message.

# Pubs	# Subs	Messages	Latency (ms)
1	1	1000	0.4325
20	20	1000	0.0321
50	50	1000	0.0260
1	1	10000	0.3950
20	20	10000	0.0299
50	50	10000	0.0173

TABLE II: Latency results NATS benchmark

These findings demonstrate a trade-off between latency and throughput in NATS. Increasing system complexity—such as by adding more publishers and subscribers—can improve performance but also requires additional resources, impacting overall system design. However, by further increasing the number of publishers and subscribers, we expect that the broker and the network will become saturated, thereby increasing the latency due to resource contention.

B. Comparison of NAS with other brokers

We selected other four brokers, encompassing both MQTT-native and MQTT-compatible options. VerneMQ and HiveMQ are MQTT-native brokers, while the others—RabbitMQ, Apache Kafka are compatible with the MQTT protocol. For all these brokers, we relied on benchmarking data from prior studies.

VerneMQ: VerneMQ is a high-performance MQTT-native broker designed for scalability and robust messaging applications [15]. According to Koziolek et al. [12], VerneMQ was developed to overcome scalability limitations in brokers like RabbitMQ, which is based on the AMQP protocol. VerneMQ is lightweight yet powerful, capable of managing millions of device connections [15].

HiveMQ: Another MQTT-native broker, HiveMQ is known for its reliability and scalability, making it well-suited for mission-critical IoT and messaging applications [16]. HiveMQ offers strong security features, including TLS encryption, authentication, and integration with third-party security systems, making it ideal for applications with stringent security requirements. Performance tests conducted by Koziolek et al. [12] (see Table III) reveal key distinctions between

VerneMQ and HiveMQ. VerneMQ achieved a throughput of 10,000 messages per second (msg/s) with a latency of 8.7 milliseconds, delivering messages quickly and efficiently for applications that require low latency. In contrast, HiveMQ achieved a throughput of 8,000 msg/s but with a higher latency of 119.4 milliseconds, making it less suitable for time-sensitive applications, despite its strengths in reliability and message handling. Both brokers offer scalability through multi-threaded architectures and support horizontal scaling, with the test environment limited to eight CPU cores. This suggests that performance could potentially improve with additional resources.

RabbitMQ: an open-source broker implementing the Advanced Message Queuing Protocol (AMQP), RabbitMQ offers flexibility and reliability in message delivery [17]. Supporting both point-to-point and pub-sub architectures, RabbitMQ is versatile for diverse application needs. However, it may not be ideal for applications requiring guaranteed precise delivery of each message, as it ensures delivery of at least one message among several. Despite this, RabbitMQ performs well in complex routing scenarios, which makes it valuable for certain applications.

Apache Kafka: Kafka is a distributed streaming platform designed for high-throughput data processing, commonly used in real-time analytics and event-driven architectures [10]. Kafka's high throughput and low latency provide strong message delivery and ordering guarantees, making it suitable for applications requiring strict sequencing. However, Kafka's complexity in setup and configuration may present challenges for simpler IoT implementations. Performance analyses by GCORE [11] and benchmarking by Confluent [18] (Table III) indicate that Kafka achieves a throughput of up to 2,000,000 msg/s with a latency of 5 milliseconds, ideal for high-speed data streams. RabbitMQ, in comparison, manages up to 60,000 msg/s with a latency of around 10 milliseconds, a figure that may vary based on message routing complexity, system configuration, and hardware resources. RabbitMQ's emphasis on reliability may contribute to slightly higher latency than other platforms.

System	Throughput (msg/s)	Latency(ms)
HiveMQ	8000	119.4
VerneMQ	10000	8.7
RabbitMQ	60000	10.0
Kafka	2000000	5.0

TABLE III: Performance comparison of HiveMQ, VerneMQ, RabbitMQ, and Kafka

In summary, based on the requirements of this work—including high throughput, low latency, MQTT compatibility, and ease of deployment—NATS emerges as the most suitable message broker for the application discussed in this study. Its strong performance metrics, combined with MQTT compatibility and a focus on simplicity, make it an ideal choice for enhancing IoT communication within the ABB IoT edge platform.

IV. How can the MQTT architecture be enhanced to meet the requirements of the ABB IoT edge platform? (RQ2)

To strengthen the MQTT protocol for the ABB IoT Edge Platform, we conducted a systematic literature review to inform architectural improvements. Al-Awami et al. [1] emphasize the importance of redundancy and scalability in IoT architectures, highlighting the need for robust messaging systems to mitigate potential failures. Al-Fuqaha et al. [2] identify MQTT's lightweight advantages for IoT applications but note its limitation in relying on a single broker, thus supporting the enhancement of MQTT with multiple brokers for greater resilience. Johari et al. [19] underscore the need for efficient routing within MQTT to ensure reliable message delivery, suggesting a more resilient broker architecture. Additionally, Shahri et al. [20] propose real-time and reliability enhancements to MQTT, such as message prioritization and deadlines, aligning with the demands of dependable IoT communication. Further studies reinforce this perspective. Koziolek et al. [12] evaluate various MQTT brokers, emphasizing performance and scalability in distributed IoT environments. Deploying each broker instance on a separate cluster node allows for future scalability to handle increased demands. Bandai et al. [21] advocate for geographically distributed brokers to boost performance and redundancy, with Kubernetes assigning brokers to distinct cluster nodes for continued stability in case of failures. Grüner et al. [6] highlight how Kubernetes orchestration can improve system stability by distributing MQTT brokers across nodes, thus minimizing single points of failure and enhancing system availability.

Drawing on these insights, we propose an enhanced MQTT architecture that integrates an event-driven model with a pub/sub pattern to support real-time communication in IoT systems. The MQTT architecture includes clients that publish or subscribe to messages, with a central broker acting as an intermediary to deliver these messages to subscribers. To support real-time communication, we propose an event-driven architecture, where events are generated by an event producer, transmitted to an event handler, and distributed among event consumers [22]. This real-time event handling is well-suited to IoT applications, where timely communication is essential. To enhance this architecture, we couple the event-driven model with the pub/sub (publisher-subscriber) design pattern, wherein a publisher sends messages to a broker that then distributes these messages to subscribers via output channels. This design supports real-time data handling and aligns with MQTT's architecture for IoT systems. Additional features, such as client properties and a centralized controller, are introduced to enable effective message delivery by prioritizing attributes like deadlines. A database aids in analyzing these real-time attributes to ensure prompt communication. In edge-platform IoT environments, Kubernetes orchestration provides a robust framework for distributed systems by organizing containers into Pods, which are deployed on virtual or physical nodes based on system requirements [23]. Running system modules within a Kubernetes cluster offers benefits such as container management, service discovery, and self-healing, although it also introduces some complexities [24]. To further enhance system stability and redundancy in MQTT-based IoT

architectures, we propose three solutions:

- Node-based broker deployment: each broker instance is deployed on a separate node within a single system cluster, allowing the system to scale as needed. Instances synchronize to transfer messages and related data. If an instance fails, others can continue to operate, thus increasing system availability and stability. This setup uses the pub/sub design pattern, allowing applications, such as mobile apps, to subscribe to topics published by device broker clients, resulting in a more resilient messaging framework.
- Clustered broker deployment without queue mirroring: this solution involves deploying multiple broker instances across different hardware nodes. Although the instances operate independently, each can handle client connections and process messages, ensuring system resilience even if one broker fails [6].
- Geographical broker deployment: this solution involves deploying brokers based on geographic locations to enhance system stability. Clients subscribe to topics based on their location, and Kubernetes assigns each broker to a distinct cluster node. This approach distributes workload across brokers, allowing the system to maintain operations if a broker fails, as other brokers take over responsibilities.

Each of these solutions enhances MQTT's performance, redundancy, and scalability in IoT systems by leveraging Kubernetes orchestration and clustered broker setups.

V. Discussion

In this study, we set out to enhance IoT communications on edge platforms by analysing the MQTT protocol, architectural considerations, and broker selection. Our focus on message brokers was guided by performance evaluations centred around the key quality attributes of throughput and latency within MQTT-based IoT architectures. Additionally, we explored the role of orchestration tools like Kubernetes in optimizing message broker performance within distributed systems.

As described in Section III, our experimental analysis primarily concentrated on benchmarking the NATS message broker. The detailed results, presented in Tables I and II, offer insights into metrics such as the number of messages sent, average message rate, and message size, providing a nuanced understanding of throughput for individual publishers and subscribers. Specifically, our tests on the NATS broker with 8 publishers and 4 subscribers, using message sizes of 2 kB and 20 kB, yield valuable findings for applications with similar configurations. However, this setup may not fully capture performance characteristics for larger-scale implementations with an extended number of clients.

Our focus throughout this study was on the MQTT protocol, which is widely recognized as a suitable choice for IoT environments due to its lightweight design. While our work did not assess protocol alternatives, as such considerations fell outside the scope of this work, discussions on protocol selection are well-covered in works such as [25]. Instead, our analysis concentrated on choosing an appropriate message broker, evaluating both MQTT-native and MQTT-compatible options. This narrowed focus provided clarity in broker selection but may limit some alternative solutions. It is also important to note that this work was conducted within a company context, where the decision to use the MQTT protocol and specific brokers had been made prior to our involvement. Nonetheless, selecting a manageable and efficient broker like NATS offers considerable benefits, particularly for software architects seeking ease of deployment and flexibility across varied applications where setup simplicity is essential.

Our findings also suggest that MQTT, when combined with an event-driven architecture and pub/sub design pattern, is effective in supporting real-time communication and message handling in IoT systems. While using MQTT as the primary messaging protocol may restrict some architectural flexibility, adopting this protocol along with a pub/sub design pattern can significantly improve the efficiency of real-time request-reply interactions in IoT communications.

Additionally, Kubernetes plays a crucial role in managing brokers within an IoT edge platform. Its capabilities extend beyond basic management; Kubernetes enables clustering of message brokers, allowing flexible allocation and failover support. In scenarios where one or more brokers fail, Kubernetes can automatically reassign responsibilities among the remaining brokers, ensuring system continuity. As discussed in Section IV, this clustering feature helps maintain the functionality of other application components even if specific brokers experience failures.

In conclusion, our proposed solution integrates the MQTT protocol with an event-driven architecture and a pub/sub design pattern within a clustered Kubernetes environment, where each node hosts a dedicated message broker. Based on our findings, we recommend NATS as the most suitable message broker due to its compatibility with MQTT and its strong performance in applications characterized by a moderate number of publishers and subscribers and message sizes between 2 kB and 20 kB. This configuration offers a viable model for system architects designing similar IoT edge platforms that require scalability, reliability, and real-time communication.

VI. Related work

In this study, our goal was to propose an efficient and reliable message broker architecture for IoT applications, specifically identifying a broker compatible with an IoT platform that utilizes the MQTT protocol for message queuing services. Previous research has proposed various architectures for client-broker communication. For example, [21] outlines two architectures: one where brokers broadcast messages to all clients and another where clients subscribe to specific subjects, allowing publisher clients to share information only with relevant subscribers. Several studies, including [19] and [26], present the basic MQTT architecture consisting of publishers, subscribers, and a broker. In this setup, publishers send messages to the broker, which then delivers these messages to subscribers based on their subscribed topics. This architecture aligns with our approach, where we adopt a pub/sub design pattern with a central broker that facilitates communication between publisher and subscriber clients.

Additionally, message brokers vary from native MQTT solutions, such as HiveMQ [16] and VerneMQ [15], to alternatives that are compatible with MQTT but not exclusively MQTT-based, like Amazon Kafka [10], RabbitMQ [17], and

NATS [13]. Various studies have compared these brokers based on different quality attributes. For instance, [12] conducted experiments focusing on throughput and latency across three MQTT-based brokers, revealing that factors such as the underlying programming language affect performance. Erlang-based brokers, such as EMQX and VerneMQ, demonstrated superior performance compared to the Java-based HiveMQ in this study, indicating that EMQX and VerneMQ achieved better results in terms of both throughput and latency.

In this work, we propose NATS as the preferred message broker due to its MQTT compatibility, despite not being natively MQTT-based. According to our experiments, NATS achieved higher throughput and lower latency compared to HiveMQ, VerneMQ, and RabbitMQ. A study by [6] also evaluated MQTT brokers, finding that EMQX had the highest throughput, followed by VerneMQ and HiveMQ. However, our results with NATS demonstrated significantly higher throughput and lower latency than these MQTT-native brokers. Additionally, NATS exhibited an estimated average latency of approximately 238.6 milliseconds per request, underscoring its suitability for high-performance IoT applications.

To ensure a robust evaluation of message brokers, we followed a structured benchmarking approach, building on the results of studies by [12] and [6]. These studies conducted comprehensive performance tests on HiveMQ and VerneMQ using a test infrastructure with Dual Intel Xeon E5-2640 v3 CPUs (32 threads), 128 GB RAM, and Gigabit connectivity, with StarlingX v3.0 on CentOS 7.6 supporting the virtualization environment. MZBench was employed for benchmarking. In comparison, our test environment consisted of a Linux Ubuntu 20.04.6 LTS Virtual Machine with a 16-core CPU, 128 GB RAM, and a 250 GB HDD. Docker version 26.0.0 was used, with JetStream [14] deployed to benchmark NATS performance.

Comparisons of broker architectures have revealed that protocol efficiency, as well as scalability, is crucial for high-performance IoT and automotive systems [27].

VII. Conclusion and future work

This work enhanced IoT communication on edge platforms by analysing the MQTT protocol. Our findings indicate that an event-driven architecture, pub/sub design pattern, and NATS message broker are effective choices for MQTT-based IoT systems, achieving high throughput and low latency. Benchmarking results showed NATS handling over 1 million messages per second, with data transfer rates up to 1.22 GB/sec, highlighting its suitability for high-performance IoT applications. Additionally, Kubernetes was found to improve system scalability and availability by effectively managing message brokers.

For future work, expanding the scope to include additional transport protocols beyond MQTT could reveal more options for IoT communication. Investigating the newer MQTT version 5, which offers improved scalability and error handling, is also recommended for applications requiring enhanced performance in large-scale IoT systems.

References

[1] S. H. Al-Awami, M. M. Al-Aty, and M. F. Al-Najar, "Comparison of iot architectures based on the seven essential characteristics," in *2023 IEEE 3rd International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering (MI-STA)*. IEEE, 2023.

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, 2015.

[3] Ansyah, Adi Surya Suwardi and Arifin, Miftahol and Alfan, Muhammad Bahauddin and Suriawan, Matthew Vieri and Farhansyah, Nadhif Haikal and Shiddiqi, Ary Mazharuddin and Studiawan, Hudan, "Mqtt broker performance comparison between aws, microsoft azure and google cloud platform," 2023 International Conference on Recent Trends in Electronics and Communication (ICRTEC), IEEE, 2023.

[4] Apukhtin, Vladyslav and Shirokopetleva, Mariya and Skovorodnikova, Victoria, "The relevance of using message brokers in robust enterprise applications," 2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), IEEE, 2019.

[5] A. Bucaioni, P. Pelliccione, and S. Mubeen, "Modelling centralised automotive e/e software architectures," *Advanced Engineering Informatics*, vol. 59, p. 102289, 2024.

[6] S. Gruener, H. Koziolek, and J. Rückert, "2021 ieee 18th international conference on software architecture (icsa)," 2021, pp. 69–79.

[7] J. W. Creswell and V. L. Plano Clark, *Designing and Conducting Mixed Methods Research*. Sage Publications, 2017.

[8] A. Bucaioni, A. Di Salle, L. Iovino, I. Malavolta, and P. Pelliccione, "Reference architectures modelling and compliance checking," *Software and Systems Modeling*, vol. 22, no. 3, pp. 891–917, 2023.

[9] B. Mishra, B. Mishra, and A. Kertesz, "Stress-testing mqtt brokers: A comparative analysis of performance measurements," *Energies*, vol. 14, no. 18, 2021. [Online]. Available: https://www.mdpi.com/1996-1073/14/18/5817

[10] "Apache Kafka," https://kafka.apache.org/.

[11] "Gcore Compare," https://gcore.com/learning/nats-rabbitmq-nsq-kafka-comparison//.

[12] H. Koziolek, S. Grüner, and J. Rückert, "A comparison of mqtt brokers for distributed iot edge computing," in *Software Architecture. ECSA 2020*, A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya, and O. Zimmermann, Eds. Cham: Springer, 2020, p. Page numbers. [Online]. Available: https://doi.org/10.1007/978-3-030-58923-3_23

[13] "NATS," https://nats.io/.

[14] "Jetstream benchmark," https://docs.nats.io/nats-concepts/jetstream/.

[15] "VerneMQ," https://vernemq.com/.

[16] "HiveMQ," https://www.hivemq.com/.

[17] "RabbitMQ," https://www.rabbitmq.com/.

[18] "Confluent.io," https://developer.confluent.io/learn/kafka-performance//.

[19] R. Johari, S. Bansal, and K. Gupta, "Routing in iot using mqtt protocol," in *2020 12th International Conference on Computational Intelligence and Communication Networks (CICN)*. IEEE, 2020.

[20] E. Shahri, P. Pedreiras, and L. Almeida, "Enhancing mqtt with real-time and reliable communication services," in *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*, 2021, pp. 1–6.

[21] A. not provided, "Edge based mqtt broker architecture for geographical iot applications," in *2020 International Conference on Information Networking (ICOIN)*. IEEE, 2020.

[22] "Event-driven architecture, wikipedia," https://en.wikipedia.org/wiki/Event-driven_architecture/.

[23] "redhat," https://www.redhat.com/en/topics/containers/what-is-kubernetespod#:~:text=A%20Kubernetes%20pod%20is%20a,a%20more%20common%20use%20case).

[24] N. Kebbani, P. Tylenda, and R. McKendrick, *The Kubernetes Bible: The definitive guide to deploying and managing Kubernetes across major cloud platforms*, 2022.

[25] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," *Defence School of Communications and Information Systems*, Year of publication.

[26] A. not provided, "Mqtt-like network management architecture," in *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE, 2021.

[27] N. Kukulicic, D. Samardzic, A. Bucaioni, and S. Mubeen, "Automotive service-oriented architectures: a systematic mapping study," in *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2022, pp. 459–466.