Scalable and reliable MQTT messaging: Evaluating TBMQ for P2P scenarios

Dmytro I. Shvaika[1,2], Andrii I. Shvaika[1,2], Dmytro I. Landiak[2,6] and Volodymyr O. Artemchuk[1,3,4,5]

[1]*G.E. Pukhov Institute for Modelling in Energy Engineering of the NAS of Ukraine, 15 Oleg Mudrak Str., Kyiv, 02000, Ukraine*
[2]*ThingsBoard, Inc., 110 Duane Str., Suite 1C, New York, 10007, USA*
[3]*Center for Information-analytical and Technical Support of Nuclear Power Facilities Monitoring of the NAS of Ukraine, 34a Palladin Ave., Kyiv, 03142, Ukraine*
[4]*Kyiv National Economic University named after Vadym Hetman, 54/1 Peremohy Ave., Kyiv, 03057, Ukraine*
[5]*State Non-Profit Enterprise "State University "Kyiv Aviation Institute", 1 Liubomyra Huzara Ave., Kyiv, 03058, Ukraine*
[6]*Ternopil Ivan Pului National Technical University, 56 Ruska Str., Ternopil, 46001, Ukraine*

Abstract
Point-to-point (P2P) communication represents a critical messaging pattern in MQTT, enabling precise, reliable one-to-one exchanges essential for IoT applications like real-time monitoring, command control, and secure data transfer. As edge computing continues to expand, resource-constrained edge nodes demand high connection density and low-latency messaging to support real-time data exchange and responsive operations. This paper evaluates the performance of TBMQ, a scalable and fault-tolerant MQTT broker designed to meet the demands of modern IoT ecosystems. By leveraging a distributed architecture built on Kafka and Redis, TBMQ efficiently supports high-throughput P2P messaging while maintaining low latency and reliability. A series of performance tests were conducted, scaling workloads from 200,000 to 1,000,000 messages per second. These tests revealed TBMQ's linear scalability, efficient resource utilization, and robust reliability under high traffic loads. The findings position TBMQ as a powerful solution for IoT and edge computing environments requiring dependable P2P communication. Future enhancements, including Redis optimizations and expanded edge computing integration, are proposed to further improve performance and adaptability.

Keywords
MQTT, TBMQ, point-to-point messaging, performance testing, Redis, Kafka

1. Introduction

Message Queuing Telemetry Transport (MQTT) [1] is a lightweight protocol widely used in the Internet of Things (IoT) ecosystem for connecting edge devices. Its point-to-point (P2P) communication pattern facilitates direct and reliable one-to-one message exchanges, which are crucial for IoT applications like private messaging, command delivery, and other direct interaction use cases.

Scaling and ensuring reliability in edge-based P2P messaging becomes increasingly challenging as message loads grow to hundreds and millions of messages per second. Existing solutions often face synchronous communication bottlenecks and the complexity of managing distributed architectures, limiting their performance in high-demand IoT environments.

Recent research [2] highlights ThingsBoard as a leading open-source IoT platform widely adopted in academia and industry. Building on insights from its extensive use, the platform's developers created TBMQ – a high-performance MQTT broker tailored to IoT needs [3]. Launched in 2020 and open-sourced in 2023, TBMQ ensures reliability and scalability, handling 100 million connections and over 3 million messages per second, leveraging Kafka for efficient messaging.

doors-2025: 5th Edge Computing Workshop, April 4, 2025, Zhytomyr, Ukraine
shvaikad@gmail.com (D. I. Shvaika); andrew.shvayka@gmail.com (A. I. Shvaika); dlandiak2110@gmail.com (D. I. Landiak); ak24avo@gmail.com (V. O. Artemchuk)
https://www.researchgate.net/profile/Volodymyr-Artemchuk-2 (V. O. Artemchuk)
0009-0001-3088-3997 (D. I. Shvaika); 0009-0006-8461-3550 (A. I. Shvaika); 0009-0006-6846-5594 (D. I. Landiak); 0000-0001-8819-4564 (V. O. Artemchuk)

While modern brokers, like TBMQ, excel in handling massive connections and high throughput, P2P messaging introduces unique challenges. These include direct message routing, increased connection tracking, and ensuring low latency, and reliable delivery. Such scenarios demand efficient mechanisms for managing dynamic client interactions and maintaining scalability under heavy loads.

The key objectives and contributions of this study are:

- Evaluating TBMQ's ability to handle increasing P2P messaging loads for persistent DEVICE clients, which use shared Kafka topics to optimize resource usage and Redis for reliable delivery. Benchmark tests progressively scaled the system while increasing the message load from 200,000 to 1,000,000 messages per second.
- Providing a detailed performance analysis, including CPU utilization, latency, and resource scaling, to validate TBMQ's efficiency.

The results demonstrate that TBMQ provides reliable, scalable, and high-performance messaging, making it an optimal solution for edge-based IoT ecosystems.

2. The P2P problem in the scalable broker context

2.1. Challenges of P2P messaging for brokers

The survey presented in [4] offers an extensive evaluation of MQTT brokers, covering aspects such as protocol-specific features, security mechanisms, cloud deployment capabilities, and data visualization tools, alongside a comparison of MQTT client libraries. However, the study does not delve into the brokers' performance or suitability in P2P messaging scenarios, leaving room for further exploration in this area.

Effective resource management in P2P messaging systems, such as handling dynamic server availability and message buffering, remains a challenge. As demonstrated by Ma et al. [5], the modeling of P2P networks using queueing theory reveals how server availability fluctuations and preemptive priority policies influence system performance and energy efficiency. The 'free-riding' behavior and energy inefficiency represent significant obstacles for P2P messaging systems, especially in IoT environments. Studies such as [6] demonstrate how these issues can undermine system performance, necessitating advanced strategies like repairable breakdown models and differentiated service levels to optimize overall network behavior.

P2P messaging introduces distinct challenges for brokers due to several key aspects:

1. Direct Message Routing: In P2P interactions, messages are sent directly from a specific publisher to a specific subscriber, bypassing shared topics or group routing. This requires brokers to accurately track which clients are interacting and ensure low-latency delivery.
2. Increased Broker Load: In P2P scenarios, each message may be unique to a specific recipient, significantly increasing the number of routed connections. This adds extra load on brokers to manage sessions, connections, and queues efficiently.
3. Scalability: With potentially hundreds of thousands or even millions of simultaneous P2P connections, brokers must handle each connection quickly and reliably. This necessitates efficient mechanisms for state management, load balancing, and latency handling.
4. Reliability assurance: For many IoT applications, message loss is unacceptable, particularly in P2P scenarios where there is no group duplication. Brokers must ensure reliable delivery with minimal delay, even under high traffic conditions.
5. Decentralization requirements: In certain P2P scenarios (e.g., ensuring privacy or fault tolerance), brokers need to operate effectively without relying on a centralized node. This complicates the implementation of traditional MQTT mechanisms.

Thus, the core challenge of P2P messaging for brokers lies in enabling scalable, reliable, and low-latency message routing across numerous real-time connections.

2.2. Limitations of existing solutions

The scalability of existing communication protocols remains a challenge, particularly in large-scale IoT deployments. As noted in [7], cloud and fog computing solutions require efficient protocols, yet many—including MQTT—were not originally designed for decentralized P2P environments. This leads to scalability constraints and inefficiencies in handling massive direct connections. Although MQTT is widely used, its effectiveness in distributed deployments remains an active research topic.

While innovative queuing-based approaches [6] contribute to energy efficiency and help address free-riding behavior, their dependence on intricate mathematical models and parameter tuning might pose challenges to scalability in real-world MQTT broker environments.

To address the limitations of static routing technologies in P2P-based edge clouds, adaptive routing approaches like ARPEC have been proposed [8]. By combining message activity analysis, network topology, and minimum cost maximum flow (MCMF) graph mapping, ARPEC improves performance metrics such as user latency and node resource utilization in edge IoT systems. Although adaptive routing approaches such as ARPEC [8] improve Edge Cloud performance, their reliance on complex algorithms like MCMF and predictive models may hinder real-time scalability and adoption in resource-constrained IoT environments.

Standard encryption methods may impose a significant overhead on MQTT brokers in P2P environments. Alternative approaches, such as Value-to-HMAC, could be more efficient in resource-constrained conditions [9].

Despite advancements, existing brokers face significant limitations in supporting P2P messaging for edge environments:

- Synchronous Persistence Bottlenecks: Many brokers rely on synchronous mechanisms (e.g., disk-based storage or synchronous Redis clients), causing delays in high-throughput P2P scenarios.
- Scalability Constraints: Distributed architectures often introduce complexities in cluster management, limiting seamless scaling for P2P messaging.
- Edge Environment Challenges: Resource limitations at the edge constrain brokers' ability to maintain high-throughput, low-latency communication.
- Latency Sensitivity: Real-time P2P communication requires ultra-low latency, which existing brokers struggle to achieve consistently under heavy message loads.

3. Methodology for scalability and efficiency testing of TBMQ in P2P scenarios

The methodology for testing the scalability and efficiency of TBMQ builds upon the approaches used in prior evaluations of MQTT brokers under stress testing, as outlined in [10]. This study examined brokers like Mosquitto, ActiveMQ, HiveMQ, and others, focusing on metrics such as CPU usage, latency, and message rate, which are critical for analyzing broker performance in resource-constrained and high-throughput environments. Incorporating these methods ensures a robust and comparative assessment of TBMQ in P2P messaging scenarios.

To assess the TBMQ's ability to handle point-to-point communication at scale, five performance tests measuring throughput, efficiency, and latency under progressively increasing workloads were performed. The test environment, illustrated in Figure 1, supported a maximum throughput of 1 million messages per second. Throughput refers to the total number of messages per second, including both incoming and outgoing messages.

The performance environment was deployed on an Amazon Web Services Elastic Kubernetes Service (AWS EKS) cluster with horizontal scaling to accommodate growing demands. The TBMQ cluster included 5 nodes (16 virtual central processing units (vCPU), 32 GiB random-access memory (RAM) each), supported by 5 Kafka instances (2 vCPU, 4 GiB RAM each) [11, 12, 13] and 11 Redis instances (2 vCPU, 4 GiB RAM each) [14]. Drawing on the comparison between Redis and MS SQL presented

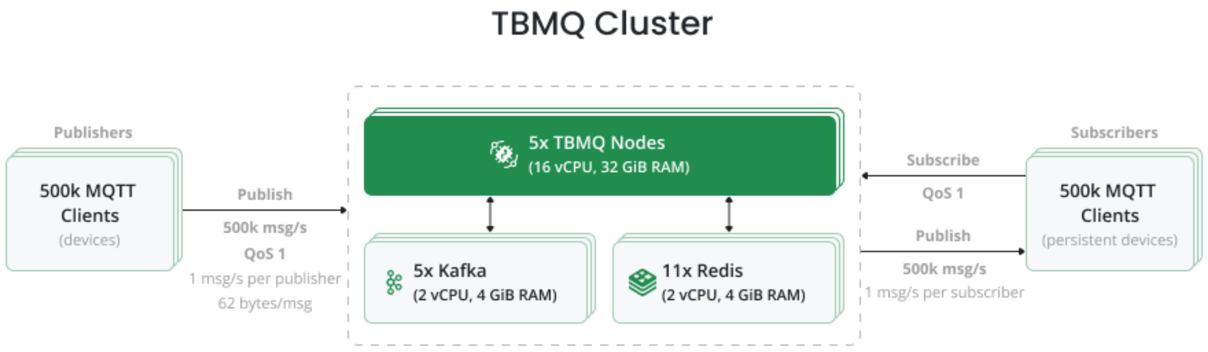

Figure 1: Architecture of the TBMQ cluster and test environment.

in the work [15], Redis is incorporated into the methodology for efficient state and message tracking, enabling high-throughput communication scenarios with minimal latency and resource overhead.

The tests simulated 500,000 publishers and subscribers, with each client pair operating at 1 message per second using Quality of Service (QoS) 1 for reliable delivery. Subscribers were configured with `clean_session=false`, ensuring that messages were retained and delivered even during offline periods. Published messages were 62 bytes in size, assigned to unique topics such as `"europe/ua/kyiv/$number"`, with corresponding subscriptions to `"europe/ua/kyiv/$number+"`, where `$number` identified each publisher-subscriber pair.

3.1. Test agent design for simulating TBMQ performance

To evaluate TBMQ's performance under increasing message traffic, a test agent architecture was designed to simulate large-scale publisher and subscriber activity. The test agent consisted of two main components: **runner pods** and an **orchestrator pod.** Each component was deployed on Amazon Elastic Compute Cloud (EC2) instances, a scalable virtual computing service provided by AWS [16, 17]. EC2 enables users to provision virtual machines with configurable CPU, memory, and network capacity, offering flexibility to handle varying workloads.

Runner pods

Runner pods were specialized for either publishing or subscribing. The number of publisher pods always matched the number of subscriber pods, ensuring symmetry in message flow. Pods were deployed on EC2 instances, with the number of instances and pods per instance adjusted based on the desired throughput.

Table 1
Scaling configuration of EC2 instances for runner pods.

Throughput (msg/sec)	Pods/Instance	Number of EC2 Instances
200k	5	1
400k	5	1
600k	10	1
800k	5	2
1M	5	4

As shown in table 1, throughput increases were managed by scaling the number of EC2 instances or pods per instance. For example, a throughput of 1 million messages per second required 4 instances, each hosting 5 pods.

Orchestrator pod

The orchestrator pod managed the execution and coordination of runner pods and was hosted on a dedicated EC2 instance. This instance also supported auxiliary monitoring tools, including:

- **Kafka Redpanda Console**: For real-time broker monitoring.
- **Redis Insight**: For analyzing database performance.

This modular architecture allowed the test agent to adapt dynamically to increasing traffic demands. By effectively distributing workloads across EC2 instances, it maintained consistent performance and reliable message delivery, even at high throughput levels.

3.2. Infrastructure overview

This section provides an overview of the test environment, highlighting the hardware specifications of the services utilized and the distribution of EKS cluster pods across AWS EC2 instances. EKS is a managed platform that simplifies the deployment and management of containers using the popular Kubernetes system [18]. It automates the orchestration of containerized applications, allowing them to run efficiently on scalable infrastructure. Kubernetes has become a cornerstone for deploying and managing large-scale distributed systems, offering advanced features such as horizontal pod autoscaling, rolling updates, and resource quotas. In our test environment, services such as TBMQ, Kafka, and Redis were deployed in containers within an EKS cluster and distributed across AWS EC2 virtual machines. This setup ensures optimal resource allocation, scalability, and performance during the testing process.

AWS RDS (Amazon Web Services Relational Database Service) is used for managing our PostgreSQL database. RDS is a managed service that simplifies the setup, operation, and scaling of relational databases in the cloud.

Table 2 below presents the hardware specifications for the services used in our tests:

Table 2
Hardware specifications for the services used in the tests.

Service name	TBMQ	Kafka	Redis	AWS RDS (PostgreSQL)
Instance type	c7a.4xlarge	c7a.large	c7a.large	db.m6i.large
vCPU	16	2	2	2
Memory (GiB)	32	4	4	8
Storage (GiB)	20	30	8	20
Network bandwidth (Gibps)	12.5	12.5	12.5	12.5

Note: To minimize costs during the load testing phase, only the Redis master nodes were used without replicas. This configuration enabled us to focus on achieving the target throughput without excessive resource provisioning.

Instance scaling was adjusted during each test to match workload demands, as described in the next section.

4. Results

TBMQ's performance was tested in phases, starting at 200k msg/sec and increasing by 200k each time, up to 1M msg/sec. In each phase, the number of TBMQ brokers and Redis nodes was scaled. For the 1M msg/sec test, the number of Kafka nodes was also scaled to handle the corresponding workload. The test configurations are summarized in table 3.

Key takeaways from the tests include:

- **Scalability:** TBMQ demonstrated linear scalability. Reliable performance was maintained as the message throughput increased from 200k to 1M msg/sec by incrementally adding TBMQ nodes, Redis nodes, and Kafka nodes at higher workloads.

Table 3
The test configurations are summarized.

Throughput (msg/sec)	Publishers/Subscribers	TBMQ nodes	Redis nodes	Kafka nodes
200k	100k	1	3	3
400k	200k	2	5	3
600k	300k	3	7	3
800k	400k	4	9	3
1M	500k	5	11	5

- **Efficient Resource Utilization:** CPU utilization on TBMQ nodes remained consistently around 90% across all test phases, indicating that the system effectively used available resources without overconsumption.
- **Latency Management:** The observed latency across all tests remained within two-digit bounds. This was predictable given the QoS 1 level chosen for our test, applied to both publishers and persistent subscribers. We also tracked the average acknowledgment latency for publishers, which stayed within single-digit bounds across all test phases.
- **High Performance:** TBMQ's one-to-one communication pattern showed excellent efficiency, processing about 8900 msg/s per CPU core. We calculated this by dividing the total throughput by the total number of CPU cores used in the setup.

Additionally, the following table and screenshots provide a comprehensive summary of the key elements and results of the final 1M msg/sec test.

Table 4
Summary of the key elements and results of the final 1M msg/sec test.

QoS	Msg latency Avg	Pub Ack Avg	TBMQ CPU Avg	Payload (bytes)
1	75ms	8ms	91%	62

Where:

- TBMQ CPU Avg: The average CPU utilization across all TBMQ nodes.
- Msg Latency Avg: The average duration from when a message is transmitted by the publisher to when it is received by the subscriber.
- Pub Ack Avg: The average time elapsed between the message transmission by the publisher and the reception of the PUBACK acknowledgment.

For detailed results of the performance evaluation, including insights into CPU utilization, Java Management Extensions (JMX) monitoring, and Redis instance metrics, please refer to the official documentation [19].

5. Discussion

Performance testing of TBMQ under extreme-scale P2P messaging revealed critical limitations in the existing broker architecture. A key issue encountered was the inefficiency of the Redis client library Jedis, which created a bottleneck due to its synchronous nature. Addressing this challenge required a migration to Lettuce, an asynchronous Redis client designed for high-concurrency scenarios.

The following sections describe this migration process, its technical challenges, and its impact on system performance.

5.1. Migrating from Jedis to Lettuce: overcoming a key testing challenge

One of the most significant challenges during performance testing was overcoming the limitations of the Jedis library, whose synchronous nature became a bottleneck in high-throughput scenarios. The

Jedis library is a popular Java client for Redis, which is an in-memory data structure store often used as a database, cache, or message broker. Jedis provides a straightforward API for interacting with Redis, allowing developers to perform operations like reading and writing data, managing keys, and executing Redis commands within Java applications. With Jedis, each Redis command is sent and processed sequentially, meaning the system has to wait for each command to complete before issuing the next one. This approach significantly limited Redis's potential to handle concurrent operations and fully utilize available system resources [20].

Solution: migrating to Lettuce

Lettuce is a scalable thread-safe Redis client for synchronous, asynchronous and reactive usage. Multiple threads may share one connection if they avoid blocking and transactional operations. To address the Jedis issue, migration was made to the Lettuce client, which leverages Netty under the hood for efficient asynchronous communication [21, 22]. Unlike Jedis, Lettuce allows multiple commands to be sent in parallel without waiting for their completion, enabling non-blocking operations and better resource utilization. This architecture made it possible to fully exploit Redis's performance capabilities, especially under high message loads.

Challenges during migration

The migration, however, was not trivial. It required:

- **Rewriting a substantial portion of the codebase** to transition from synchronous to asynchronous workflows.
- **Restructuring how Redis commands were issued and handled**.

Careful planning and rigorous testing ensured that these changes maintained system reliability and correctness.

By migrating to Lettuce, the bottleneck caused by Jedis was resolved, enabling the full utilization of Redis's potential in high-load scenarios. Although the migration process was challenging and required significant effort, it delivered substantial improvements in system performance.

5.2. Planned optimizations for Redis performance

Current State: Currently, Lua scripts are used in Redis to process messages, ensuring the atomicity of operations such as saving, updating, and deleting messages. This approach is critical for maintaining data consistency. However, due to the limitations of the Redis Cluster architecture, each script operates on a single client. This restriction arises because all keys accessed within a script must reside in the same hash slot [23].

Planned optimizations: Future plans include modifying the hashing mechanism for client identifiers (client IDs) to group more clients into the same Redis hash slot. This adjustment would allow a single Lua script to process multiple clients within the same hash slot. Such an approach is expected to reduce overhead and improve the efficiency of Redis operations while adhering to the cluster's constraints.

Future work in enhancing MQTT brokers for edge computing could benefit from incorporating dynamic data serialization methods [24, 25] and leveraging advanced state synchronization techniques like CRDTs [26]. These approaches could further address challenges in IoT scalability, interoperability, and real-time data consistency in distributed environments.

6. Conclusion

The evaluation of TBMQ in point-to-point (P2P) messaging scenarios has demonstrated its capability to meet the demands of scalable, reliable, and low-latency communication in IoT applications. By leveraging a robust architecture built on Kafka and Redis, TBMQ has shown linear scalability and efficient resource utilization, maintaining performance across a wide range of throughput conditions—from 200,000 to 1 million messages per second.

Key performance insights include consistent CPU utilization at 90% under high loads, efficient memory management, and low message latency even during peak operations. The migration from Jedis to Lettuce was instrumental in overcoming bottlenecks in Redis communication, enabling asynchronous workflows and higher concurrency. Additionally, planned optimizations for Redis, such as refining Lua scripting mechanisms, are expected to further enhance TBMQ's operational efficiency.

The results position TBMQ as a powerful solution for edge-based IoT ecosystems, addressing the unique challenges of P2P messaging, including direct message routing, dynamic client management, and reliability. Future developments, focusing on Redis optimizations and enhanced integration with edge computing frameworks, will strengthen TBMQ's adaptability to evolving IoT requirements.

In conclusion, TBMQ emerges as a scalable and dependable MQTT broker, setting a new benchmark for P2P communication in distributed IoT environments.

Author Contributions: Dmytro I. Shvaika and Andriy I. Shvaika contributed to the main text of the manuscript; methodology, Volodymyr O. Artemchuk; Dmytro I. Shvaika and Dmytro I. Landyak contributed to the software development and were responsible for the development and execution of performance test; Review writing and editing: all authors contributed. All authors have read and agreed with the published version of the manuscript.

Funding: This research was supported by ThingsBoard, Inc., which provided the cloud infrastructure necessary for conducting the performance tests. The authors gratefully acknowledge their contribution to this research.

Data Availability Statement: The TBMQ source code is publicly available at https://github.com/thingsboard/tbmq. Additionally, the source code of the performance test tool used in this study can be accessed at https://github.com/thingsboard/tb-mqtt-perf-tests.

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

Acknowledgments: The authors express their sincere gratitude to the entire ThingsBoard, Inc. team for their contributions and collaborative efforts in the development and evaluation of the system.

Declaration on Generative AI: During the preparation of this work, the authors used ChatGPT-4 and Grammarly for: abstract drafting, content enhancement, grammar and spelling check, sentence polishing, and rephrasing to improve clarity and conciseness. After using these tools, the authors reviewed and edited the content as needed and took full responsibility for the publication's content. No AI-generated text was used to create new content or sections of this work.

References

[1] MQTT, MQTT: The standard for IoT messaging, 2022. URL: https://mqtt.org/, Accessed: January 2025.

[2] P. Di Felice, G. Paolone, Papers mentioning things board: A systematic mapping study, Journal of Computer Science 20 (2024) 574–584. doi:10.3844/jcssp.2024.574.584.

[3] ThingsBoard, TBMQ, 2023. URL: https://thingsboard.io, Accessed: January 2025.

[4] B. Mishra, A. Kertesz, The use of MQTT in M2M and IoT systems: A survey, IEEE Access 8 (2020) 201071–201086. doi:10.1109/ACCESS.2020.3035849.

[5] Z. Ma, M. Yan, R. Wang, S. Wang, Performance analysis of P2P network content delivery based on queueing model, Cluster Computing 27 (2023). doi:10.1007/s10586-023-04111-w.

[6] Y. Shen, Z. Ma, The analysis of P2P networks with malicious peers and repairable breakdown based on Geo/Geo/1+1 queue, Journal of Parallel and Distributed Computing 195 (2025) 104979. URL: https://www.sciencedirect.com/science/article/pii/S0743731524001436. doi:https://doi.org/10.1016/j.jpdc.2024.104979.

[7] J. Dizdarevic, F. Carpio, A. Jukan, X. Masip, A survey of communication protocols for IoT and related challenges of fog and cloud computing integration, ACM Computing Surveys 51 (2018). doi:10.1145/3292674.

[8] B. Dong, J. Chen, An adaptive routing strategy in P2P-based Edge Cloud, Journal of Cloud Computing 13 (2024). doi:10.1186/s13677-023-00581-w.

[9] D. Dinculeană, X. Cheng, Vulnerabilities and limitations of MQTT protocol used between IoT devices, Applied Sciences 9 (2019). URL: https://www.mdpi.com/2076-3417/9/5/848. doi:10.3390/app9050848.

[10] B. Mishra, B. Mishra, A. Kertész, Stress-testing MQTT brokers: A comparative analysis of performance measurements, Energies 14 (2021) 5817. doi:10.3390/en14185817.

[11] Kafka, Apache Kafka., 2024. URL: https://kafka.apache.org/, Accessed: January 2025.

[12] S. Vyas, R. K. Tyagi, C. Jain, S. Sahu, Performance evaluation of apache kafka – a modern platform for real time data streaming, in: 2022 2nd International Conference on Innovative Practices in Technology and Management (ICIPTM), volume 2, 2022, pp. 465–470. doi:10.1109/ICIPTM54933.2022.9754154.

[13] S. Park, J.-H. Huh, A study on big data collecting and utilizing smart factory based grid networking big data using apache kafka, IEEE Access 11 (2023) 96131–96142. doi:10.1109/ACCESS.2023.3305586.

[14] Redis, Redis Pub/Sub, 2024. URL: https://redis.io/docs/latest/develop/interact/pubsub/, Accessed: January 2025.

[15] G. Muradova, M. Hematyar, J. Jamalova, Advantages of redis in-memory database to efficiently search for healthcare medical supplies using geospatial data, in: 2022 IEEE 16th International Conference on Application of Information and Communication Technologies (AICT), 2022, pp. 1–5. doi:10.1109/AICT55583.2022.10013544.

[16] AWS, AWS, 2023. URL: https://aws.amazon.com/, Accessed: January 2025.

[17] A. Wittig, M. Wittig, Amazon Web Services in Action: An in-depth guide to AWS, Simon and Schuster, 2023.

[18] Kubernetes, Kubernetes documentation, 2023. URL: https://kubernetes.io/, Accessed: January 2025.

[19] ThingsBoard, 1M Throughput P2P Performance Test, https://thingsboard.io/docs/mqtt-broker/reference/1m-throughput-p2p-performance-test/#performance-tests, 2025.

[20] Baeldung, Intro to Jedis – the Java Redis Client library, 2024. URL: https://www.baeldung.com/jedis-java-redis-client-library.

[21] Netty, The netty project, 2024. URL: https://netty.io/, Accessed: January 2025.

[22] M. Norman, W. Marvin, Netty in action, Manning, 2015.

[23] BoynerTechnology, Limitations of scripting with Lua in RedisApache, 2023. URL: https://boynergroup.medium.com/limitations-of-scripting-with-lua-in-redis-b4381bd9629f, Accessed: January 2025.

[24] D. I. Shvaika, A. I. Shvaika, V. O. Artemchuk, Advancing IoT interoperability: dynamic data serialization using ThingsBoard, Journal of Edge Computing 3 (2024) 126–135. doi:10.55056/jec.745.

[25] D. I. Shvaika, A. I. Shvaika, V. O. Artemchuk, Data serialization protocols in IoT: problems and solutions using the ThingsBoard platform as an example, in: T. A. Vakaliuk, S. O. Semerikov (Eds.), Proceedings of the 4th Edge Computing Workshop (doors 2024), Zhytomyr, Ukraine, April 5, 2024, volume 3666 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024, pp. 70–75. URL: https://ceur-ws.org/Vol-3666/paper11.pdf.

[26] A. Prymushko, I. Puchko, M. Yaroshynskyi, D. Sinko, H. Kravtsov, V. Artemchuk, Efficient State Synchronization in Distributed Electrical Grid Systems Using Conflict-Free Replicated Data Types, IoT 6 (2025) 6. doi:10.3390/iot6010006.