Article

Not peer-reviewed version

MQTT Broker Architectural Enhancements for High-Performance P2P Messaging: TBMQ Scalability and Reliability in Distributed IoT Systems

Dmytro Shvaika *, Andrii Shvaika , Volodymyr Artemchuk

Posted Date: 7 May 2025

doi: 10.20944/preprints202505.0445.v1

Keywords: MQTT; TBMQ; P2P messaging; Kafka; Redis; Lettuce

 doi:10.20944/preprints202505.0445.v1

Article

MQTT Broker Architectural Enhancements for High-Performance P2P Messaging: TBMQ Scalability and Reliability in Distributed IoT Systems

Dmytro Shvaika [1,2,*,†], Andrii Shvaika [1,2,†] and Volodymyr Artemchuk [1,3,4,5,†]

[1] G.E. Pukhov Institute for Modelling in Energy Engineering of the NAS of Ukraine, Kyiv, 02000, Ukraine
[2] ThingsBoard, Inc., New York, 10007, USA
[3] Center for Information-Analytical and Technical Support of Nuclear Power Facilities Monitoring of the NAS of Ukraine, Kyiv, 03142, Ukraine
[4] Kyiv National Economic University named after Vadym Hetman, Kyiv, 03057, Ukraine
[5] State Non-Profit Enterprise "State University "Kyiv Aviation Institute", Kyiv, 03058, Ukraine
* Correspondence: shvaikad@gmail.com
† These authors contributed equally to this work.

Abstract: The Message Queuing Telemetry Transport (MQTT) protocol remains a key enabler for lightweight and low-latency messaging in Internet of Things (IoT) applications. However, traditional broker implementations often struggle with the demands of large-scale point-to-point (P2P) communication. This paper presents a performance and architectural evaluation of TBMQ, an open-source MQTT broker designed to support reliable P2P messaging at scale. The broker employs Redis Cluster for session persistence and Apache Kafka for message routing, with additional optimizations including asynchronous Redis access via Lettuce and Lua-based atomic operations. Stepwise load testing was performed using Kubernetes-based deployments on Amazon EKS, progressively increasing message rates to 1 million messages per second (msg/s). The results demonstrate that TBMQ achieves linear scalability and stable latency under increasing load, reaching an average throughput of 8900 msg/s per CPU core while maintaining end-to-end delivery latency within two-digit milliseconds bounds. These findings confirm that TBMQ's architecture provides an effective foundation for reliable, high-throughput messaging in distributed IoT systems.

Keywords: MQTT; TBMQ, P2P messaging; kafka; redis; lettuce

1. Introduction

The increasing complexity and scale of IoT systems have created a pressing need for messaging solutions that combine lightweight protocols with the ability to handle high device density, low latency, and reliable delivery [1,2]. However, in high-throughput IoT deployments, these requirements remain a challenge for many MQTT brokers—particularly under point-to-point (P2P) communication patterns that impose strict performance and persistence guarantees. With billions of connected devices exchanging data across domains such as industrial automation, urban infrastructure, and healthcare, efficient device-to-device communication is becoming increasingly important in scenarios that demand direct, low-latency delivery between endpoints [3,4].

As the ecosystem evolved and performance demands increased, the ThingsBoard development team—well known in the academic community due to the platform's widespread adoption in scientific research, as evidenced by the mapping study of Di Felice and Paolone [5] and numerous applied studies [6–12]—identified the need for a specialized messaging solution capable of providing scalability, fault tolerance, and real-time delivery.

This led to the creation of TBMQ, a distributed MQTT broker optimized for high-performance communication in the Internet of Things. The system was initially developed in 2020, deployed in production environments by 2021, and released as open-source software in 2023. Internal testing

 doi:10.20944/preprints202505.0445.v1

demonstrated its ability to support over 100 million concurrent connections and process several million messages per second.

The majority of MQTT brokers face significant bottlenecks in terms of scalability and latency, primarily due to their reliance on disk-based storage systems aimed at maximizing crash durability. While MQTT brokers typically excel in one-to-many [13] and many-to-one [14] communication patterns, they often encounter performance bottlenecks in high-throughput one-to-one (point-to-point, P2P) messaging scenarios. These challenges become more pronounced in large-scale deployments, where high concurrency, low latency, and efficient resource utilization are critical. Common architectural limitations include blocking I/O operations, centralized routing logic, and poor scalability under concurrent load.

The scientific problem addressed in this work is how to design an MQTT broker architecture that overcomes these scalability and latency limitations in the context of high-throughput P2P messaging, while ensuring reliable message persistence and support for horizontal scalability. While distributed message brokers have been studied extensively, there remains a lack of horizontally scalable architectures that implement multi-layered persistence models capable of delivering low-latency message processing while preserving system resilience and fault tolerance.

This research addresses that gap by presenting the architectural evolution of TBMQ, an open-source MQTT broker that was originally designed to reliably aggregate data from IoT devices and forward it to back-end systems using Kafka for durable message routing. To support high-throughput point-to-point messaging at scale, the persistence layer of TBMQ was redesigned by transitioning from a PostgreSQL-based architecture to a horizontally scalable, in-memory model based on Redis. This shift eliminated key performance bottlenecks associated with disk-based storage and enabled low-latency message delivery with reliable session management. This paper details how this transformation was implemented and demonstrates its practical benefits in distributed IoT environments where efficient resource usage, resilience, and reliable message delivery are essential.

The scientific contribution of this study lies in the design and implementation of a horizontally scalable, multi-layered persistence architecture for MQTT brokers, which combines Redis for low-latency in-memory session management with Kafka for fast and reliable message routing. Unlike traditional approaches that rely on monolithic, disk-based databases for session and message persistence, this architecture offloads operational workloads to distributed, in-memory storage layers. This enables true horizontal scalability for high-throughput point-to-point (P2P) messaging patterns in MQTT systems, while preserving reliability and message ordering.

The goal of this study is to demonstrate how the architectural decisions and performance improvements defined in the recent 2.0.x releases of TBMQ [15], optimize persistent session handling, enhance P2P messaging, and improve overall system efficiency within a scalable IoT architecture.

1.1. Related Work

The architectural scalability of MQTT brokers has been extensively studied in recent years. Spohn [16] provides a comprehensive analysis of typical MQTT scalability bottlenecks and explores clustering and federation techniques to mitigate broker overload in distributed environments. Hmissi and Ouni [17] proposed SDN-DMQTT, a reconfigurable MQTT architecture that leverages software-defined networking to dynamically adapt broker topologies, thereby improving resilience and adaptability. Akour et al. [18] introduced a multi-level elasticity model that enables brokers to scale responsively based on workload demands. Despite these advancements, most existing work focuses on one-to-many or many-to-one communication patterns and does not explicitly address point-to-point (P2P) messaging scenarios.

Several researchers have conducted benchmark-driven evaluations of MQTT brokers under load. Mishra et al. [19**?**] performed stress testing of various brokers under concurrent workloads, identifying key factors influencing throughput and latency. Dizdarevic et al. [**?**] experimentally benchmarked multiple open-source MQTT brokers, revealing significant performance variation across implementations. Kashyap et al. [**?**] analyzed EMQX's internal architecture to demonstrate how it supports

 doi:10.20944/preprints202505.0445.v1

scalability and high availability. However, these studies rarely examine one-to-one communication or persistence strategies under high-throughput conditions.

Recent efforts have explored integrating MQTT brokers with modern stream-processing and caching backends such as Kafka and Redis. Gupta [20] investigated a Redis–Kafka architecture for time-series ingestion, highlighting its suitability for latency-sensitive workloads. Likewise, EMQX [21] demonstrated how Redis-based extensions can enhance MQTT with real-time data processing capabilities. These integrations have informed the design of TBMQ, which employs Redis for in-memory session persistence and Kafka for durable message routing.

Although point-to-point messaging is increasingly critical for direct device-to-device communication in IoT systems, it remains underrepresented in MQTT research. Mishra and Kertesz [22] surveyed MQTT usage in M2M and IoT contexts but did not focus on the architectural challenges of P2P messaging. Ma et al. [23] applied queueing theory to model server availability and contention in P2P content delivery, showing its impact on performance and energy efficiency. Shen and Ma [24] extended this analysis to networks with malicious peers, proposing repairable breakdown models to maintain robustness. Dong and Chen [25] proposed ARPEC, an adaptive routing strategy for edge P2P systems using graph-based optimization, though its computational complexity may hinder real-time deployment. Security concerns were addressed by Dinculeană and Cheng [26], who proposed lightweight alternatives to conventional encryption, such as Value-to-HMAC, to reduce overhead in resource-constrained P2P deployments.

Despite these contributions, horizontally scalable architectures that explicitly support reliable, high-throughput point-to-point messaging in MQTT systems remain underexplored. To address this gap, the present study introduces TBMQ—an open-source MQTT broker enhanced with a multi-layered persistence design combining Kafka for durable message routing and Redis for low-latency in-memory session state management. This architecture is intended to support scalable and fault-tolerant P2P communication in distributed IoT environments. Preliminary results and architectural motivations were previously outlined in a short paper presented at CEUR Workshop Proceedings [27]. This article extends that work with a more detailed analysis, broader experimental validation, and a structured discussion of architectural trade-offs, particularly concerning horizontal scalability, persistent session handling, and reliable point-to-point messaging at scale.

2. Materials and Methods

This section describes the architectural and implementation details of TBMQ, the configuration and optimization of the persistence layer, and the experimental environment used to evaluate the system's performance.

2.1. Architecture and Implementation Details

While the TBMQ 1.x version can 100 million clients [28] at once and 3 million msg/s [29], as a high-performance MQTT broker it was primarily designed to aggregate data from IoT devices and deliver it to back-end applications reliably (QoS 1). This architecture is based on operational experience accumulated by the TBMQ development team through IIoT and other large-scale IoT deployments, where millions of devices transmit data to a limited number of applications.

These deployments highlighted that IoT devices and applications follow distinct communication patterns. IoT devices or sensors publish data frequently but subscribe to relatively few topics or updates. In contrast, applications subscribe to data from tens or even hundreds of thousands of devices and require reliable message delivery. Additionally, applications often experience periods of downtime due to system maintenance, upgrades, failover scenarios, or temporary network disruptions.

To address these differences, TBMQ introduces a key feature: the classification of MQTT clients as either standard (IoT devices) or application clients. This distinction enables optimized handling of persistent MQTT sessions for applications. Specifically, each persistent application client is assigned a separate Kafka topic. This approach ensures efficient message persistence and retrieval when an MQTT client reconnects, improving overall reliability and performance. Additionally, application

 doi:10.20944/preprints202505.0445.v1

clients support MQTT's shared subscription feature, allowing multiple instances of an application to efficiently distribute message processing.

Kafka [30] serves as one of the core components. Designed for high-throughput, distributed messaging, Kafka efficiently handles large volumes of data streams, making it an ideal choice for TBMQ [31]. With the latest Kafka versions capable of managing a huge number of topics, this architecture is well-suited for enterprise-scale deployments. Kafka's robustness and scalability have been validated across diverse applications, including real-time data streaming and smart industrial environments [31–33].

Figure 1 illustrates the full fan-in setup in a distributed TBMQ cluster.

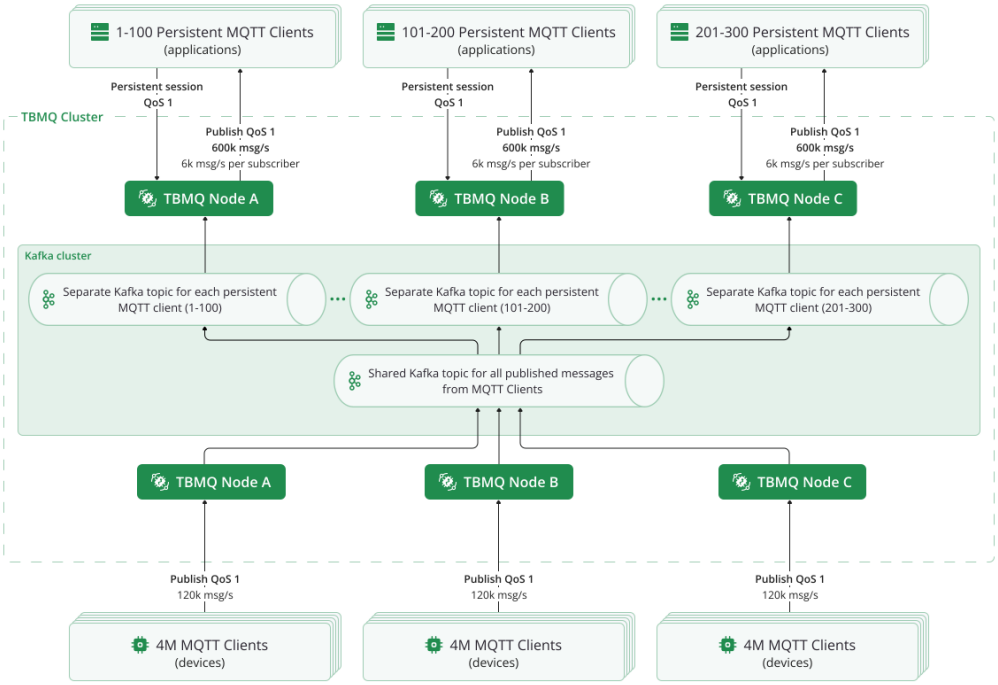

Figure 1. Overview of the TBMQ fan-in communication pattern in a distributed cluster.

In TBMQ 1.x, standard MQTT clients relied on PostgreSQL for message persistence and retrieval, ensuring that messages were delivered when a client reconnected. While PostgreSQL performed well initially, it had a fundamental limitation—it could only scale vertically. It was anticipated that, as the number of persistent MQTT sessions grew, PostgreSQL's architecture would eventually become a bottleneck. To address these scalability limitations, more robust alternatives were investigated to meet the increasing performance demands of TBMQ. Redis was quickly chosen as the best fit due to its horizontal scalability, native clustering support, and widespread adoption.

Unlike the fan-in, the point-to-point (P2P) communication pattern enables direct message exchange between MQTT clients. Typically implemented using uniquely defined topics, P2P is well-suited for private messaging, device-to-device communication, command transmission, and other direct interaction use cases.

One of the key differences between fan-in and peer-to-peer MQTT messaging is the volume and flow of messages. In a P2P scenario, subscribers do not handle high message volumes, making it unnecessary to allocate dedicated Kafka topics and consumer threads to each MQTT client. Instead, the primary requirements for P2P message exchange are low latency and reliable message delivery, even for clients that may go offline temporarily. To meet these needs, TBMQ optimizes persistent session management for standard MQTT clients, which include IoT devices.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 7 May 2025 doi:10.20944/preprints202505.0445.v1

Figure 2 shows how standard MQTT clients publish and receive direct messages via TBMQ, while Redis supports session persistence and Kafka facilitates routing.

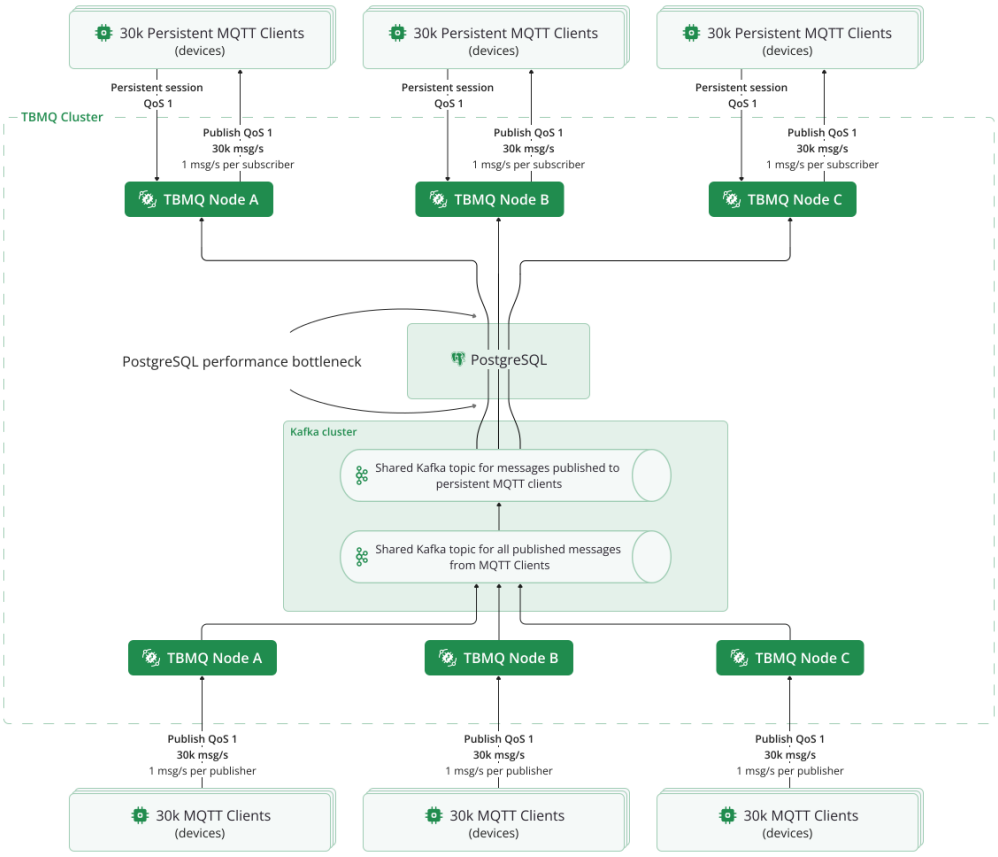

Figure 2. P2P communication in TBMQ using Redis for scalable persistence.

2.2. PostgreSQL Usage and Limitations

To fully understand the reasoning behind this shift, it's important to first examine how MQTT clients operated within the PostgreSQL architecture. This architecture was built around two key tables.

The `device_session_ctx` was responsible for maintaining the session state of each persistent MQTT client:

```
Table "public.device_session_ctx"
 Column            | Type                   | Nullable
-------------------+------------------------+----------
 client_id         | character varying(255)| not null
 last_updated_time | bigint                 | not null
 last_serial_number| bigint                 |
 last_packet_id    | integer                |
Indexes:
 "device_session_ctx_pkey" PRIMARY KEY, btree (client_id)
```

The key columns are last_packet_id and last_serial_number, which is used to maintain message order for persistent MQTT clients:

- *last_packet_id* represents the packet ID of the last MQTT message received.
- *last_serial_number* acts as a continuously increasing counter, preventing message order issues when the MQTT packet ID wraps around after reaching its limit of 65535.

The `device_publish_msg table` was responsible for storing messages that must be published to persistent MQTT clients (subscribers).

 doi:10.20944/preprints202505.0445.v1


```
Table "public.device_publish_msg"
 Column                  | Type                   | Nullable
-------------------------+------------------------+----------
 client_id               | character varying(255)| not null
 serial_number           | bigint                 | not null
 topic                   | character varying      | not null
 time                    | bigint                 | not null
 packet_id               | integer                |
 packet_type             | character varying(255)|
 qos                     | integer                | not null
 payload                 | bytea                  | not null
 user_properties         | character varying      |
 retain                  | boolean                |
 msg_expiry_interval     | integer                |
 payload_format_indicator| integer                |
 content_type            | character varying(255)|
 response_topic          | character varying(255)|
 correlation_data        | bytea                  |
Indexes:
 "device_publish_msg_pkey" PRIMARY KEY, btree (client_id, serial_number)
 "idx_device_publish_msg_packet_id" btree (client_id, packet_id)
```

The key columns to highlight:

- *time* – captures the system time (timestamp) when the message is stored. This field is used for periodic cleanup of expired messages.
- *msg_expiry_interval* – represents the expiration time (in seconds) for a message. This is set only for incoming MQTT 5 messages that include an expiry property. If the expiry property is absent, the message does not have a specific expiration time and remains valid until it is removed by time or size-based cleanup.

While this design ensured reliable message delivery, it also introduced performance constraints. To better understand its limitations, prototype testing was performed to evaluate PostgreSQL's performance under the P2P communication pattern. Using a single instance with 64GB RAM and 12 CPU cores, message loads were simulated with a dedicated performance testing tool [34] capable of generating MQTT clients and simulating the desired message load. The primary performance metric was the average message processing latency, measured from the moment the message was published to the point it was acknowledged by the subscriber. The test was considered successful only if there was no performance degradation, meaning the broker consistently maintained an average latency in the two-digit millisecond range.

Prototype testing ultimately revealed a throughput limit of 30k msg/s when using PostgreSQL for persistent message storage. Throughput refers to the total number of msg/s, including both incoming and outgoing messages, Figure 3.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 7 May 2025 doi:10.20944/preprints202505.0445.v1

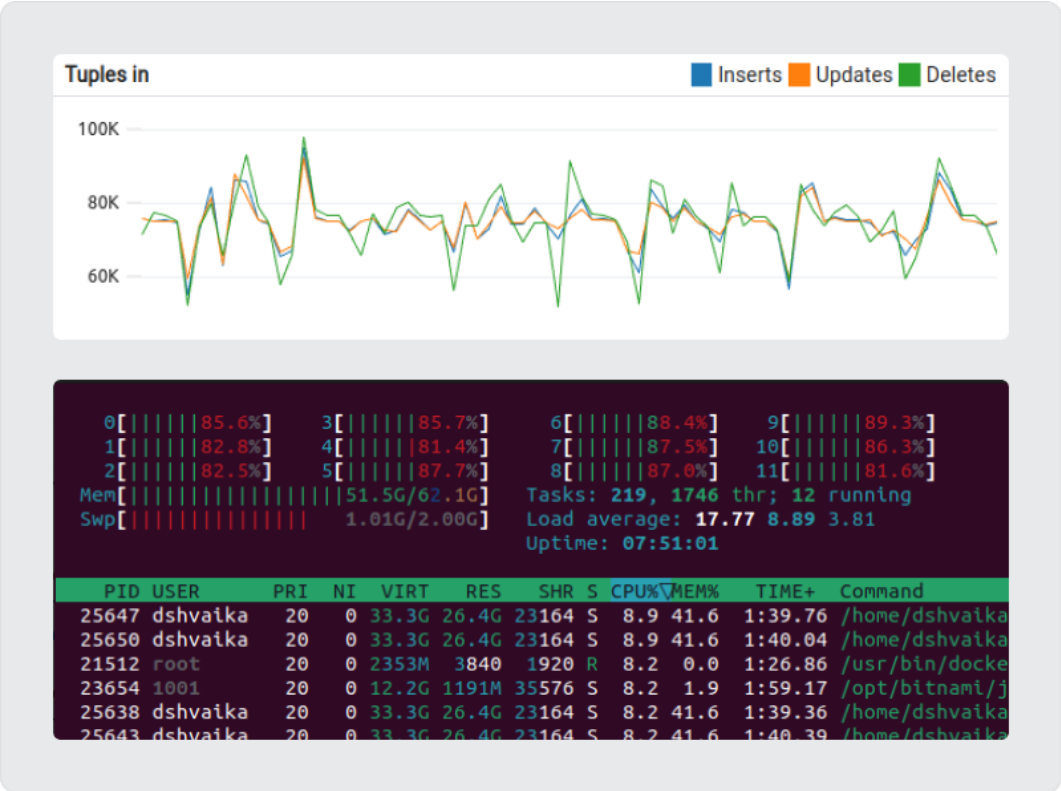

Figure 3. The graph reflects 75k tuples in/5s, corresponding to 15k msg/s for persistent MQTT clients, or half of the 30k msg/s throughput measured.

Based on the TimescaleDB blog post [35], vanilla PostgreSQL can handle up to 300k inserts per second under ideal conditions. However, this performance depends on factors such as hardware, workload, and table schema. While vertical scaling can provide some improvement, PostgreSQL's per-table insert throughput eventually reaches a hard limit. This experiment confirmed a fundamental scalability limit inherent to PostgreSQL's vertically scaled architecture. Although PostgreSQL has demonstrated strong performance in benchmark studies under concurrent read-write conditions [36], and has been used in large-scale industrial systems handling hundreds of thousands of transactions per second [37], its architecture is primarily optimized for single-node operation and lacks built-in horizontal scaling. As the number of persistent sessions in TBMQ deployments scaled into the millions, this model created a bottleneck. Confident in Redis's ability to overcome this bottleneck, the migration process was initiated to achieve greater scalability and efficiency. The migration process began with an evaluation of Redis data structures that could replicate the essential logic implemented with PostgreSQL.

2.3. Redis as a Scalable Alternative

The decision to migrate to Redis was driven by its ability to address the core performance bottlenecks encountered with PostgreSQL. Unlike PostgreSQL, which relies on disk-based storage and vertical scaling, Redis operates primarily in memory, significantly reducing read and write latency. Additionally, Redis's distributed architecture enables horizontal scaling, making it an ideal fit for high-throughput messaging in P2P communication scenarios [38]. Recent studies demonstrate the successful application of Redis in cloud and IoT scenarios, including enhancements through asynchronous I/O frameworks such as io_uring to further boost throughput under demanding conditions [39]. In addition, migration pipelines from traditional relational databases to Redis have been validated as efficient strategies for improving system responsiveness and scalability [40]. Figure 4 illustrates the updated architecture, where Redis replaces PostgreSQL as the persistence layer for standard MQTT clients.

 doi:10.20944/preprints202505.0445.v1

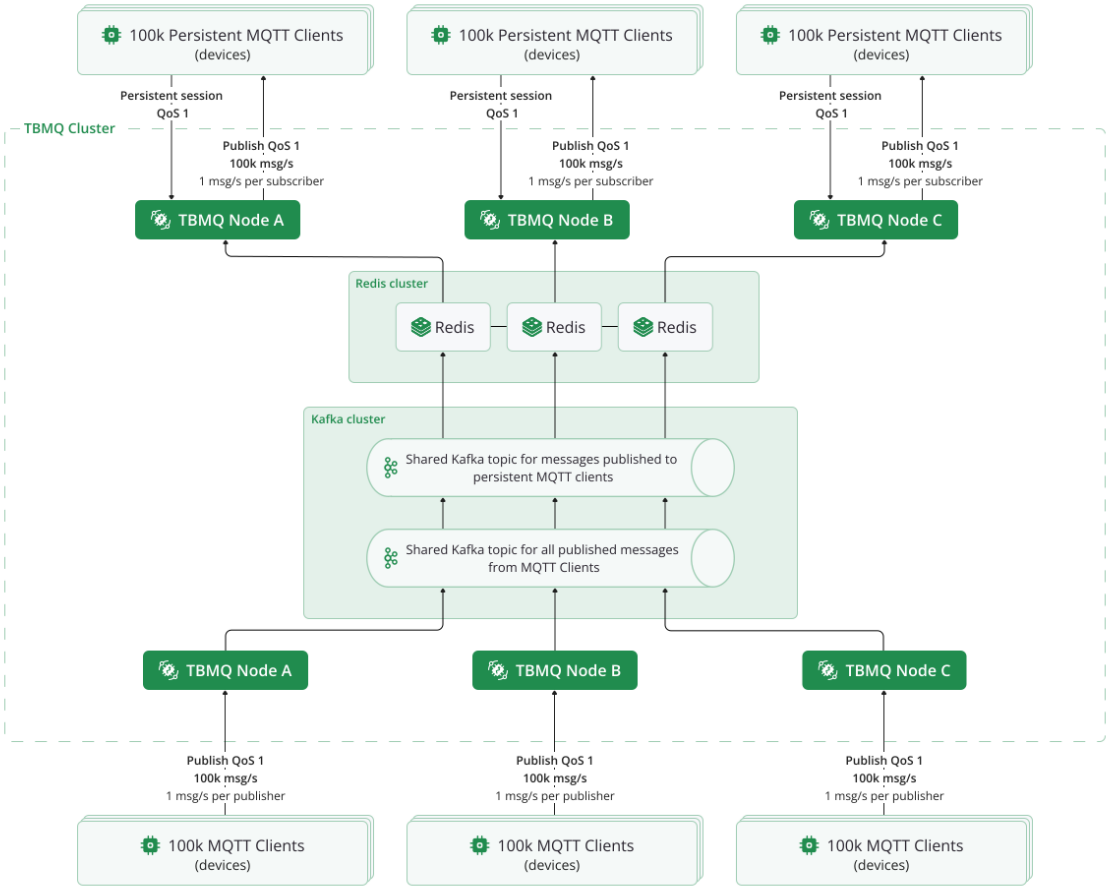

Figure 4. Updated TBMQ architecture using Redis for session persistence in high-throughput P2P messaging scenarios.

With these benefits in mind, the migration process was initiated by evaluating data structures capable of preserving the functionality of the PostgreSQL approach while aligning with Redis Cluster constraints to enable efficient horizontal scaling. This also presented an opportunity to improve certain aspects of the original design, such as periodic cleanups, by leveraging Redis features like built-in expiration mechanisms.

2.3.1. Redis Cluster Constraints

During the migration from PostgreSQL to Redis, it was identified that replicating the existing data model would require the use of multiple Redis data structures to efficiently handle message persistence and ordering. This, in turn, meant using multiple keys for each persistent MQTT Client session (see Figure 5)..

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 7 May 2025 doi:10.20944/preprints202505.0445.v1

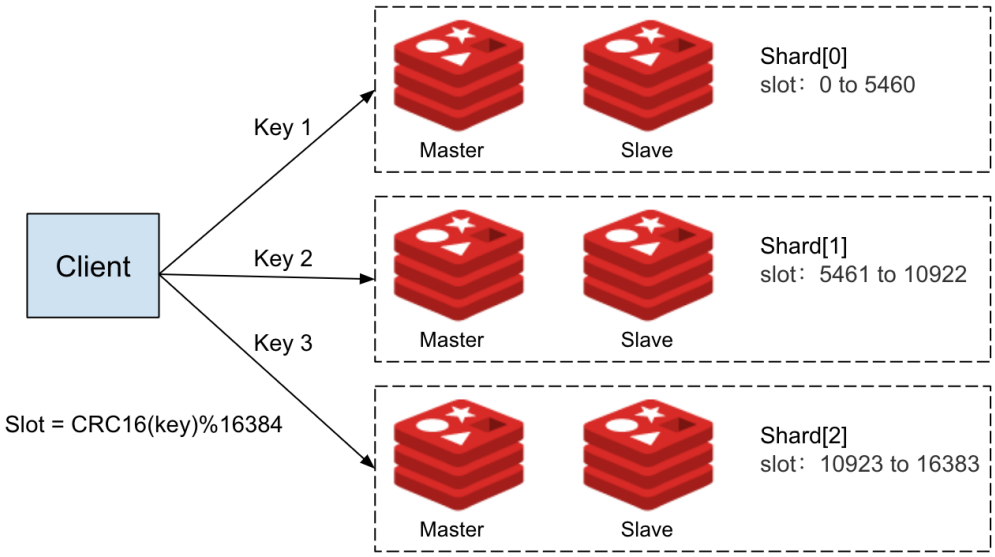

Figure 5. Redis Cluster slot-based sharding model. Each key is hashed to one of 16,384 slots and routed to the corresponding shard, [41]

Redis Cluster distributes data across multiple slots to enable horizontal scaling. However, multi-key operations must access keys within the same slot. If the keys reside in different slots, the operation triggers a cross-slot error, preventing the command from executing. The persistent MQTT client ID was embedded as a hash tag in key names to address this. By enclosing the client ID in curly braces , Redis ensures that all keys for the same client are hashed to the same slot. This guarantees that related data for each client stays together, allowing multi-key operations to proceed without errors.

2.3.2. Atomic Operations with Lua Scripting

Consistency is critical in a high-throughput environment like TBMQ, where many messages can arrive simultaneously for the same MQTT client. Hashtagging helps to avoid cross-slot errors, but without atomic operations, there is a risk of race conditions or partial updates. This could lead to message loss or incorrect ordering. It is important to make sure that operations updating the keys for the same MQTT client are atomic, Figure **??**.

While Redis ensures atomic execution of individual commands, updating multiple data structures for each MQTT client required additional handling. Executing these sequentially without atomicity opens the door to inconsistencies if another process modifies the same data in between commands. That's where Lua scripting comes in. Lua script executes as a single, isolated unit. During script execution, no other commands can run concurrently, ensuring that the operations inside the script happen atomically.

Based on this information, for operations such as saving messages or retrieving undelivered messages upon reconnection, a separate Lua script is executed. This ensures that all operations within a single Lua script reside in the same hash slot, maintaining atomicity and consistency.

2.3.3. Choosing the right Redis data structures

One of the key requirements of the migration was maintaining message order, a task previously handled by the `serial_number` column in PostgreSQL's `device_publish_msg` table. An evaluation of Redis data structures identified sorted sets (ZSETs) as the most suitable replacement.

Redis sorted sets naturally organize data by score, enabling quick retrieval of messages in ascending or descending order. While sorted sets provided an efficient way to maintain message order, storing full message payloads directly in sorted sets led to excessive memory usage. Redis does not support per-member TTL within sorted sets. As a result, messages persisted indefinitely unless explicitly removed. Periodic cleanups using `ZREMRANGEBYSCORE` were required, similar to the approach used in PostgreSQL, to remove expired messages. This operation carries a complexity of $O(\log N + M)$, where

 doi:10.20944/preprints202505.0445.v1

N is the number of elements in the set and M is the number of elements removed. To address this limitation, message payloads were stored in string data structures, while the sorted set maintained references to these string keys. Figure 6 illustrates this structure, `client_id` is a placeholder for the actual client ID, while the curly braces around it are added to create a hash tag.

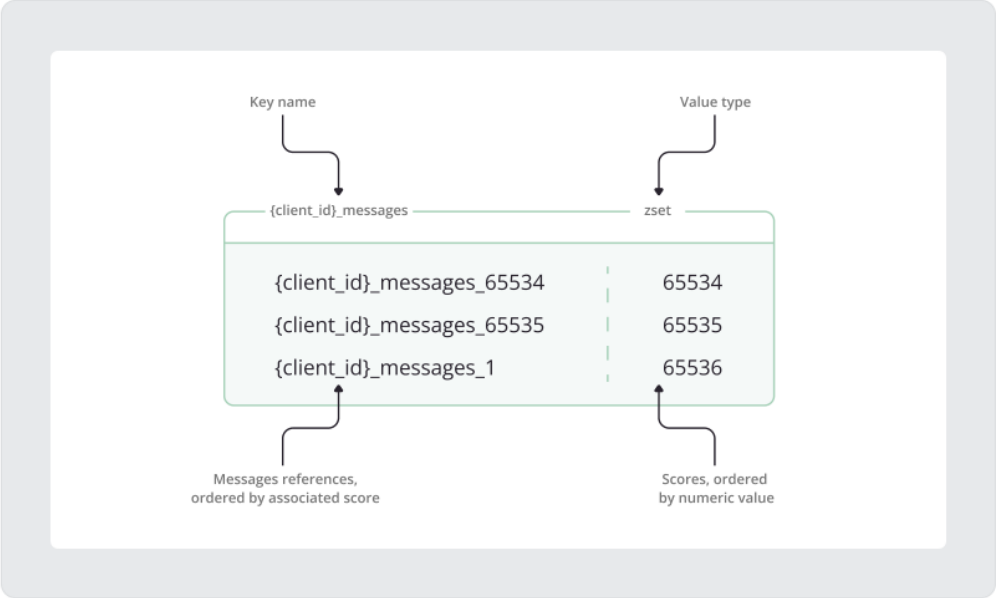

Figure 6. Redis sorted set structure used for MQTT message ordering.

In the image above, you can see that the score continues to grow even when the MQTT packet ID wraps around. Figure 6 illustrates the details illustrated in this image. At first, the reference for the message with the MQTT packet ID equal to 65534 was added to the sorted set:

```
ZADD {client_id}_messages 65534 {client_id}_messages_65534
```

Here, `client_id_messages` is the sorted set key name, where `client_id` acts as a hash tag derived from the persistent MQTT client's unique ID. The suffix `_messages` is a constant added to each sorted set key name for consistency. Following the sorted set key name, the score value 65534 corresponds to the MQTT packet ID of the message received by the client. Finally, the reference key links to the actual payload of the MQTT message. Similar to the sorted set key, the message reference key uses the MQTT client's ID as a hash tag, followed by the `_messages` suffix and the MQTT packet ID value.

In the following step, the message reference with a packet ID of 65535 is added to the sorted set. This is the maximum packet ID, as the range is limited to 65535.

```
ZADD {client_id}_messages 65535 {client_id}_messages_65535
```

Since the MQTT packet ID wraps around after 65535, the next message will receive a packet ID of 1. To preserve the correct sequence in the sorted set, the score is incremented beyond 65535 using the following Redis command:

```
ZADD {client_id}_messages 65536 {client_id}_messages_1
```

So at the next iteration MQTT packet ID should be equal to 1, while the score should continue to grow and be equal to 65536.

This approach ensures that the message's references will be properly ordered in the sorted set regardless of the packet ID's limited range.

Message payloads are stored as string values with SET commands that support expiration (EX), providing $O(1)$ complexity for writes and TTL applications:

 doi:10.20944/preprints202505.0445.v1


```
SET {client_id}_messages_1 "{
  \"packetType\":\"PUBLISH\",
  \"payload\":\"eyJkYXRhIjoidGJtcWlzYXdlc29tZSJ9\",
  \"time\":1736333110026,
  \"clientId\":\"client\",
  \"retained\":false,
  \"packetId\":1,
  \"topicName\":\"europe/ua/kyiv/client/0\",
  \"qos\":1
}" EX 600
```

Another benefit aside from efficient updates and TTL applications is that the message payloads can be retrieved:

```
GET {client_id}_messages_1
```

or removed:

```
DEL {client_id}_messages_1
```

with constant complexity $O(1)$ without affecting the sorted set structure.

Another very important element of Redis architecture for persistence message storage is the use of a string key to store the last MQTT packet ID processed:

```
GET {client_id}_last_packet_id "1"
```

This approach serves the same purpose as in the PostgreSQL solution. When a client reconnects, the server must determine the correct packet ID to assign to the next message that will be saved in Redis. An initial approach involved using the highest score in the sorted set as a reference. However, because scenarios may arise where the sorted set is empty or removed, storing the last packet ID separately was identified as the most reliable solution.

2.3.4. Managing Sorted Set Size Dynamically

This hybrid approach, leveraging sorted sets and string data structures, eliminates the need for periodic cleanups based on time, as per-message $TTLs$ are now applied. In addition, to remain consistent with the PostgreSQL design, it was necessary to implement cleanup of the sorted set based on the message limit defined in the configuration.

```
Maximum number of PUBLISH messages stored for each persisted DEVICE client
limit: "${MQTT_PERSISTENT_SESSION_DEVICE_PERSISTED_MESSAGES_LIMIT:10000}"
```

This limit is an important part of a design that allows control and prediction of the memory allocation required for each persistent MQTT client. For example, a client might connect, triggering the registration of a persistent session, and then rapidly disconnect. In such scenarios, it is essential to ensure that the number of messages stored for the client (while waiting for a potential reconnection) remains within the defined limit, preventing unbounded memory usage.

```
if (messagesLimit > 0xffff) {
   throw new IllegalArgumentException(
    "Persisted messages limit can't be greater than 65535!");
}
```

To reflect the natural constraints of the MQTT protocol, the maximum number of persisted messages for individual clients is set to 65535.

Dynamic management of the sorted set's size was introduced in the Redis implementation to address this requirement. When new messages are added, the sorted set is trimmed to ensure the total number of messages remains within the desired limit, and the associated strings are also cleaned up to free up memory.

 doi:10.20944/preprints202505.0445.v1


```
-- Get the number of elements to be removed
local numElementsToRemove = redis.call('ZCARD', messagesKey) - maxMessagesSize
-- Check if trimming is needed
if numElementsToRemove > 0 then
    -- Get the elements to be removed (oldest ones)
    local trimmedElements = redis.call('ZRANGE', messagesKey, 0, numElementsToRemove - 1)
    -- Iterate over the elements and remove them
    for _, key in ipairs(trimmedElements) do
        -- Remove the message from the string data structure
        redis.call('DEL', key)
        -- Remove the message reference from the sorted set
        redis.call('ZREM', messagesKey, key)
    end
end
```

2.3.5. Message Retrieval and Cleanup

The design not only ensures dynamic size management during the persistence of new messages but also supports cleanup during message retrieval, which occurs when a device reconnects to process undelivered messages. This approach keeps the sorted set clean by removing references to expired messages.

```
-- Define the sorted set key
local messagesKey = KEYS[1]
-- Define the maximum allowed number of messages
local maxMessagesSize = tonumber(ARGV[1])
-- Get all elements from the sorted set
local elements = redis.call('ZRANGE', messagesKey, 0, -1)
-- Initialize a table to store retrieved messages
local messages = {}
-- Iterate over each element in the sorted set
for _, key in ipairs(elements) do
    -- Check if the message key still exists in Redis
    if redis.call('EXISTS', key) == 1 then
        -- Retrieve the message value from Redis
        local msgJson = redis.call('GET', key)
        -- Store the retrieved message in the result table
        table.insert(messages, msgJson)
    else
        -- Remove the reference from the sorted set if the key does not exist
        redis.call('ZREM', messagesKey, key)
    end
end
-- Return the retrieved messages
return messages
```

By leveraging Redis' sorted sets and strings, along with Lua scripting for atomic operations, TBMQ achieves efficient message persistence and retrieval, as well as dynamic cleanup. This design addresses the scalability limitations of the PostgreSQL-based solution.

The following sections present the performance comparison between the new Redis-based architecture and the original PostgreSQL solution.

2.4. Migration from Jedis to Lettuce

As previously discussed, a prototype test revealed the limit of 30k msg/s throughput when using PostgreSQL for persistent message storage. At the time of the migration to Redis, the Jedis library was already in use for Redis interactions, primarily for cache management, and had been extended to handle message persistence for persistent MQTT clients. However, the initial results of the Redis implementation with Jedis were unexpected. Although Redis was expected to significantly outperform PostgreSQL, the performance improvement was modest, reaching only 40k msg/s throughput compared to the 30k msg/s limit with PostgreSQL (Figure 7).

 doi:10.20944/preprints202505.0445.v1

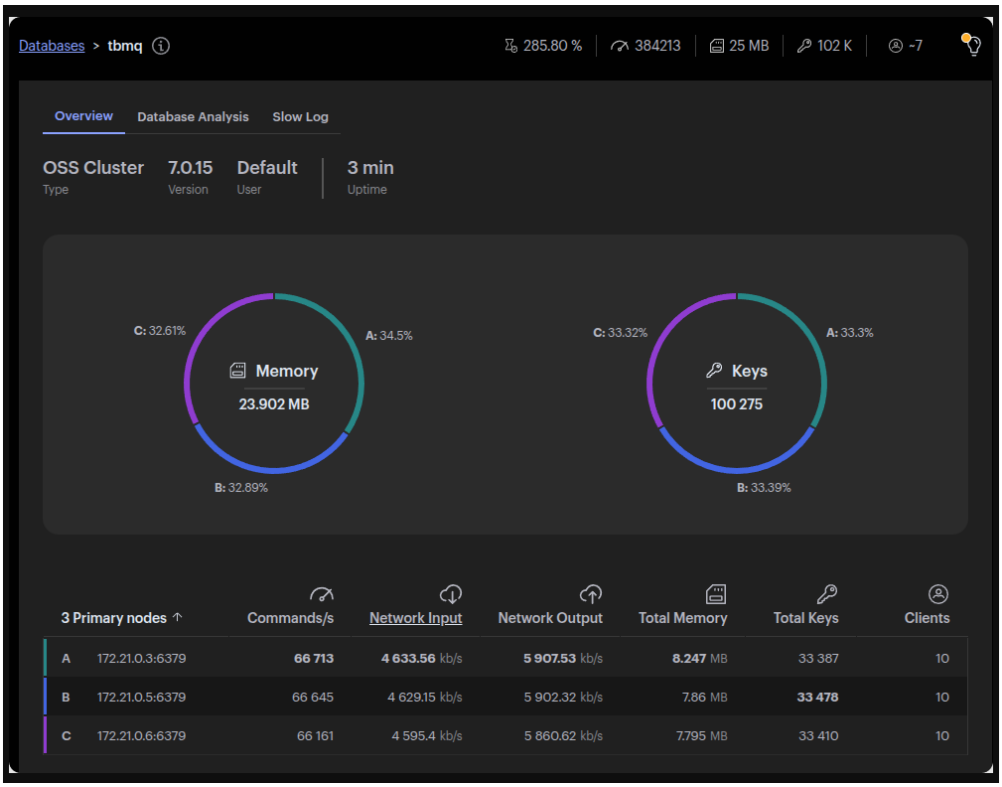

Figure 7. RedisInsight shows ~66k commands/s per node, aligning with TBMQ's 40k msg/s, as Lua scripts trigger multiple Redis operations per message.

This observation prompted an investigation into the bottlenecks, which revealed that Jedis was a limiting factor. While reliable, Jedis operates synchronously, processing each Redis command sequentially. This forces the system to wait for one operation to complete before executing the next. In high-throughput environments, this approach significantly limited Redis's potential, preventing the full utilization of system resources.

To overcome this limitation, the migration to Lettuce, an asynchronous Redis client built on top of Netty, was performed [42–44]. With Lettuce, throughput increased to 60k msg/s, demonstrating the benefits of non-blocking operations and improved parallelism, Figure 8

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 7 May 2025 doi:10.20944/preprints202505.0445.v1

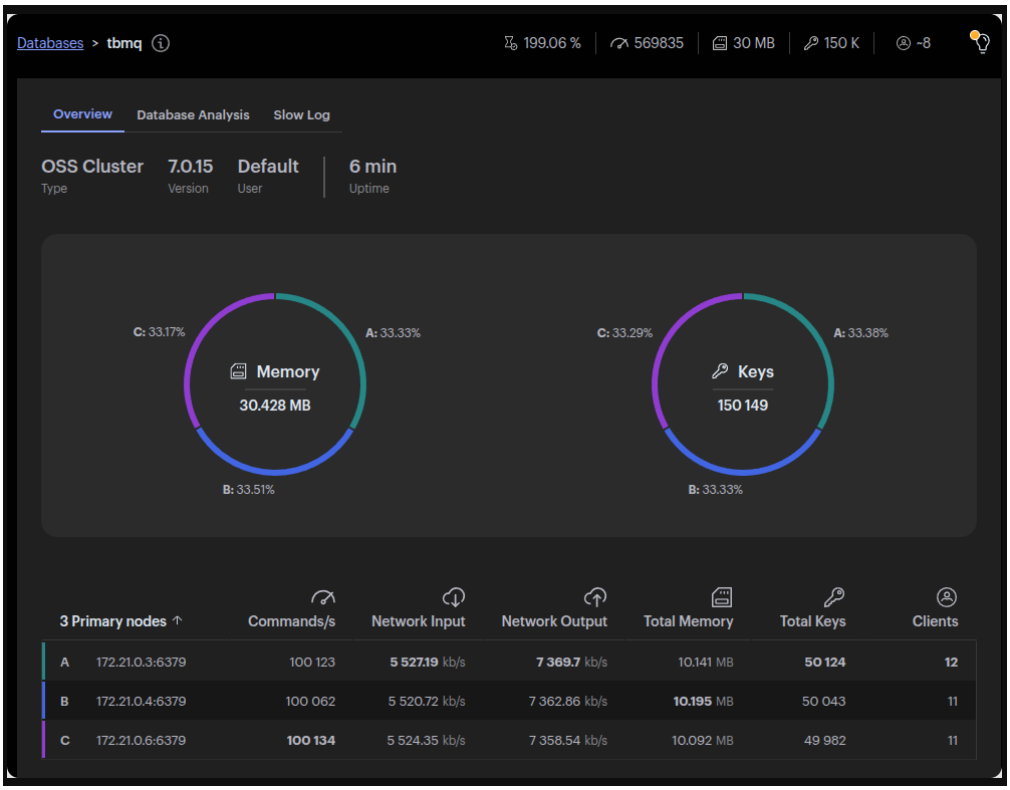

Figure 8. At 60k msg/s, RedisInsight shows ~100k commands/s per node, aligning with the expected increase from 40k msg/s, which produced ~66k commands/s per node.

Lettuce allows multiple commands to be sent and processed in parallel, fully exploiting Redis's capacity for concurrent workloads. Ultimately, the migration unlocked the expected performance gains from Redis, paving the way for successful P2P testing at scale.

2.5. Experimental Setup

With Redis and Lettuce fully integrated, the next challenge was ensuring TBMQ's ability to handle large-scale P2P messaging in a distributed environment. To simulate real-world conditions, TBMQ was deployed on AWS Elastic Kubernetes Service (EKS) [45], enabling dynamic scaling and stress testing of the system.

2.5.1. Test Methodology

To assess the TBMQ's ability to handle point-to-point communication at scale, five tests were conducted to measure performance, efficiency, and latency, with a maximum throughput of 1M msg/sec. Throughput refers to the total number of messages per second, including both incoming and outgoing messages. The performance test environment was deployed on an AWS EKS cluster and scaled horizontally as the workload increased. This allowed to evaluate how TBMQ handles growing demands while maintaining reliable performance.

Each test ran for 10 minutes, using an equal number of publishers and subscribers. Both publishers and subscribers operated with QoS 1, ensuring reliable message 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.

2.5.2. Test Agent Setup

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

 doi:10.20944/preprints202505.0445.v1

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 [45,46]. EC2 enables users to provision virtual machines with configurable CPU, memory, and network capacity, offering flexibility to handle varying workloads [47].

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 shows that 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. This flexible configuration enabled the test agent to adapt to rising traffic demands while addressing infrastructure constraints, such as port limitations.

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

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 [48].
- Redis Insight: For analyzing database performance [49].

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.

2.5.3. Infrastructure Overview

This section provides an overview of the test infrastructure, highlighting the hardware specifications of the services utilized in EKS cluster. EKS is a managed platform that simplifies the deployment and management of containers using the popular Kubernetes system [50]. In the 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 PostgreSQL database where the TBMQ stores different entities such as users, user credentials, MQTT client credentials, statistics, WebSocket connections, WebSocket subscriptions, and others.

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

 doi:10.20944/preprints202505.0445.v1

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.

2.5.4. Performance tests

To evaluate performance and prove that the system can scale efficiently, testing started with 200,000 msg/s, with the load increased by 200,000 messages in each iteration. In each phase, the number of TBMQ brokers and Redis nodes was scaled to handle the growing traffic while maintaining system stability. For the 1M msg/sec test, the number of Kafka brokers was also increased to accommodate the corresponding workload (Table 3).

Table 3. Scaling configuration for P2P throughput evaluation.

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

Beyond adding resources, each increase in load required fine-tuning of Kafka topic partitioning and Lettuce command batching parameters. These adjustments ensured even traffic distribution and stable latency, effectively preventing bottlenecks during system scaling (Table 4).

Table 4. Kafka and Redis tuning parameters at different throughput levels.

Throughput (msg/s)	Kafka Partitions	Lettuce Batch Size
200k	12	150
400k	12	250
600k	12	300
800k	16	400
1M	20	500

The target of 1 million msg/s was successfully reached, validating TBMQ's capability to support high-throughput, reliable P2P messaging. To better illustrate the test setup and results, the following diagram provides a visual breakdown of the final performance test, Figure 9.

 doi:10.20944/preprints202505.0445.v1

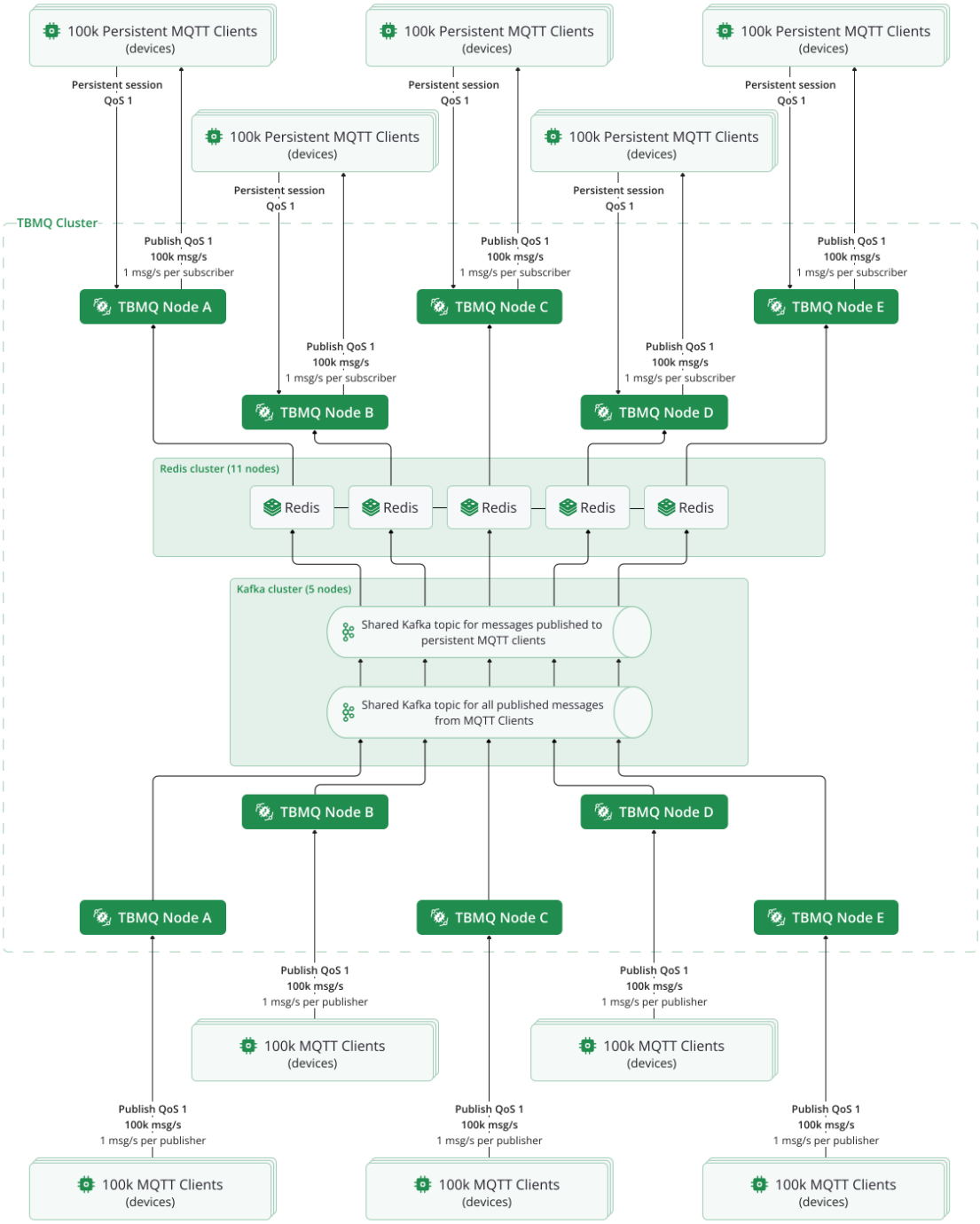

Figure 9. TBMQ architecture and traffic distribution during the 1 million msg/s test. Each TBMQ node handled 100k publishers and 100k subscribers using persistent MQTT sessions with QoS 1. Redis nodes provided session storage, while Kafka brokers handled message routing.

3. Results

Throughout the testing process, key performance indicators such as CPU utilization, memory usage, and message processing latency were continuously monitored. One of TBMQ's advantages, highlighted in P2P testing, is its exceptional messages-per-second per CPU core performance. Compared to public benchmarks of other brokers, TBMQ consistently delivers higher throughput with fewer resources, reinforcing its efficiency in large-scale deployments.

Key takeaways from tests include:

 doi:10.20944/preprints202505.0445.v1

- Scalability: TBMQ exhibited linear scalability, with reliable performance maintained as message throughput increased from 200k to 1M msg/s through the incremental addition of TBMQ nodes, Redis nodes, and Kafka nodes.
- 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 the test, applied to both publishers and persistent subscribers. The average acknowledgment latency for publishers was also tracked, 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. This was calculated by dividing the total throughput by the total number of CPU cores used in the setup.

Additionally, the following table provide a comprehensive summary of the key elements and results of the final 1M msg/sec test, Table 5:

Table 5. Summary of performance metrics at 1M msg/s throughput

QoS	P2P Latency (ms)	Publish Latency (ms)	TBMQ CPU Usage (avg)	Payload (bytes)
1	~75	~8	91%	62

TBMQ CPU usage: The average CPU utilization across all TBMQ nodes.

P2P latency: The average duration from when a PUB message is sent by the publisher to when it is received by the subscriber.

Publish latency: The average time elapsed between the PUB message sent by the publisher and the reception of the PUBACK acknowledgment.

Figure 10 demonstrates CPU utilization (%) of five managed nodes in the TBMQ cluster during the performance evaluation experiment. The peak load was observed at 16:40 UTC, with the highest value reaching 96.4% for one of the nodes. After this peak, the CPU utilization gradually decreased, demonstrating effective system stabilization after the peak period.

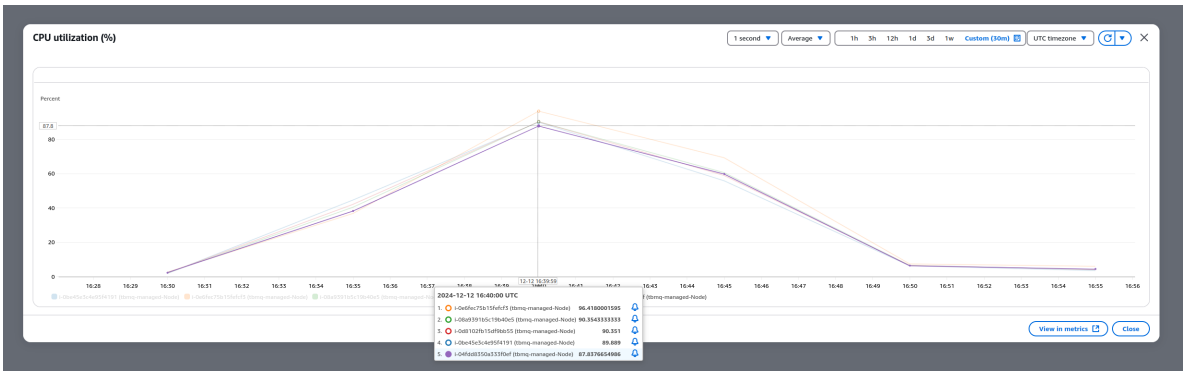

Figure 10. CPU utilization (%) of five managed nodes in the TBMQ cluster.

Figure 11 shows the Java Management Extensions (JMX) monitoring of the Central Processing Unit (CPU) confirming steady CPU load.

- The system handles the load effectively even under high activity levels (approximately 90% CPU usage).
- The absence of GC (Garbage Collection) activity confirms the stability and efficiency of Java Virtual Machine (JVM) performance during the tests.
- The current low CPU usage after the test completion indicates that the system quickly releases resources and returns to its normal operational state.

 doi:10.20944/preprints202505.0445.v1

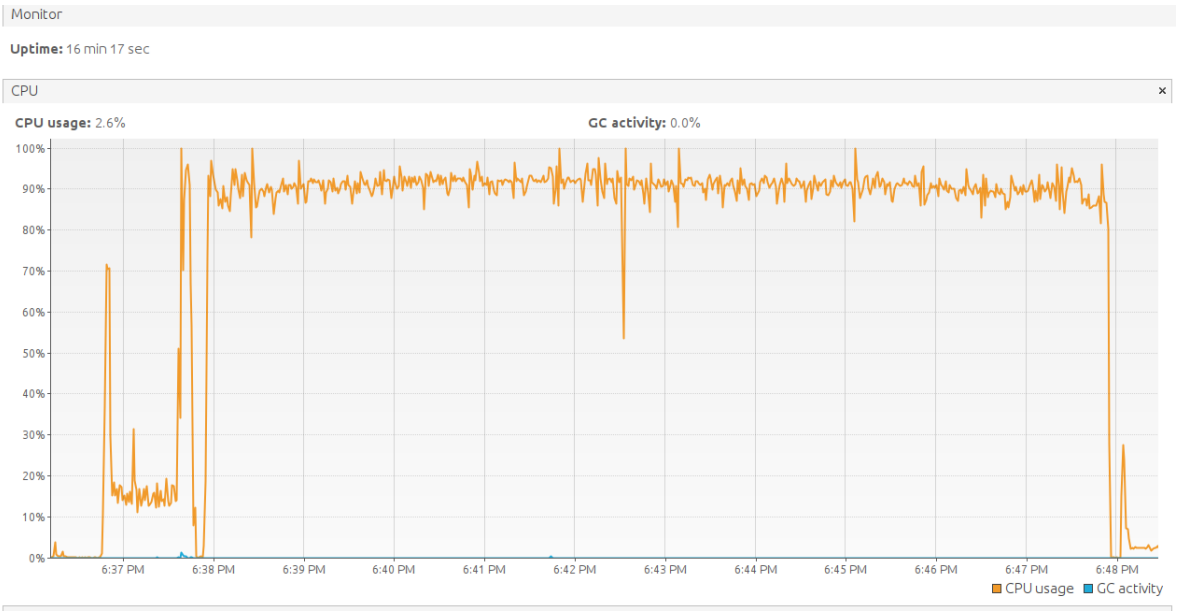

Figure 11. JMX monitoring of CPU on the TBMQ node.

Figure 12 shows the Java Management Extensions (JMX) monitoring of the RAM usage on one of the TBMQ nodes during the test, confirming no memory leaks after the warm-up period.

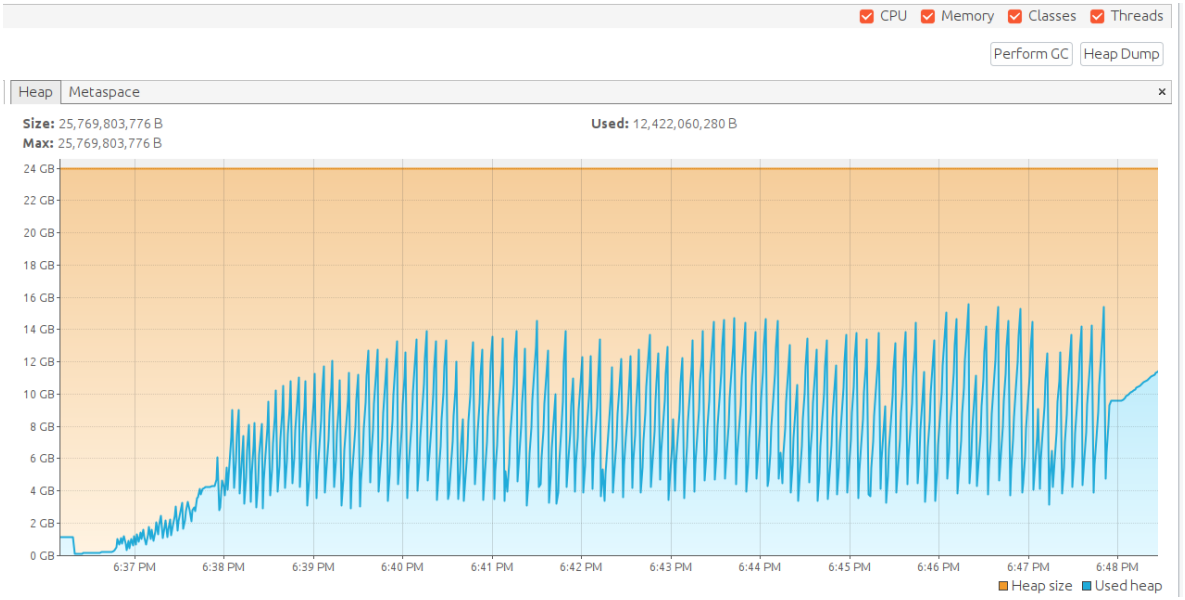

Figure 12. JMX monitoring of RAM on the TBMQ node.

- Initial Growth of Heap Memory: The initial increase in heap memory usage (from 2 GB to 12 GB) indicates the start of the performance test and the allocation of memory required for handling the workload.
- Cyclic Memory Usage Patterns: Following stabilization, cyclic patterns in memory usage are observed, reflecting the regular activity of the Garbage Collection (GC) process. GC effectively frees unused memory, maintaining a stable memory footprint.
- No Memory Overflow: The total allocated heap size (25 GB) was sufficient, as the memory usage never reached a critical threshold, preventing any OutOfMemory errors.
- Stable Performance Under Load: The consistent memory usage patterns without significant spikes confirm the system's ability to handle high workloads efficiently while maintaining GC effectiveness.

 doi:10.20944/preprints202505.0445.v1

- Return to Baseline: After test completion, memory usage gradually decreased, demonstrating the system's capability to release resources promptly and return to its baseline state.

These results highlight the system's optimized JVM configuration and heap management, ensuring reliable performance under intensive load conditions.

For a deeper dive into the testing architecture, methodology, and results, check out the detailed performance testing article [51].

4. Discussion

The results demonstrate that TBMQ achieves linear scalability and low-latency message delivery in large-scale P2P scenarios, reaching throughput levels of 1 million messages per second while maintaining approximately 8900 messages per second per CPU core and 2-digit bound millisecond end-to-end delivery latencies. The Redis-based session persistence layer enabled efficient handling of persistent MQTT sessions, while Kafka provided reliable backend message durability and routing.

Compared to EMQX and HiveMQ approaches, TBMQ shows clear advantages in infrastructure cost efficiency and architectural simplicity. While EMQX and HiveMQ demonstrate substantial progress in scaling MQTT brokers using one-to-one communication patterns, the underlying persistence architectures and infrastructure requirements differ significantly from those of TBMQ. Both EMQX and HiveMQ rely primarily on disk-based persistence (RocksDB and file-based storage, respectively) to ensure high data durability. In contrast, TBMQ leverages an in-memory persistence model via Redis Cluster, combined with Kafka-based routing, to prioritize low-latency delivery and infrastructure cost efficiency.

Table 6 summarizes the architectural differences between TBMQ, EMQX, and HiveMQ.

Table 6. Comparison of Persistence Models and Infrastructure Focus Across MQTT Brokers.

Feature	TBMQ	EMQX	HiveMQ
Primary Storage	Redis (in-memory)	RocksDB (disk)	File system (disk)
Latency	Very low	Moderate (SSD-dependent)	Moderate (SSD-dependent)
Crash Durability	Medium (depends on Redis snapshot/AOF)	High (RocksDB durability)	Very High (file persistence)
Memory Usage	High	Lower	Lower
Persistence Overhead	Minimal (in-memory ops)	Higher (disk writes)	Higher (disk writes)

Compared to previous approaches, such as EMQX and HiveMQ, TBMQ shows clear advantages in infrastructure cost efficiency and architectural simplicity. These suggest that TBMQ provides a compelling alternative for edge-oriented IoT deployments where minimizing latency and operational overhead is critical. The Redis-based model efficiently serves real-time messaging needs, while Kafka acts as a durable safety net, capturing message streams even in the event of transient Redis issues.

However, several limitations should be acknowledged. First, while Redis enables extremely low-latency operations, its volatility compared to disk-based storage remains a factor to be carefully managed through persistence policies (AOF, snapshots) and cluster replication. Second, the performance evaluation focused on typical IoT message sizes; extremely large payloads (exceeding 1 MB) were not tested. Such workloads could introduce additional memory pressure and might require architecture adjustments, although these cases are uncommon in practice for P2P IoT scenarios.

Future work could focus on optimizing Redis utilization further by adjusting Lua scripting strategies. Currently, Lua scripts operate per client session to comply with Redis Cluster's slot boundaries. By grouping multiple clients into the same hash slot, batch processing could be achieved, reducing scripting overhead and improving Redis efficiency. Additionally, exploring dynamic payload-aware routing strategies between Redis and Kafka could further enhance TBMQ's flexibility and performance for a broader range of IoT messaging workloads.

 doi:10.20944/preprints202505.0445.v1

5. Conclusions

This study presented the architectural transformation of TBMQ, an open-source MQTT broker, to support high-throughput, low-latency point-to-point (P2P) messaging at scale. By migrating from a PostgreSQL-backed persistence layer to a horizontally scalable Redis-based architecture, and optimizing access using Lettuce and Lua scripting, TBMQ overcomes critical limitations faced by traditional MQTT brokers under high-concurrency loads. The redesigned architecture effectively separates session state management and message routing concerns, using Redis and Kafka respectively to achieve both performance and reliability.

The results of comprehensive performance tests—reaching 1 million messages per second while maintaining sub-100ms end-to-end latency and 8900 msg/s per CPU core—demonstrate that the new architecture provides true horizontal scalability without compromising QoS guarantees. The system maintained high CPU efficiency, stable memory usage, and consistent delivery guarantees even under sustained peak load.

The findings confirm that TBMQ can serve as a robust and scalable foundation for next-generation IoT deployments, particularly those requiring reliable P2P communication between massive numbers of connected devices. This work also contributes to the broader research on distributed messaging systems by demonstrating a viable, cloud-native, multi-layered persistence model for MQTT brokers.

Future work will focus on further performance optimizations such as cross-session batching in Redis and tighter integration with external systems. One promising direction involves enhancing embedded integration support within TBMQ, enabling direct traffic routing from IoT devices to various platforms without intermediate transformation. This capability opens the door to advanced features like dynamic payload serialization (e.g., Protocol Buffers), which can be leveraged to reduce overhead and improve interoperability. Preliminary considerations for such enhancements are aligned with recent research on dynamic data serialization in IoT platforms [52].

Author Contributions: Conceptualization, A.S.; methodology, V.A.; software, D.S.; validation, A.S. and D.S.; formal analysis, D.S.; investigation, D.S.; resources, A.S.; writing—original draft preparation, D.S.; writing—review and editing, A.S. and V.A.; visualization, D.S.; supervision, V.A.; project administration, A.S.; funding acquisition, A.S. All authors have read and agreed to 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. The performance test tool used in this study is available at https://github.com/thingsboard/tb-mqtt-perf-tests. All test configurations and datasets used for benchmarking can be reproduced using the provided scripts and documentation.

Acknowledgments: The authors would like to thank the team at ThingsBoard, Inc. for their support in the development and evaluation of the TBMQ platform. During the preparation of this manuscript, the authors used ChatGPT-4 (OpenAI, 2024) and Grammarly for grammar and spelling checks, sentence refinement, and improving overall clarity. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors are employees of ThingsBoard, Inc., which supported this research by providing access to infrastructure and computing resources. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

 doi:10.20944/preprints202505.0445.v1

Abbreviations

The following abbreviations are used in this manuscript:

AWS	Amazon Web Services
CPU	Central Processing Unit
EKS	Elastic Kubernetes Service
IoT	Internet of Things
JMX	Java Management Extensions
MQTT	Message Queuing Telemetry Transport
P2P	Point-to-Point
pub/sub	Publish/Subscribe
QoS	Quality of Service
RAM	Random Access Memory
TCP	Transmission Control Protocol

References

1. Zhang, W.; Liu, J. Advanced Quality of Service Approaches in Edge IoT Communication: A Review. *IoT* **2023**, *4*, 123–138. https://doi.org/10.3390/iot4020009.
2. Fernandez, C.; Rahman, F. SDN-Based Dynamic Routing Architectures for Scalable IoT Communication. *IoT* **2022**, *3*, 55–70. https://doi.org/10.3390/iot3010005.
3. Ali, S.; Wang, R. Blockchain-Based Trust Models for Secure IoT Communication. *IoT* **2023**, *4*, 250–267. https://doi.org/10.3390/iot4030018.
4. Singh, A.; Becker, T. Distributed Learning and Optimization Strategies for Resource-Aware IoT Systems. *IoT* **2023**, *4*, 311–327. https://doi.org/10.3390/iot4040025.
5. Di Felice, P.; Paolone, G. Papers Mentioning Things Board: A Systematic Mapping Study. *Journal of Computer Science* **2024**, *20*, 574–584. https://doi.org/10.3844/jcssp.2024.574.584.
6. ThingsBoard. ThingsBoard IoT Platform. https://thingsboard.io, 2016. Accessed: March 2024.
7. Aghenta, L.O.; Iqbal, M.T. Design and implementation of a low-cost, open source IoT-based SCADA system using ESP32 with OLED, ThingsBoard and MQTT protocol. *AIMS Electronics and Electrical Engineering* **2019**, *4*, 57 – 86. https://doi.org/10.3934/ElectrEng.2020.1.57.
8. Bestari, D.N.; Wibowo, A. An IoT-Based Real-Time Weather Monitoring System Using Telegram Bot and Thingsboard Platform. *International Journal of Interactive Mobile Technologies* **2023**, *17*, 4 – 19. https://doi.org/10.3991/ijim.v17i06.34129.
9. De Paolis, L.T.; De Luca, V.; Paiano, R. Sensor data collection and analytics with thingsboard and spark streaming. Institute of Electrical and Electronics Engineers Inc., 2018, p. 1 – 6. https://doi.org/10.1109/EESMS.2018.8405822.
10. Casillo, M.; Colace, F.; De Santo, M.; Lorusso, A.; Mosca, R.; Santaniello, D. VIOTLab: A Virtual Remote Laboratory for Internet of Things Based on ThingsBoard Platform. Institute of Electrical and Electronics Engineers Inc., 2021, Vol. 2021-October. https://doi.org/10.1109/FIE49875.2021.9637317.
11. Jang, S.I.; Kim, J.Y.; Iskakov, A.; Fatih Demirci, M.; Wong, K.S.; Kim, Y.J.; Kim, M.H. Blockchain Based Authentication Method for ThingsBoard. *Lecture Notes in Electrical Engineering* **2021**, *715*, 471 – 479. https://doi.org/10.1007/978-981-15-9343-7_65.
12. Okhovat, E.; Bauer, M. Monitoring the Smart City Sensor Data Using Thingsboard and Node-Red. Institute of Electrical and Electronics Engineers Inc., 2021, p. 425 – 432. https://doi.org/10.1109/SWC50871.2021.00064.
13. Team, A.I.C. MQTT Communication Patterns: One-to-Many (Broadcast). https://docs.aws.amazon.com/whitepapers/latest/designing-mqtt-topics-aws-iot-core/mqtt-communication-patterns.html#broadcast, 2023. Accessed: May 2025.
14. Team, A.I.C. MQTT Communication Patterns: Many-to-One (Fan-In). https://docs.aws.amazon.com/whitepapers/latest/designing-mqtt-topics-aws-iot-core/mqtt-communication-patterns.html#fan-in, 2023. Accessed: May 2025.
15. ThingsBoard. TBMQ Release Notes v2.0.1 (December 31, 2024). https://thingsboard.io/docs/mqtt-broker/releases/#v201-december-31-2024, 2022. Accessed: May 2025.
16. Spohn, M.A. On MQTT Scalability in the Internet of Things: Issues, Solutions, and Future Directions. *Journal of Electronics and Electrical Engineering* **2022**, *1*, 1–11. https://doi.org/10.37256/jeee.1120221687.

 doi:10.20944/preprints202505.0445.v1

17. Hmissi, F.; Ouni, S. SDN-DMQTT: SDN-Based Platform for Re-configurable MQTT Distributed Brokers Architecture. In Proceedings of the Mobile and Ubiquitous Systems: Computing, Networking and Services; Zaslavsky, A.; Ning, Z.; Kalogeraki, V.; Georgakopoulos, D.; Chrysanthis, P.K., Eds., Cham, 2024; pp. 393–411. https://doi.org/10.1007/978-3-031-63992-0_26.
18. Akour, M.; Rousan, I.A.; Hassanein, H. Multi-level Just-Enough Elasticity for MQTT Brokers of Internet of Things Applications. *Cluster Computing* **2022**, *25*, 1025–1045. https://doi.org/10.1007/s10586-022-03636-w.
19. Mishra, B., Performance Evaluation of MQTT Broker Servers; 2018; pp. 599–609. https://doi.org/10.1007/978-3-319-95171-3_47.
20. Gupta, A. Processing Time-Series Data with Redis and Apache Kafka. https://redis.io/blog/processing-time-series-data-with-redis-and-apache-kafka/, 2021. Accessed: 2025-05-03.
21. Team, E. MQTT and Redis: Creating a Real-Time Data Statistics Application for IoT. https://www.emqx.com/en/blog/mqtt-and-redis, 2023. Accessed: 2025-05-03.
22. Mishra, B.; Kertesz, A. The Use of MQTT in M2M and IoT Systems: A Survey. *IEEE Access* **2020**, *8*, 201071–201086. https://doi.org/10.1109/ACCESS.2020.3035849.
23. Ma, Z.; Yan, M.; Wang, R.; Wang, S. Performance Analysis of P2P Network Content Delivery Based on Queueing Model. *Cluster Computing* **2023**, *27*. https://doi.org/10.1007/s10586-023-04111-w.
24. Shen, Y.; Ma, Z. The Analysis of P2P Networks with Malicious Peers and Repairable Breakdown Based on Geo/Geo/1+1 Queue. *Journal of Parallel and Distributed Computing* **2025**, *195*, 104979. https://doi.org/10.1016/j.jpdc.2024.104979.
25. Dong, B.; Chen, J. An Adaptive Routing Strategy in P2P-Based Edge Cloud. *Journal of Cloud Computing* **2024**, *13*. https://doi.org/10.1186/s13677-023-00581-w.
26. Dinculeană, D.; Cheng, X. Vulnerabilities and Limitations of MQTT Protocol Used Between IoT Devices. *Applied Sciences* **2019**, *9*. https://doi.org/10.3390/app9050848.
27. Shvaika, D.I.; Shvaika, A.I.; Landiak, D.I.; Artemchuk, V.O. Scalable and reliable MQTT messaging: Evaluating TBMQ for P2P scenarios. In Proceedings of the CEUR Workshop Proceedings. CEUR-WS.org, 2025, pp. 58–66. https://ceur-ws.org/Vol-3943/paper12.pdf, Accessed: 2025-05-03.
28. ThingsBoard. TBMQ 1.0: 100 Million Connections Performance Test. https://thingsboard.io/docs/mqtt-broker/reference/100m-connections-performance-test/, 2022. Accessed: May 2025.
29. ThingsBoard. TBMQ 1.0: 3 Million Messages Per Second Throughput on a Single Node. https://thingsboard.io/docs/mqtt-broker/reference/3m-throughput-single-node-performance-test/, 2022. Accessed: May 2025.
30. Apache Software Foundation. Apache Kafka Documentation. https://kafka.apache.org/documentation, 2025. Accessed: April 2025.
31. Vyas, S.; Tyagi, R.K.; Jain, C.; Sahu, S. Performance Evaluation of Apache Kafka – A Modern Platform for Real Time Data Streaming. In Proceedings of the 2022 2nd International Conference on Innovative Practices in Technology and Management (ICIPTM), 2022, Vol. 2, pp. 465–470. https://doi.org/10.1109/ICIPTM54933.2022.9754154.
32. Park, S.; Huh, J.H. A Study on Big Data Collecting and Utilizing Smart Factory Based Grid Networking Big Data Using Apache Kafka. *IEEE Access* **2023**, *11*, 96131–96142. https://doi.org/10.1109/ACCESS.2023.3305586.
33. Elshoubary, E.E.; Radwan, T. Studying the Efficiency of the Apache Kafka System Using the Reduction Method, and Its Effectiveness in Terms of Reliability Metrics Subject to a Copula Approach. *Applied Sciences* **2024**, *14*. https://doi.org/10.3390/app14156758.
34. ThingsBoard Inc.. TBMQ Performance Tests: P2P Messaging Benchmark Suite. https://github.com/thingsboard/tb-mqtt-perf-tests/tree/p2p-perf-test, 2024. Accessed: 2025-05-03.
35. Timescale Team. PostgreSQL + TimescaleDB: 1000x Faster Queries, 90% Data Compression, and Much More. https://www.timescale.com/blog/postgresql-timescaledb-1000x-faster-queries-90-data-compression-and-much-more, 2018. Accessed: May 2025.
36. Salunke, S.; Ouda, A. A Performance Benchmark for the PostgreSQL and MySQL Databases. *Future Internet* **2024**, *16*, 382. https://doi.org/10.3390/fi16100382.
37. Ünal, H.T.; Mendi, A.F.; Mete, S.; Ömer Özkan.; Özgür Umut Vurgun.; Nacar, M.A. PostgreSQL Database Management System: ODAK. In Proceedings of the 2023 Innovations in Intelligent Systems and Applications Conference (ASYU). IEEE, 2023, pp. 1–6. https://doi.org/10.1109/ASYU58738.2023.10296600.
38. Redis. Redis Pub/Sub. https://redis.io/docs/latest/develop/interact/pubsub/, 2024. Accessed: April 2025.

 doi:10.20944/preprints202505.0445.v1

39. Chen, L.G.; Li, Y.; Laohakangvalvit, T.; Sugaya, M. Asynchronous I/O Persistence for In-Memory Database Servers: Leveraging io_uring to Optimize Redis Persistence. In Proceedings of the CLOUD Computing – CLOUD 2024; Wang, Y.; Zhang, L.J., Eds., Cham, 2025; pp. 11–20.
40. Muradova, G.; Hematyar, M.; Jamalova, J. Advantages of Redis in-memory database to efficiently search for healthcare medical supplies using geospatial data. In Proceedings of the 2022 IEEE 16th International Conference on Application of Information and Communication Technologies (AICT), 2022, pp. 1–5. https://doi.org/10.1109/AICT55583.2022.10013544.
41. Redis Lua Scripting Documentation. https://redis.io/docs/latest/commands/eval/, 2024. Accessed: April 2025.
42. Lettuce Project Contributors. Lettuce - Scalable Redis Client. https://lettuce.io/, 2024. Accessed: April 2025.
43. Lee, T. Netty: Asynchronous Event-Driven Network Application Framework. *Netty.io* **2021**. Accessed: April 2025.
44. Maurer, N.; Wolfthal, M.A. Netty in Action. 2015.
45. AWS. AWS. https://aws.amazon.com/, 2023. Accessed: April 2025.
46. Wittig, A.; Wittig, M. *Amazon Web Services in Action: An in-depth guide to AWS*; Simon and Schuster, 2023.
47. ThingsBoard Inc.. How to Repeat the 1M msg/sec Throughput Test. https://thingsboard.io/docs/mqtt-broker/reference/1m-throughput-p2p-performance-test/#how-to-repeat-the-1m-msgsec-throughput-test, 2024. Accessed: 2025-05-03.
48. Redpanda Data. Redpanda Console: A Developer-Friendly UI for Kafka. https://www.redpanda.com/redpanda-console-kafka-ui, 2024. Accessed: 2025-05-03.
49. Redis Ltd.. RedisInsight: Developer Tool for Managing Redis. https://redis.io/docs/latest/operate/redisinsight/, 2024. Accessed: 2025-05-03.
50. Kubernetes. Kubernetes documentation. https://kubernetes.io/, 2023. Accessed: April 2025.
51. ThingsBoard. Scaling P2P Messaging to 1M Msg/sec with Persistent MQTT Clients. https://thingsboard.io/docs/mqtt-broker/reference/1m-throughput-p2p-performance-test/, 2022.
52. Shvaika, D.I.; Shvaika, A.I.; Artemchuk, V.O. Advancing IoT interoperability: dynamic data serialization using ThingsBoard. *Journal of Edge Computing* **2024**, *3*, 126–135.