T.C.
MANİSA CELAL BAYAR UNIVERSITY
GRADUATE SCHOOL OF

MASTER's THESIS
DEPARTMENT OF COMPUTER ENGINEERING

A Comprehensive Comparison of Open-Source MQTT Brokers in Terms of Architectural, Technical, and Functional Characteristics

Gürkan GÜNEY

Supervisor
Doç. Dr. Övünç ÖZTÜRK

MANİSA-2025

GÜRKAN GÜNEY

A Comprehensive Comparison of Open-Source MQTT Brokers in Terms of Architectural, Technical, and Functional Characteristics

2025

COMMITMENT

I declare that this thesis was written in Manisa Celal Bayar University, Faculty of Engineering, Department of Computer Engineering in accordance with academic and ethical rules, and all the literature information used is included in the thesis with reference.

Gürkan GÜNEY

ÖZET

Yüksek Lisans Tezi

Gürkan GÜNEY
Manisa Celal Bayar Üniversitesi
Lisansüstü Eğitim Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı

Danışman:
Doç. Dr Övünç ÖZTÜRK

Bu tez, MQTT protokolünü kullanan açık kaynak broker yazılımlarının (Mosquitto, EMQX ve RabbitMQ) performans, güvenlik ve özellik bakımından karşılaştırılmasını amaçlamaktadır. Çalışmada, her broker için benzer test senaryoları uygulanarak; kimlik doğrulama, konu yönlendirme (topic routing), hizmet kalitesi (QoS), gecikme (latency), bant genişliği (throughput) ve sistem kararlılığı (stability) gibi çeşitli metrikler üzerinden analiz yapılmıştır.

Tez üç ana bölümden oluşmaktadır. İlk bölümde MQTT protokolünün temel prensipleri ve broker mimarisi anlatılmıştır. İkinci bölümde test ortamı ve kullanılan yöntem detaylandırılmış, üçüncü bölümde ise elde edilen sonuçlar karşılaştırmalı olarak sunulmuştur. Test ortamı, kaynak sınırlandırmaları yapılmış Docker konteynerlerinde yapılandırılmış ve tüm brokerlar eşit şartlar altında değerlendirilmiştir.

Sonuç olarak, her brokerın farklı kullanım senaryoları için avantaj ve dezavantajları olduğu görülmüştür. Mosquitto düşük kaynak tüketimi ile öne çıkarken, EMQX geniş protokol desteği ve stabil çalışmasıyla dikkat çekmiştir. RabbitMQ ise mesaj yönlendirme ve QoS gibi alanlarda bazı sınırlamalara sahip olsa da, güçlü kuyruk yönetimi altyapısıyla belirli uygulamalar için tercih edilebilir bir seçenek sunmaktadır.

Anahtar Kelimeler: MQTT, Mesaj Aracısı, Performans Testi, Nesnelerin İnterneti İletişimi, Açık Kaynak Yazılım

2025, 83 sayfa

ABSTRACT

M.Sc. Thesis

Gürkan GÜNEY

Manisa Celal Bayar University
Graduate School of Education
Department of Computer Engineering

Supervisor:
Assoc. Prof. Dr. Övünç ÖZTÜRK

This thesis aims to compare open-source MQTT broker software (Mosquitto, EMQX, and RabbitMQ) in terms of performance, security, and feature sets. In this study, a series of test scenarios were applied to each broker to analyze various metrics such as authentication, topic routing, Quality of Service (QoS), latency, throughput, and system stability.

The thesis consists of three main sections. The first section presents the fundamental principles of the MQTT protocol and broker architecture. The second section details the testing environment and methodology. In the third section, the obtained results are presented and compared. The test environment was configured using Docker containers with resource limitations applied equally across all brokers to ensure a fair evaluation.

As a result, it was observed that each broker has distinct advantages and disadvantages depending on the intended use case. Mosquitto stands out with its low resource consumption, while EMQX demonstrates robust performance and broad protocol support. Although RabbitMQ shows certain limitations in areas such as topic routing and QoS, it offers a strong queuing infrastructure that may be suitable for specific applications.

Keywords: MQTT, Message Broker, Performance Testing, IoT Communication, Open-Source Software

2025, 83 pages

ACKNOWLEDGEMENTS

This thesis focuses on a comprehensive comparison of the architectural, technical, and functional features of open-source MQTT brokers. Within the scope of the study, Mosquitto, EMQX, and RabbitMQ brokers were tested for performance criteria such as authentication, topic routing, QoS levels, latency, and data transfer rate.

The first section of the thesis describes the purpose, scope, and methodology of the study. The second section discusses the MQTT protocol and the technical infrastructure of the examined brokers. The third section details the test environment, methods used, and test scenarios. The fourth section presents the test results, and the fifth section analyzes and evaluates the findings. The final section includes the conclusions and potential future work.

I extend my sincere gratitude to my advisor, Assoc. Prof. Dr. Övünç Öztürk, who provided guidance with his knowledge and experience at every stage of this study and provided invaluable contributions and support, and to my esteemed instructor, Prof. Dr. Tuğba Özacar Öztürk, for her valuable contributions. They have played a major role in the emergence of this thesis with their patient approach, constructive criticism and academic guidance.

Gürkan GÜNEY
Manisa, 2025

LIST OF ABBREVIATIONS

MQTT	Message Queuing Telemetry Transport
IoT	Internet of Things
QoS	Quality of Service
LWT	Last Will and Testament
IIoT	Industrial Internet of Things
LAN	Local Area Network
WAN	Wide Area Network
CPU	Central Processing Units
I/O	Input/Output
RAM	Random Access Memory
MB	Megabayt
TCP	Transmission Control Protocol
IP	Internet Protocol
LDAP	Lightweight Directory Access Protocol
TLS	Transport Layer Security
DDoS	Distributed Denial of Service
HA	High Availability
LB	Load Balancer
FS	File System
API	Application Programming Interface
ACL	Access Control List
HTTP	Hyper Text Transfer Protocol
AMQP	Advanced Message Queue Protocol
CoAP	Constrained Application Protocol
LwM2M	Lightweight Machine-to-Machine
gRPC	Remote Procedure Calls
UDP	User Datagram Protocol
QUIC	Quick UDP Internet Connections
EMQX	Minimum Message Queuing Telemetry Transport
STOMP	Simple Text Oriented Messaging Protocol
AWS	Amazon Web Services

ELB	Elastic Load Balancing
SDK	Software Development Kit
CLI	Command Line Interface
REST	Representational State Transfer
SLA	Service Level Agreement
EPL	Eclipse Public License
EDL	Eclipse Distribution License
MPL	Mozilla Public License
LTS	Long Term Support
GB	Gigabayt
SSD	Solid State Disk
ID	Identification Number
ms	Millisecond
Min	Minimum
Max	Maximum
msg/sec	Microseconds Per Message
Pub	Publisher
Sub	Subscriber

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1. Introduction

Today, Internet of Things (IoT) technologies are rapidly evolving, enabling millions of devices to exchange data with each other [1][2][3]. Due to constraints such as low bandwidth, limited energy consumption, and low latency requirements, lightweight and efficient communication protocols are needed between these devices [4][5]. In this context, MQTT (Message Queuing Telemetry Transport) has become one of the most widely preferred communication protocols for IoT applications [6][7].

This thesis presents a comparative evaluation of three open-source MQTT broker software platforms: Mosquitto, EMQX, and RabbitMQ, based on various performance metrics [8][9]. The goal is to reveal how these systems behave under load and assess their reliability, ultimately aiding the selection of the most suitable broker for real-time data transmission needs [10].

The primary objective of the study is to test the aforementioned brokers in terms of throughput, latency, connection stability, authentication performance, topic routing, and Quality of Service (QoS), and to analyze the results in a structured manner [11][12].

Within this scope, the thesis first introduces the MQTT protocol and the architecture of the brokers under evaluation; then, it describes the test environment, methodology, and test scenarios in detail [13]. In the final section, the results are compared and interpreted, followed by recommendations for broker selection [14][15].

This study aims to serve as a guide for researchers and software developers who plan to build IoT-based systems in areas such as industrial automation, smart cities, and remote monitoring systems [16].

2. Fundamentals of MQTT Brokers

2.1. What is an MQTT Broker?

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol designed to enable data transmission for devices with limited resources and low bandwidth [17][18]. It is widely used in Internet of Things (IoT) applications, where it facilitates reliable and low-latency communication between devices [19]. A key component of the MQTT protocol is the broker, which serves as the central hub for message exchange between clients and ensures the continuity of communication [20].

The MQTT architecture is based on a publish/subscribe model. In this model, data-producing devices are referred to as *publishers*, while devices that consume or subscribe to the data are known as subscribers [21]. The MQTT broker acts as an intermediary between these two endpoints. Messages published to a specific *topic* by a publisher are delivered to all subscribers that have subscribed to that topic, via the broker. This approach enables centralized communication without requiring direct connections between clients [22]. The primary responsibilities of an MQTT broker can be summarized as follows:

- *Message Routing:* The broker distributes messages sent by publishers to all clients subscribed to the corresponding topic [23].
- *Session Management:* It establishes, maintains, and terminates connections with MQTT clients as needed. Additionally, it can retain session information even after the client disconnects, through persistent sessions [24].
- *Quality of Service (QoS) Management:* MQTT supports three levels of message delivery quality (QoS 0, QoS 1, QoS 2), the broker ensures message delivery in accordance with the selected QoS level [25].
- *Authentication and Authorization:* Modern MQTT broker solutions include mechanisms for verifying client identities and enforcing access control to specific topics [26].
- *Message Retention and Last Will and Testament (LWT):* The broker can retain specific messages for future delivery and transmit predefined messages to other clients if a publisher disconnects unexpectedly.

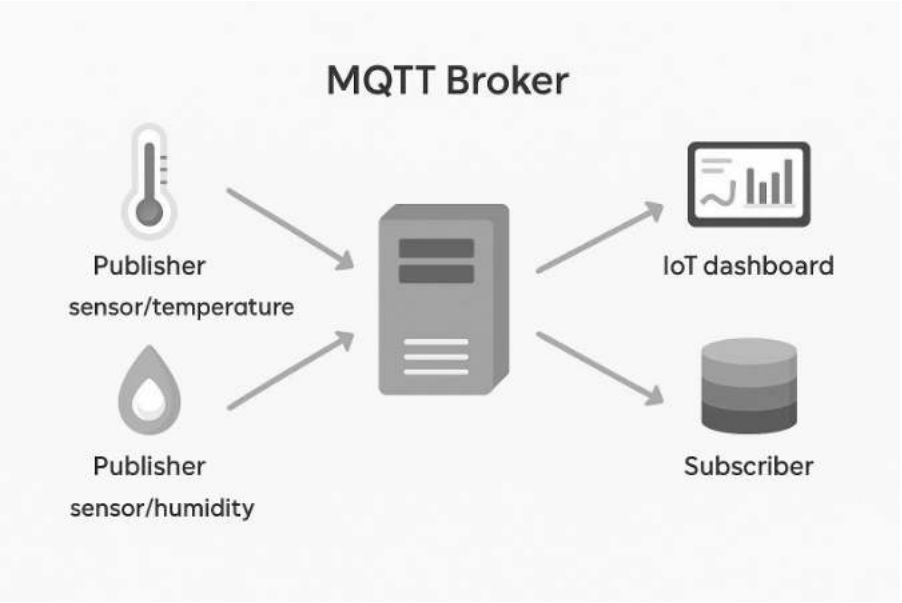

Figure 2.1. MQTT broker architecture.

Since the MQTT broker serves as the central element through which all communication traffic flows within the network, its performance, security, and scalability directly affect the overall efficiency of the system [27]. Therefore, when developing IoT solutions, the selection of an appropriate broker must consider key factors such as expected data volume, connection density, security requirements, and high availability.

There are numerous open-source and commercial MQTT broker implementations available on the market. Among the most commonly used brokers are Eclipse Mosquitto, EMQX, HiveMQ, and the RabbitMQ MQTT plugin [28][29]. Each of these solutions offers distinct advantages in areas such as scalability, remote management, load balancing, and performance, making them more suitable for specific use cases.

In conclusion, the MQTT broker is a central component that forms the backbone of IoT communication infrastructure, managing data exchange between clients and implementing the full functionality offered by the MQTT protocol. Therefore, a thorough understanding of broker architecture and its proper configuration are critical for the successful design of any MQTT-based system.

QoS (Quality of Service) Levels: MQTT offers different levels of Quality of Service (QoS) to accommodate various reliability and message delivery requirements [30]:

- *QoS 0 – At Most Once:* The message is delivered once, with no acknowledgment or guarantee of delivery.
- *QoS 1 – At Least Once:* The message is delivered at least once, and an acknowledgment is required to confirm receipt.
- *QoS 2 – Exactly Once:* The message is delivered exactly once using a four-step handshake process. This is the most reliable and secure delivery level.

These QoS levels make MQTT particularly suitable for scenarios where fast and reliable data transmission is critical, such as IoT applications and sensor data communication [31][32].

2.2. The Role of the Broker in the IoT Ecosystem

The concept of the Internet of Things (IoT) refers to the communication between physical objects or with a central system over digital networks. For these systems to operate efficiently, a communication protocol that supports low energy consumption, minimal bandwidth requirements, and reliable data transmission is essential [33].

In this context, MQTT (Message Queuing Telemetry Transport) plays a critical role in IoT communication infrastructure. Its architectural design and low resource consumption have made MQTT one of the foundational technologies of modern IoT systems [34].

One of the most vital components regulating communication in these systems is the MQTT broker. Acting as a centralized intermediary that manages data flow between clients, the broker enables reliable, scalable, and real-time operation of the IoT ecosystem. The broker's central position allows for decoupled communication, which enhances the system's flexibility and maintainability [35].

The role of MQTT in the IoT ecosystem and the reasons for its widespread adoption include:

Lightweight and Efficient Protocol Structure: MQTT is a lightweight publish/subscribe protocol built on top of the TCP/IP protocol stack. It is specifically designed to meet the needs of IoT devices that operate under constraints such as low power, limited bandwidth, and reduced processing capacity.

- Low Bandwidth Usage: MQTT minimizes protocol overhead by using a fixed header of only 2 bytes. This efficient structure enables reliable and economical communication, particularly for sensor-based devices operating in remote areas or over mobile networks.
- Low Power Consumption: Instead of maintaining continuous connections, MQTT supports event-driven communication. This approach allows battery-powered devices to conserve energy by activating only during data transmission and remaining in sleep mode otherwise, significantly reducing power usage.
- Simplicity and Compatibility: Due to its simple data format and communication model, MQTT can be easily integrated into microcontroller-based systems. This simplicity makes it especially suitable for embedded system applications, contributing to its widespread adoption in resource-constrained environments.

Flexibility Through the Publish/Subscribe Model: The publish/subscribe architecture of MQTT enables clients to communicate without establishing direct connections with one another. This is made possible through the use of a central component, the broker which manages all communication processes.

- Topic-Based Publishing and Subscription: Devices can publish data to a specific *topic*, while other devices can subscribe to these topics to receive the data. This structure helps organize and manage data traffic effectively, especially in systems with a large number of sensors and actuators.
- Real-Time Data Transmission: MQTT supports real-time data delivery with minimal latency. This feature is particularly advantageous in scenarios that require immediate analysis or automated response such as detecting machinery failures or triggering security alarms.
- Client Decoupling: Publishers and subscribers operate independently of one another. This decoupling simplifies network architecture and enhances system

flexibility, ensuring consistent performance even as the number of devices in the network increases.

Reliability Through Various QoS Levels: One of the fundamental reasons for the widespread adoption of the MQTT protocol in IoT systems is its Quality of Service (QoS) mechanism, which enables control over the reliability of data transmission. QoS is a service quality parameter that defines how frequently and accurately messages are delivered between clients. Different IoT applications have varying requirements regarding data security, tolerance to message duplication, and latency.

MQTT offers three distinct QoS levels, allowing flexible configurations to meet these diverse needs. QoS levels can be defined independently for the client-to-broker and broker-to-subscriber communication paths. Each level determines the system's response to potential issues during message transmission, such as data loss, network interruptions, or client reconnections.

2.3. Applications of MQTT in IoT Projects

MQTT is widely adopted across several domains of modern connected systems. In Industrial IoT (IIoT), it is used for transferring performance data from factory machines to centralized systems and supporting predictive maintenance processes. It is also preferred for monitoring key production-line parameters—such as temperature and vibration—through sensor networks. In Smart Home and Building Automation, MQTT enables communication among smart thermostats, security cameras, lighting systems, and energy-management units, allowing them to interact with central controllers. It is especially favored in home-automation scenarios where IoT devices require low bandwidth and energy-efficient communication. In Agricultural Technologies (AgriTech), irrigation controllers, weather stations, and soil-moisture sensors send operational data to cloud platforms via MQTT, supporting precision farming. Because MQTT is designed for low energy consumption, it is well suited for automated agricultural equipment that needs to be remotely controlled by farmers. In Smart Cities, MQTT is employed in traffic-control systems, air-quality monitoring units, smart lighting infrastructure, and waste-management solutions, enabling municipalities to collect and analyze sensor data rapidly and thereby optimize resource

usage and environmental performance. In Healthcare Technologies, portable medical devices—such as heart-rate monitors and blood-glucose meters—transmit readings to hospital or physician information systems using MQTT; since these devices send data only when a measurement occurs, MQTT's lightweight message structure offers clear advantages. In In-Vehicle IoT (Connected Vehicles), onboard sensors and infotainment systems exchange data via MQTT to provide real-time updates on traffic and road conditions. In fleet applications, MQTT supports secure data transmission without requiring devices to maintain constant connectivity, enabling efficient location tracking and operational-data sharing.

3. Performance Comparison

With the growing adoption of IoT applications, the performance of MQTT brokers which form the backbone of messaging infrastructure has become critically important. Each broker is designed with different software architectures, processing capabilities, and resource management strategies.

Therefore, evaluating the performance of an MQTT broker requires comprehensive comparisons based on objective and measurable criteria. This section focuses on three key metrics used to assess the performance of MQTT brokers: message delivery latency, throughput (messages processed per second), and efficiency based on resource consumption.

3.1. Message Delivery Latency

Message delivery latency refers to the time elapsed between the moment a message is sent by the publisher and the moment it is received by the subscriber. Latency is of vital importance in time-sensitive IoT applications such as industrial automation, health monitoring systems, and intelligent transportation solutions [36].

Low latency increases the system's ability to provide real-time feedback and enables prompt responses to events. When comparing latency among different brokers, the following parameters should be taken into account:

- Message size
- Network type and bandwidth (e.g., LAN, WAN, cellular)
- Quality of Service (QoS) level
- Number of concurrently connected clients

In such tests, each message is timestamped at the publisher and the delivery time at the subscriber is measured. Performance evaluation is based on the analysis of average latency, jitter (variation in delay), and maximum latency values.

3.2. Throughput (Messages Per Second)

Throughput refers to the number of messages a broker can process within a given time frame and is typically measured in messages per second [37]. A broker with high throughput is capable of handling high volumes of message traffic from numerous devices, preventing bottlenecks in the system. Typical conditions for a throughput test generally involve multiple clients sending messages concurrently, where each client transmits messages of a fixed size at consistent intervals. Each publisher operates at a

defined publishing frequency, ensuring controlled and repeatable load generation. Additionally, the test progressively increases the number of subscribers and the overall subscription density to push the system toward its limits and observe performance under high-stress conditions.

During testing, it is observed whether messages are delivered without loss, processed according to their QoS levels, and how long the broker can remain stable under load. Throughput is especially critical in applications with high-frequency data streams, such as smart city infrastructures and industrial production lines.

3.3. Efficiency and Resource Consumption

Efficiency is directly related to how effectively a broker utilizes system resources (CPU, memory, disk I/O) relative to its processing capacity. By monitoring CPU and memory usage under load, it can be evaluated whether performance is sustainable. The following measurements are commonly conducted in such comparisons:

- Average and peak CPU usage (%)
- Memory (RAM) consumption (in MB)
- Broker restart times
- Stability under load (e.g., connection drops, crashes, reconnection rates)

While some brokers operate with minimal resource usage, others may require more robust hardware infrastructure. This is particularly critical in embedded systems, microcontroller-based platforms, or resource-constrained gateway devices.

4. Scalibility

In today's rapidly expanding IoT ecosystem, not only the current performance of an MQTT broker but also its resilience and adaptability to increasing system demands have become critically important. Scalability refers to a system's ability to adapt to increases in the number of users, connection density, and data volume without sacrificing performance. In this context, factors such as the number of users and connections as well as queue capacity play a crucial role in evaluating the suitability of an MQTT broker for real-world usage scenarios.

4.1. Number of Users and Connections

The number of simultaneous clients an MQTT broker can support directly affects its overall capacity and performance under heavy load. Large-scale IoT deployments often involve thousands, even millions, of devices connecting to the same broker infrastructure. Therefore, the broker's behavior as the number of concurrent connections increases must be evaluated in terms of connection stability, message latency, and packet loss [38].

Each broker employs different architectural models and resource management strategies. Eclipse Mosquitto offers high performance for a small number of clients due to its lightweight structure, but performance degradation may occur as the number of connections scales into the thousands. EMQX, with its distributed architecture, can handle higher numbers of simultaneous connections by balancing load more effectively, offering stable performance under pressure [39]. RabbitMQ, being based on a traditional message queuing system, excels at managing queued messages as client count grows. However, its MQTT performance may suffer due to architectural overhead.

These tests should also examine how system resources (CPU, RAM) are affected as connection counts rise. Additionally, technical parameters such as disconnection rates, reconnection times, and maximum supported connection limits must be measured.

4.2. Queue Capacity

Due to network issues or the absence of subscribers, MQTT may temporarily store undelivered messages in a queue especially under QoS 1 and QoS 2 levels. Queue capacity refers to how much data a broker can retain in memory or persistent storage and for how long [40]. To evaluate queue capacity, the following metrics are typically used:

- The number of messages successfully queued over a given period,
- Queue wait times and delivery delays,
- Memory footprint of the queues in active operation.

Queuing behavior varies across brokers. Mosquitto typically provides lightweight, transient queuing with limited support for persistent storage. EMQX offers configurable queue management with timeout policies and the option to use persistent storage such as databases or disk-based solutions, making it more robust. RabbitMQ, as a conventional queue-based system, can store large volumes of messages for extended periods, although this may introduce additional latency in MQTT-specific use cases.

4.3. Evaluation

Scalability comparisons reveal how suitable MQTT brokers are for large-scale IoT environments. In addition to the number of connections and queue capacity, factors such as system stability under load, fault tolerance, and resource efficiency must be assessed holistically. These results not only highlight the technical capabilities of brokers but also serve as valuable decision-making tools when selecting broker solutions for enterprise-level IoT deployments.

5. Network Capacity and Bandwidth Utilization

In today's IoT systems, where millions of devices continuously exchange data, network capacity and bandwidth usage have become critical performance indicators. An MQTT broker's ability to optimize its network load is of significant importance, both in terms of system performance and operational cost. Therefore, brokers must be evaluated based on parameters such as message size, message frequency, and bandwidth efficiency.

5.1. Message Size and Frequency

The volume of data generated in IoT networks is largely determined by the size and frequency of messages. Many sensors send small-sized data (a few bytes) multiple times per second. This results in low-volume but high-frequency traffic. Brokers in such scenarios must manage fast client connections and enable efficient data routing with minimal protocol overhead [41]. In some applications, devices transmit large volumes of data (e.g., image data or batch measurements) at set intervals. This reduces the frequency of connections but demands significant bandwidth during transmission bursts.

In both scenarios, brokers are expected to make optimal use of network resources through techniques like packetization, compression, and retransmission mechanisms. Failure to do so may result in network congestion and data loss.

5.2. Bandwidth Efficiency

Bandwidth is a critical and costly resource in IoT systems, affecting both performance and expenses. Efficient bandwidth management not only reduces the total volume of transmitted data but also ensures that messages are delivered timely and accurately to the intended recipients [42]. The main factors affecting bandwidth efficiency in MQTT brokers include:

Protocol Overhead: Although MQTT is a lightweight protocol operating on top of TCP/IP, it has a relatively small overhead. Brokers that minimize protocol overhead can reduce unnecessary network traffic with each transmission.

Message Routing Strategies: Some brokers employ selective routing, delivering messages only to relevant subscribers instead of broadcasting to all clients. This

significantly reduces traffic and enhances bandwidth utilization.

Retransmission Mechanisms: In cases of network interruptions or disconnections, some brokers implement automatic retransmission strategies. However, excessive retransmissions may increase network traffic, especially in low-bandwidth environments. Therefore, a balance between fault tolerance and bandwidth usage is crucial.

Compression Techniques: For transmitting large datasets, advanced brokers may use compression to reduce the size of the transmitted data. This not only saves bandwidth but also shortens transmission times.

Bandwidth efficiency becomes particularly critical in the following scenarios:

Mobile or Cellular Networks: Where data transmission costs are high, efficient bandwidth usage has a direct impact on operational expenses.

Low-Capacity Networks: In rural areas or in systems operating on suboptimal infrastructure, minimizing bandwidth usage helps ensure service continuity.

High-Traffic Systems: In large-scale IoT deployments generating millions of messages simultaneously, efficient bandwidth usage prevents network congestion and supports uninterrupted system operation [43].

Hence, when selecting a broker, considerations should extend beyond message processing capacity to include its bandwidth management strategies. Measurements in this area reveal which brokers offer higher data transmission efficiency under limited resources, enabling more accurate choices for real-world applications.

6. Security Features

In IoT applications, the secure transmission of data is a critical requirement for maintaining system integrity, data privacy, and user confidentiality. Continuous data exchange between devices in IoT architectures makes such networks vulnerable to various security threats. Especially in applications involving personal data or industrial control systems, protecting transmitted information against unauthorized access and tampering is essential for both regulatory compliance and operational safety.

In this context, MQTT brokers operating with lightweight protocols must implement comprehensive security mechanisms to minimize vulnerabilities. The security features of MQTT brokers encompass a multi-dimensional structure including authentication, authorization, data encryption, and attack resilience. Authentication and authorization ensure that only authorized users can access system resources, while data encryption protects the confidentiality and integrity of communication.

Furthermore, a broker's resilience to service interruptions and malicious attacks directly affects the overall security posture of the IoT ecosystem [44]. This section provides a technical evaluation of the security mechanisms offered by different brokers, highlighting their strengths and limitations.

6.1. Authentication and Authorization

Authentication and authorization are foundational steps in MQTT broker security. Authentication typically involves username-password combinations. However, more advanced systems may support OAuth 2.0, LDAP, or Active Directory integrations. Authorization defines which topics a user or device is permitted to access. In large-scale IoT deployments, implementing fine-grained authorization policies is crucial for ensuring data security.

Modern MQTT brokers also support advanced identity verification methods such as TLS-based client certificate authentication (mutual TLS), which allows only authorized devices to connect to the network.

6.2. Encryption

One of the most vital components of data security is encryption. While the MQTT protocol itself does not include built-in encryption, brokers generally rely on the Transport Layer Security (TLS) protocol to secure communication channels. TLS ensures both the confidentiality and integrity of transmitted data and provides protection against man-in-the-middle (MitM) attacks [45].

Some advanced broker solutions also offer message-level end-to-end encryption, ensuring that only specific recipients can read the message content. This adds an extra layer of security, particularly in multi-tier network architectures or in cases where the broker's physical security cannot be fully guaranteed.

6.3. Attack Resilience

IoT networks are particularly vulnerable to threats such as Distributed Denial of Service (DDoS) attacks due to their distributed and large-scale nature. MQTT brokers must be equipped with protection mechanisms against high connection volumes, abnormal traffic spikes, and malicious messages [46]. Certain brokers enhance their resilience with features such as:

- *Connection throttling* to limit the number of requests from a single client.
- *IP blacklisting* to block suspicious sources.
- *Anomaly detection* for abnormal message rates.
- *Automatic disconnection* of malicious or overactive clients.

Moreover, brokers can implement limitations on message size and number of connections to prevent resource abuse. To support secure deployments, the following additional safeguards are recommended:

- *Rate limiting*: Restricting the number of messages that can be sent within a specific time window.
- *Topic wildcard filtering:* Preventing overly broad use of wildcard characters in topic subscriptions.
- *Session expiry policies:* Automatically terminating idle sessions after a defined period.

- *Logging and audit trails:* Recording all authentication events and message transmission activities.

These measures help ensure that brokers maintain their service availability not only under normal operating conditions but also in the face of unexpected attacks or system failures, thereby enhancing the reliability and continuity of IoT systems.

7. Additional Features

The evaluation of MQTT brokers cannot be limited solely to core performance metrics such as latency, throughput, and behavior under load. In real-world IoT applications, factors related to system sustainability, manageability, and long-term scalability are equally critical. In this context, additional features—including persistence mechanisms, high availability and clustering capabilities, management interfaces, supported protocols, load balancing strategies, flexibility and extensibility, cloud and edge deployment support, pricing and community support, as well as backward compatibility—emerge as key criteria that directly influence broker selection.

This chapter presents a comprehensive analysis of Mosquitto, EMQX, and RabbitMQ with respect to these additional features, adopting a holistic perspective. The architectural approaches, functional capabilities, and suitability of each broker for different usage scenarios are examined comparatively. By considering not only technical performance but also practical aspects such as operational ease, scalability, integration capability, and long-term maintenance costs, this chapter aims to support more informed and effective decision-making in the selection of MQTT brokers for diverse IoT system requirements.

7.1. Persistence

In IoT applications, the reliability of data transmission is not solely dependent on real-time connection quality but also closely tied to the system's persistence mechanisms. The MQTT protocol offers various strategies to prevent data loss, particularly in scenarios where network connectivity is unstable or devices are frequently offline. At the core of these strategies are the protocol's different Quality of Service (QoS) levels. QoS 0, QoS 1, and QoS 2 allow users to adjust the level of delivery guarantee and message durability according to their application needs.

An essential factor in persistence is how effectively and reliably brokers support these QoS levels. Especially for QoS 1 and QoS 2, ensuring that messages reach the intended recipient without loss is critical. Brokers can store sent messages in memory or persistent storage for a certain duration, enabling message delivery even after disconnections or system failures. This capability allows for secure message recovery

and delivery once the connection is reestablished [47].

The handling of message loss and retransmission varies depending on the broker's architecture, the type of storage used (e.g., RAM, disk-based databases), and its configuration. Some brokers also offer advanced features such as persistent sessions and retained messages to enhance end-to-end message reliability. These features are particularly important in scenarios requiring high availability and disaster recovery, where robust persistence mechanisms are vital for maintaining data integrity and ensuring continuous system operation.

This section provides a detailed analysis of the persistence strategies implemented by different broker solutions, along with technical performance comparisons under disruption and reconnection scenarios.

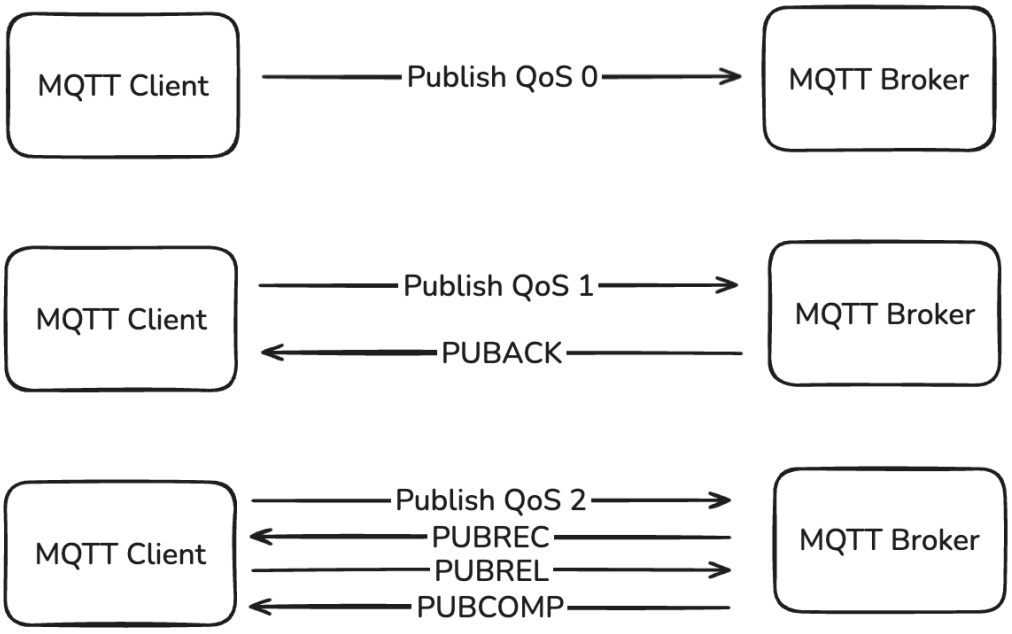

Figure 7.1. MQTT broker QoS Level.

7.2. High Availability and Clustering

In applications such as IoT systems, where continuous data flow is essential, maintaining uninterrupted communication infrastructure is of critical importance. In this context, MQTT brokers' ability to provide high availability (HA) and support clustering capabilities are key factors that directly influence system reliability and scalability.

Clustering refers to the architecture where multiple brokers operate under a single logical structure, enabling load sharing and fault tolerance. A clustered broker setup allows the system to continue functioning without interruption in the event of a node failure. Moreover, through load balancing mechanisms, the system can maintain performance under increasing user and device demand. Broker solutions that support clustering are typically equipped with advanced features such as message synchronization, session sharing, and state replication. These features are especially vital for geographically distributed deployments and large-scale IoT infrastructures [48].

High availability, on the other hand, ensures that a broker remains online for as long as possible while minimizing service interruptions. Key criteria in this area includes broker uptime ratios, fault tolerance mechanisms, and auto-recovery capabilities. Some brokers employ active/active or active/passive replication architectures, which both distribute the load and ensure uninterrupted service in the event of a failure. Additionally, the ability of a system to monitor itself, automatically detect faults, and take corrective action provides a significant advantage for maintaining high availability.

In this section, the clustering architectures, high availability strategies, and failure response capabilities of various MQTT brokers will be examined theoretically. The comparison will address brokers’ clustering features, replication methods, session synchronization, and automated recovery mechanisms. A comparative table summarizing the clustering and HA strategies of each broker is provided below.

Broker	Cluster Support	Session Sync	Queue Replication	Auto-Reco very	Geo-Distri buted
Mosquitto	No (LB+shared FS)	Shared FS	No	Limited	No
EMQX	Native (active/active)	Yes	Yes	Yes	Yes
RabbitMQ	Native	Yes	Mirrored Queues	Yes	Limited

Table 7.1. Clustering and high availability comparison for brokers.

7.3. Broker Management and User Interface

In Internet of Things (IoT) projects, maintaining broker service continuity and optimizing performance depend not only on a strong backend communication infrastructure but also on effective user interfaces and API layers capable of supervising and managing that infrastructure. A broker's management console and its API capabilities give operators real-time visibility into system behavior while supporting automation and integration processes.

Manageability is therefore a crucial factor for open-source MQTT brokers, as it directly affects system monitoring, configuration tasks, and interoperability with external platforms. Widely used open-source brokers—such as Mosquitto, EMQX, and RabbitMQ—provide different levels of management tooling and API functionality to address these operational requirements [49]. Their management dashboards typically present essential performance indicators through visual interfaces, including:

- *Connection Metrics and Session Status:* Number of active connections, persistent sessions, and pending clients.
- *Message Traffic:* Total published and received messages, categorized by QoS levels.
- *Queues and Retained Messages:* Queue lengths, number of retained messages, and message aging statistics.
- *System Resource Usage:* CPU, memory, disk, and network consumption.

- *Historical Logs and Alerts:* Event logs, error reports, and system-generated alerts.

EMQX is equipped with a comprehensive web-based management interface. This panel allows administrators to visually monitor active connections, subscribed topics, message throughput, queue statuses, and system resource utilization. Additionally, user management and Access Control List (ACL) configurations can be handled directly through the interface. EMQX also provides RESTful API support, enabling seamless integration with automation systems for dynamic broker management.

Mosquitto, known for its lightweight and minimal structure, does not include a built-in graphical management interface. Most administrative tasks are performed through command-line tools (e.g., mosquitto_ctrl) and configuration files. While Mosquitto does not provide native API support, system administrators can implement basic monitoring and control using third-party tools or custom scripts.

RabbitMQ is a general-purpose messaging server that supports MQTT via a plugin. Its open-source edition includes a web-based management console that allows users to monitor queues, connections, exchanges, and message flow. Moreover, RabbitMQ offers an HTTP-based REST API, which facilitates programmatic management of users, queues, and configurations.

In conclusion, the scope of the management interface and API support varies across open-source MQTT brokers. EMQX offers one of the most extensive solutions in this area, while Mosquitto provides a more minimalistic approach suitable for basic tasks. RabbitMQ stands out with its multi-protocol support and advanced management capabilities. When selecting a broker, these administrative and integration features should be taken into careful consideration.

	Eclipse Mosquitto	**EMQX**	**RabbitMQ**
Management Panel	No	Yes	Yes
API Support	No	Yes	Yes

Table 7.2. Management panel for brokers.

7.4. Supported Protocols

The flexibility of open-source MQTT brokers is not limited to their compatibility with the MQTT protocol alone. In the context of ensuring interoperability among various IoT systems, support for additional communication protocols plays a crucial role. Beyond MQTT, the ability to handle alternative protocols such as HTTP, AMQP, and CoAP enables integration within heterogeneous system architectures.

EMQX stands out as one of the most comprehensive open-source solutions in terms of multi-protocol support. In addition to full compatibility with MQTT versions 3.1.1 and 5.0, it can integrate a wide range of protocols—including CoAP, LwM2M, WebSocket, HTTP, QUIC, and even gRPC—through its plugin-based architecture. This versatility allows EMQX to be deployed not only in traditional IoT environments but also in microservices and edge computing infrastructures. Features introduced with MQTT 5.0, such as user-defined properties, enhanced error handling, request-response messaging, and session state transfer, are fully supported by EMQX [50].

Mosquitto, on the other hand, is a lightweight broker designed primarily for MQTT 3.1, 3.1.1, and 5.0, making it ideal for embedded systems and resource-constrained devices. While it does not natively support other IoT protocols, limited extensions such as WebSocket support can be configured manually. However, Mosquitto is optimized for pure MQTT communication rather than functioning within a multi-protocol environment.

RabbitMQ offers MQTT support via a plugin rather than as a core feature. Its primary strength lies in the AMQP protocol, which it supports natively. In addition, RabbitMQ is capable of handling HTTP APIs, STOMP, and WebSocket communications. This makes RabbitMQ particularly suitable for acting as a bridge in multi-protocol systems, and it is often used in enterprise messaging scenarios outside of IoT. However, its support for MQTT is comparatively limited, especially with regard to the advanced features introduced in MQTT 5.0, which are not yet fully implemented in its MQTT plugin.

In conclusion, EMQX is the most capable open-source option for systems requiring broad protocol support. Mosquitto is well-suited for applications that prioritize simplicity and reliable MQTT-only communication.

Meanwhile, RabbitMQ, with its wide protocol compatibility, offers strong interoperability across different messaging environments, although its MQTT capabilities remain secondary to its core AMQP functionality. These distinctions are particularly significant when considering scalability and integration capacity in complex IoT ecosystems.

Broker	MQTT 3.1.1	MQTT 5.0	AMQP	HTTP	CoAP	Web Socket	Other Protocols
EMQX	Yes	Yes	Yes (plugin)	Yes (REST API)	Yes (plugin)	Yes	LwM2M, QUIC, gRPC (plugin)
Mosquitto	Yes	Yes	No	Yes (with proxy)	No	Yes	No
RabbitM Q	Yes (with plugin)	Yes (limited)	Yes	Yes	No	Yes	STOMP, MQTT-SN, STOMP-over-WS

Table 7.3. Supported protocols for brokers.

7.5. Load Balancing

In large-scale IoT systems, it is common for thousands or even millions of devices to communicate with an MQTT broker simultaneously. This intense connection traffic can lead to serious performance issues, connection interruptions, and message loss on a single broker instance. Therefore, implementing load balancing mechanisms becomes a critical requirement to ensure system scalability and stability. Load balancing distributes client connection requests across multiple broker instances, thereby balancing network and CPU load. This not only allows for more efficient use of system resources but also prevents potential bottlenecks .

From an MQTT broker perspective, load balancing is generally achieved through two main approaches: external load balancing (via services such as Nginx, HAProxy, or AWS ELB) and built-in clustering support.

Lightweight brokers like Mosquitto are designed for single-node deployments and lack native clustering or automatic load balancing mechanisms. As a result, Mosquitto must typically be deployed in conjunction with an external load balancer in multi-node setups. While this approach is advantageous in resource-constrained environments due to its simplicity and low overhead, it is limited in terms of flexibility and scalability.

EMQX, on the other hand, is a modern MQTT broker that provides advanced load balancing capabilities. With its distributed architecture, it offers horizontal scalability and supports automatic clustering, including message synchronization between broker nodes. EMQX's built-in load balancing mechanisms significantly enhance system stability in high-traffic environments. Additionally, EMQX provides real-time visibility into client distribution and traffic loads, allowing system administrators to quickly respond to fluctuations in usage [51].

RabbitMQ, although originally designed around the AMQP protocol, also supports MQTT via a plugin. RabbitMQ is well known for its robust clustering and load balancing mechanisms. It is often preferred in high-availability scenarios due to its ability to replicate messages between broker nodes, ensuring both connection continuity and data persistence. However, compared to Mosquitto and EMQX, RabbitMQ may involve more complex configuration and management.

It is important to note that load balancing not only increases the system's connection capacity but also enhances its fault tolerance. In the event of a broker instance failure, the load balancing layer can redirect traffic to active nodes, ensuring continuous service. In this regard, a broker's scalability is directly tied to its high availability strategy, and the overall system architecture should be designed with these elements in mind.

In conclusion, the load balancing capabilities of MQTT brokers play a decisive role in the success of large-scale IoT deployments. Therefore, performance, architectural flexibility, fault tolerance, and ease of management must all be considered when evaluating broker options.

	Eclipse Mosquitto	EMQX	RabbitMQ
Built-in Load Balancer	No	Yes	No
External LB Compatibility	Yes (HAProxy, Nginx)	Yes (HAProxy, Nginx)	Yes (HAProxy, Nginx)
Cluster Support	Limited (via bridges)	Yes (auto-discovery, full clustering)	Yes (native clustering)
Message Queue Replication	No	Yes	Yes (via mirrored queues)
Session Persistence	Basic (QoS-based)	Advanced (clustered stateful sessions)	Yes (durable queues, persistent messages)

Table 7.4. Load balancing comparison for brokers.

7.6. Flexibility and Extensibility

IoT applications require flexible and extensible software infrastructures to adapt to evolving needs, increasing numbers of devices, and advancements in technology. In this context, MQTT brokers with modular architectures offer a significant advantage for developers. A modular structure allows new features to be integrated easily on top of the core broker functionality. Brokers that support a plugin-based architecture enable dynamic addition of various functionalities—from security features to message routing. For instance, EMQX, despite being open source, has a rich plugin ecosystem that allows the integration of features such as authentication, message routing, and data transformation through external modules.

In addition to modularity, brokers that provide Software Development Kits (SDKs) and APIs greatly simplify the application development process, offering developers broader programming flexibility. The support of MQTT brokers for multiple programming languages is critical for integration with other software components in an IoT system. Brokers like EMQX and RabbitMQ provide RESTful APIs, command-line interface (CLI) tools, and comprehensive SDKs, enabling seamless integration with popular languages such as Python, Java, and Node.js. In contrast, Mosquitto adopts a more minimalist approach and offers limited API support; however, it remains sufficient for rapid prototyping in smaller systems.

Such flexibility and extensibility not only address current requirements but also lay the foundation for systems that are prepared for future integrations and expansion. Therefore, when selecting an MQTT broker, organizations should not only consider present use cases but also evaluate how the system might grow and which technologies it may need to integrate with over time.

	Eclipse Mosquitto	**EMQX**	**RabbitMQ**
Modular Structure	Limited extension support (not plugin-based)	Highly modular (plugin/module-b ased architecture)	Broad plugin/module support (plugin-based)
Multi Language Support	Written in C; limited external integration	Supports many languages (Python, Node.js, Java, Go, etc.)	Multi-language support (Elixir, Java, Python, etc.)
Plugin Support	None / limited	Available (e.g., WebHook, Kafka, Auth plugins)	Available (AMQP plugins, MQTT plugin, etc.)
SDK / API Support	No native API support; basic config files	REST API, gRPC, WebSocket APIs available	REST API, CLI, and management APIs available
Application Extensibility	Limited	High (plugin architecture + distributed support)	High (customizable and extensible plugin system)

Table 7.5. Comparison of broker characteristics.

7.7. Cloud and Edge Support

In IoT systems, processing data not only in centralized environments but also near the edge closer to the devices is crucial for reducing latency and enhancing system reliability. In this regard, the ability of MQTT brokers to operate seamlessly in both cloud and edge environments plays a key role in ensuring the flexibility and sustainability of system architectures.

From a cloud deployment perspective, brokers such as EMQX and RabbitMQ can be easily deployed on major cloud platforms, including AWS, Azure, and Google Cloud. Thanks to their support for Docker and Kubernetes, these brokers can be

containerized and run in auto-scalable configurations across cloud infrastructures. Additionally, EMQX offers its own managed cloud service, which allows users to deploy MQTT brokers without dealing with infrastructure management. Mosquitto, being a lightweight broker, can also be deployed in cloud environments, but it may require more manual configuration and additional supporting infrastructure compared to more feature-rich brokers.

Regarding edge device deployments, Mosquitto is particularly suitable due to its low resource consumption and compact footprint. It performs efficiently on devices such as Raspberry Pi, ARM-based boards, or embedded systems, making it a preferred choice in edge computing scenarios. EMQX also provides strong support for edge environments through its EMQX Edge component, which offers an independent messaging structure designed for low-latency, localized communication. RabbitMQ, on the other hand, is relatively limited in edge deployments and is better suited for integration into centralized, enterprise-grade systems.

In conclusion, when selecting an MQTT broker, it is essential to consider not only its performance but also the target deployment environment whether cloud-based or edge-oriented as well as the hardware capabilities of that environment. Mosquitto offers a lightweight and efficient solution for edge devices, EMQX provides a balanced and comprehensive option for both cloud and edge scenarios, while RabbitMQ excels in enterprise infrastructures with centralized, high-volume messaging demands.

	Eclipse Mosquitto	EMQX	RabbitMQ
Cloud Compatibility	Deployable manually	Supported both locally and via EMQX Cloud	Runs on major cloud platforms
Docker/Kubernetes Compatibility	Basic support	Helm and Operator support available	Helm and Operator support available
Edge Device Compatibility	High (very lightweight)	Supported with EMQX Edge module	Limited usage in edge environments
Hardware Resource Consumption	Low	Moderate	Moderate
Edge Customization Support	Not supported	Customization via EMQX Edge	Not supported

Table 7.6. Cloud and edge support comparison.

7.8. Pricing and Community Support

When selecting an MQTT broker, it is essential to consider not only technical capabilities but also ownership and operational costs as well as community support—factors that play a critical role in the sustainability of long-term deployments. One of the key distinctions in this context lies in whether the broker is open source or commercially licensed. Open-source brokers are typically available free of charge, benefit from large user communities, and are frequently updated. Community forums, GitHub repositories, and other collaborative platforms often provide accessible support. Mosquitto and RabbitMQ fall into this category and are widely used both in academic research and production environments.

Brokers such as EMQX offer both open-source and commercial editions. While the open-source version is generally sufficient for many basic use cases, advanced features—including high availability, enterprise-grade monitoring, and SLA-backed technical support—are usually part of the commercial offerings. Therefore, depending on the intended application, it may be necessary to strike a balance between the open-source and commercial versions. For instance, open-source versions may suffice for small-scale IoT projects, whereas industrial or enterprise-level applications may require the enhanced capabilities and support found in commercial editions.

Community support is especially vital for open-source software. The presence of an active community accelerates problem resolution and contributes to the stability and robustness of the system [52]. Long-standing projects such as Mosquitto benefit from extensive documentation and a well-established contributor base. Similarly, EMQX maintains an open-source core supported by a vibrant developer community.

In summary, open-source MQTT brokers typically offer a cost-effective solution, while commercial versions cater to enterprise needs with advanced functionalities and professional support services. Thus, when choosing a broker, factors such as project scale, security requirements, maintenance resources, and total cost of ownership should be thoroughly evaluated to make an informed decision.

	Eclipse Mosquitto	**EMQX**	**RabbitMQ**
License Type	Open Source (EPL/EDL)	Open Source (Apache 2.0)	Open Source (MPL 2.0)
Commercial Edition Available	No	Yes (EMQX Enterprise)	Yes (Commercial support available)
Community Activity	Moderate-High (GitHub, forums)	High (GitHub, forums, Slack, Discord)	High (Large user base, forums)
Official Documentation	Basic and minimal	Comprehensive and up-to-date	Extensive and enterprise-grade
Ease of Installation	Very easy (lightweight architecture)	Moderate (more components involved)	Moderate (based on AMQP, needs configs)
Release Updates	Regular	Regular	Regular
Cost	Free	Free (Community) / Paid (Enterprise)	Free (Open Source Edition)

Table 7.7. Pricing and community comparison.

7.9. Backward Compability and Version Updates

Ensuring sustainability and long-term usability in IoT systems requires MQTT brokers to provide strong backward compatibility and a reliable approach to version updates. In this context, it is crucial that upgrading to newer versions does not disrupt existing infrastructures. Maintaining operational continuity during updates not only

supports uninterrupted software development cycles but also contributes to reducing operational costs.

During version upgrades, several critical elements must be evaluated for backward compatibility, including configuration files, user permissions, data persistence settings, and protocol versions. For example, newer versions of EMQX support MQTT 5.0 while still maintaining compatibility with the earlier MQTT 3.1.1 protocol. This enables legacy devices to remain operational without requiring immediate firmware changes. Similarly, Mosquitto, despite its minimalist architecture, has added MQTT 5.0 support in recent releases while continuing to support older versions, thus ensuring integration with a wide range of devices.

Moreover, comprehensive and well-structured documentation accompanying each version release plays a significant role in minimizing upgrade-related issues. Open-source brokers like EMQX typically offer detailed upgrade guides for each major release. On the other hand, RabbitMQ, which is primarily based on the AMQP protocol, provides tools and configurations to help users transition smoothly from AMQP-based systems to MQTT-based messaging where needed.

Backward compatibility is not limited to protocol versions alone. It also encompasses the broker's ability to maintain stable connections with devices that have limited processing power. Key features such as Quality of Service (QoS) handling, session persistence, and connection timeout configuration must be flexible to accommodate older hardware. In this respect, Mosquitto continues to be a reliable option for resource-constrained systems, while EMQX, although more complex and feature-rich, can also be tailored to support legacy devices through specific configurations.

In conclusion, when designing scalable and maintainable IoT systems, it is essential to consider how MQTT brokers handle version transitions and maintain compatibility with existing infrastructure and legacy devices. Brokers that ensure backward compatibility can significantly reduce technical debt and facilitate smoother upgrades in dynamic IoT environments.

	Eclipse Mosquitto	EMQX	RabbitMQ
Update Frequency	Moderate (every 6–9 months)	Regular (every 3–4 months)	Periodic (major releases)
MQTT 3.1.1 Support	Yes	Yes	Yes (via plugin)
MQTT 5.0 Support	Yes	Yes	Limited (via plugin)
Backward Compatibility	High	High	Moderate
Legacy Device Support	High	Medium–High	Moderate
Update Documentation	Basic level	Comprehensive	Comprehensive

Table 7.8. Compatibility and version update comparison.

8. Testing and Evaluation

The MQTT protocol is widely used in IoT systems due to its lightweight structure, low bandwidth requirements, and publish-subscribe communication model. However, the effectiveness of this protocol heavily depends on the performance, scalability, and functional capabilities of the underlying MQTT broker software. In this study, a series of tests have been conducted on three widely adopted open-source MQTT brokers Mosquitto, EMQX, and RabbitMQ to evaluate their behavior under different conditions.

The primary objective of these tests is to analyze and compare how each broker performs in various scenarios, thus helping to determine which one is most suitable for specific use cases. These evaluations provide valuable insights for making informed decisions when selecting a broker for real-world IoT applications, considering factors such as performance, stability, scalability, security, and feature set [53]. The results obtained from these tests reveal the strengths and limitations of each broker and serve as a guide for system designers and developers.

The conducted tests include critical metrics such as latency, load handling, throughput, long-term stability, quality of service (QoS), topic routing, and authentication mechanisms. All tests were implemented using Python with the Paho-MQTT client library, ensuring standardization and flexibility in scenario development. Each broker was deployed under identical conditions using Docker and Docker Compose on an Ubuntu 22.04 Linux environment, providing a fair and reproducible testing ground.

In the following sections, detailed information about the test environment, test scenarios, and the outcomes will be systematically presented, offering a comprehensive comparison of the brokers' capabilities and practical implications.

8.1. Description of the Test Environment

In this study, various performance and functionality tests were conducted on three different open-source MQTT broker software: Mosquitto, EMQX, and RabbitMQ, which all support the MQTT protocol. To ensure fair, repeatable, and consistent results, the test environment was carefully designed and configured.

All tests were executed on a physical machine running Ubuntu 22.04 LTS. The test system was equipped with a 4-core CPU, 8 GB of RAM, and 320 GB of SSD storage. The environment utilized Docker version 27.0.3 and Docker Compose v2.28.1, and all brokers were deployed in isolated containers. For secure and low-latency communication between the containers, a dedicated Docker bridge network was created, and all brokers and clients were connected within this network.

Each MQTT broker was deployed using its official and up-to-date Docker image:

- Mosquitto: eclipse-mosquitto:2.0
- EMQX: emqx/emqx:5.0.13
- RabbitMQ: rabbitmq:3.13-management

To ensure consistent test conditions and a fair comparison, each container was limited to 512 MB of RAM and 0.5 CPU cores. These constraints were chosen to reflect real-world IoT scenarios where limited system resources are common. The resource usage and system performance of each container were monitored using Portainer (portainer-ce:2.21.0 image). Two main approaches were used during testing:

Load, throughput, and stability tests: These tests examined the behavior of the MQTT brokers under high traffic. The open-source stress-testing tool 'flaviostutz/mqtt-stresser' was used as a Docker container. Different combinations of publishers and subscribers were tested to evaluate message throughput, connection handling, and message loss.

Latency, QoS, topic routing, and authentication tests: Custom test scripts were developed in Python using the following libraries:

- paho.mqtt.client (for connecting to the broker and message handling)
- time (for measuring response durations)
- uuid (to generate unique client IDs)
- threading (for creating concurrent test scenarios)
- statistics (for data analysis)
- json and pandas (for log parsing and result processing)Each broker was configured

specifically for the test scenarios. For instance, MQTT support was enabled in RabbitMQ using plugins, and EMQX was configured with specific authentication and QoS settings. In the case of Mosquitto, the mosquitto.conf file was customized to test different user authentication and QoS scenarios.

As a result, this test environment allowed for the detailed evaluation of broker performance and functional capabilities across various scenarios, providing the basis for in-depth analysis and comparison.

```
root@nsrvb02:/mqtt-comparison/brokers/emqx# docker ps
CONTAINER ID   IMAGE                    COMMAND                  CREATED          STATUS                    PORTS
NAMES
b9cb45e0e0d2   rabbitmq:3.13-management   "docker-entrypoint.s…"   12 minutes ago   Up 12 minutes (healthy)   4369/tcp,
5671/tcp, 0.0.0.0:5672->5672/tcp, :::5672->5672/tcp, 15691/tcp, 15692/tcp          rabbitmq-broker
1b58c0265d83   emqx/emqx:5.0.13          "/usr/bin/docker-ent…"   39 hours ago     Up 39 hours (healthy)     4370/tcp, 5369/tcp,
8883/tcp, 18084/tcp, 0.0.0.0:8083->8083/tcp, :::8083->8083/tcp          emqx-broker
444027e1ff34   eclipse-mosquitto:2.0     "/bin/sh -c 'chmod 6…"   43 hours ago     Up 43 hours               0.0.0.0:1883->1883/tcp,
:::1883->1883/tcp                                   mosquitto-broker
```

Figure 8.1. MQTT brokers on docker.

Figure 8.1. shows the docker ps output captured during the test environment setup, listing the running MQTT broker containers. As shown, EMQX, Mosquitto, and RabbitMQ were configured as separate containers on the same Docker network with specific port mappings and resource limitations. This setup enabled isolated yet equal testing conditions for each broker.

Additionally, to ensure test repeatability and comparability, each broker was executed with identical system resources.

8.2. Installation of Brokers on Docker

All MQTT brokers in this study were deployed on a single Ubuntu 22.04 system using Docker and Docker Compose. A custom Docker network (mqtt-net) enabled inter-container communication, and each container was connected to it to create a controlled test setup. Resource limits of 0.5 CPU and 512 MB RAM per container were applied to simulate constrained IoT conditions.

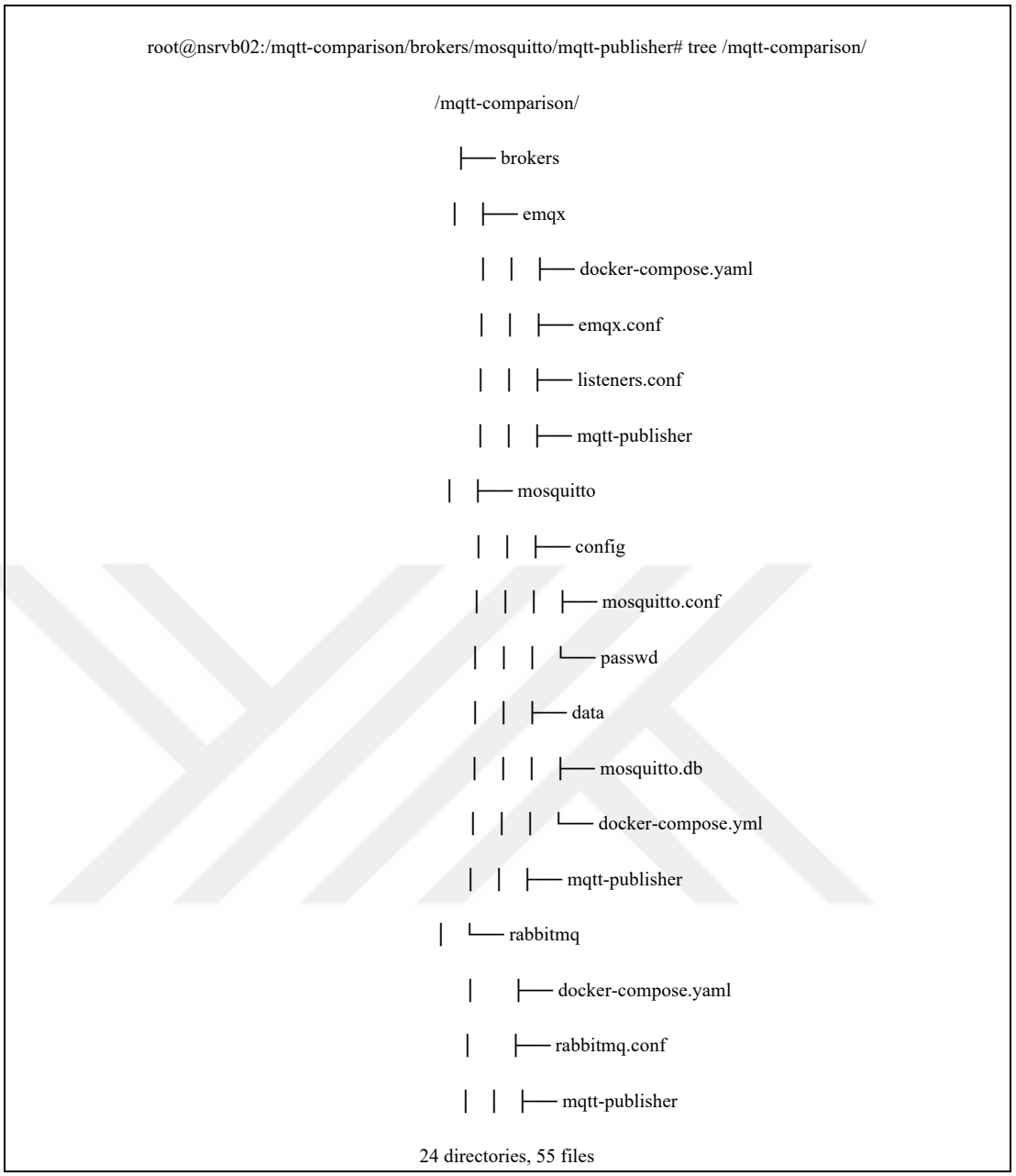

Figure 8.2. Installation document's directories.

8.3. Mosquitto Installation

The official eclipse-mosquitto:2.0 image was used. A custom password file was created using the mosquitto_passwd utility and mounted into the container for authentication. After configuring access credentials, the container was launched via Docker Compose. To test the broker functionality, MQTT clients were used via the mosquitto-clients package. Figure 8.3. presents the sample test commands used in this setup.

```
docker network create mqtt-net
docker run --rm -it -v "$PWD/config:/mosquitto/config" eclipse-mosquitto \
mosquitto_passwd -c /mosquitto/config/passwd testuser
password: 123456

chown 1883:1883 config/passwd
docker compose up -d
# For test;
sudo apt install -y mosquitto-
clients
mosquitto_sub -h localhost -p 1883 -u testuser -P 123456 -t test/topic
# In a seperate terminal
mosquitto_pub -h localhost -p 1883 -u testuser -P 123456
-t test/topic   -m "Merhaba MQTT!"
```

Figure 8.3. Example Mosquitto setup and MQTT client test commands used in the experimental environment.

8.4. EMQX Installation

The EMQX broker was deployed with the emqx/emqx:5.0.13 image and started via Docker Compose. For the initial test phase, no extra authentication mechanisms were enabled. Figure 8.4. presents the sample test commands used in this setup.

```
1. docker compose up -d
          2.
  # For test;
      mosquitto_sub -h localhost -p 1884 -t test/topic
  # In a separate terminal
      mosquitto_pub -h localhost -p 1884 -t test/topic -m "Hello EMQX!"
```

Figure 8.4. Sample EMQX deployment and test commands used in the experimental setup.

8.5. RabbitMQ Installation

The RabbitMQ broker was installed using the rabbitmq:3.13-management image via Docker Compose. MQTT functionality was enabled through plugin support. Figure 8.5. presents the sample test commands used in this setup.

```
1. docker compose up -d
2.
      # For test;
      mosquitto_sub -h localhost -p 1885 -t test/topic
      # In a separate terminal
      mosquitto_pub -h localhost -p 1885 -t test/topic -m "Hello RabbitMQ!"
```

Figure 8.5. Sample RabbitMQ deployment and test commands used in the experimental setup.

8.6. MQTT Broker Testing Framework

In this study, the performance and functionality levels of the MQTT protocol-supporting brokers (Mosquitto, EMQX, and RabbitMQ) were evaluated comparatively through various test scenarios. The testing methods were designed to reflect real-world use cases while also measuring key capabilities such as communication efficiency, security features, and system stability.

The purpose of these tests was not limited to measuring performance metrics alone; they also examined how brokers behave under high load, their message delivery latency, support for protocol-level features, and authentication mechanisms. In this context, the selected tests aimed to reveal the suitability of each MQTT broker for IoT applications from a multidimensional perspective.

The test environment was configured to ensure a controlled and equal setup for all brokers. Each broker was deployed on the same system inside Docker containers with limited computing resources, which ensured fair and comparable results. The tests were conducted using custom client scripts written in Python, along with open-source testing tools.

The conducted tests were organized into seven main categories: (1) Latency Test, (2) Load Test, (3) Stability Test, (4) Authentication Test, (5) Topic Routing Test, (6) Throughput Test, and (7) QoS Test. Each of these test categories will be presented in the following subsections.

8.6.1. Latency Tests

Latency refers to the time it takes for a message to travel from the publisher to the subscriber. In this test, the transmission time of each published message was measured using timestamps, allowing for the calculation of message delay. In time-sensitive IoT applications such as sensor-based alert systems low latency is critically important. This test demonstrates how fast and consistent each broker is in delivering messages. Additionally, the latency tests and results for each broker are presented below.

8.6.1.1. Mosquitto Latency Test

As shown in Figure 8.6, timestamped messages are published to the latency/test topic, and it is observed that these messages are successfully received by the subscriber. The message delivery is confirmed, and the transmission times of the corresponding messages will be calculated upon completion of the test.

```
root@nsrvb02:/home/nsadmin# mosquitto_sub -h localhost -p 1883 -u testuser -P 123456 -t latency/test
1748955507.380611
1748955509.005461
1748955510.7095935
1748955512.412579
1748955514.116375
1748955515.8207877
1748955517.5247521
1748955519.229249
1748955520.93279
1748955522.6365085
1748955524.3401375
1748955526.044661
```

Figure 8.6. Mosquitto latency test message.

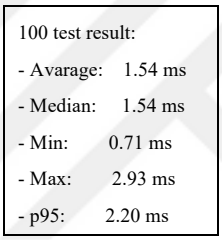

Figure 8.7. Mosquitto latency test results.

Figure 8.7 presents the latency test results, providing an evaluation of the Mosquitto MQTT broker's message transmission times. Based on a test of 100 messages, the statistics show an average latency of 1.54 ms, a median latency of 1.54 ms, a minimum latency of 0.71 ms, and a maximum latency of 2.93 ms. Additionally, the 95th percentile latency (p95) remains below 2.20 ms.

These results indicate that the Mosquitto broker delivers low latency and consistent performance in message transmission. The equality of the average and median values suggests stable operation, while the low minimum and limited maximum values imply that spikes in latency are rare. Therefore, Mosquitto demonstrates the capability to provide a reliable communication infrastructure suitable for latency-sensitive IoT applications.

8.6.1.2. EMQX Latency Test

As shown in Figure 16.8, timestamped messages are published to the latency/test topic, and it is observed that these messages are successfully received by the subscriber. The message delivery is confirmed, and the transmission times of the corresponding messages will be calculated upon completion of the test.

```
root@nsrvb02:/home/nsadmin# mosquitto_sub -h localhost -p 1884 -u testuser -P testpassword -t latency/test
1748956005.9118867
1748956007.6169548
1748956009.3216724
1748956011.025573
1748956012.7295073
1748956014.4327016
1748956016.1374028
1748956017.8407633
1748956019.545183
1748956021.2485103
1748956022.9532623
1748956024.6587942
1748956026.3621545
```

Figure 8.8. EMQX latency test message.

```
100 test result:
- Avarage:   10.28 ms
- Median:    2.55 ms
- Min:       1.05 ms
- Max:       71.22 ms
- p95:       58.60 ms
```

Figure 8.9. EMQX latency test results.

Figure 16.9 presents the latency test results for the EMQX broker, providing significant insights into its message delivery performance. Based on a test involving 100 messages, the average latency was measured at 10.28 ms, with a median of 2.55 ms, a minimum latency of 1.05 ms, and a maximum latency of 71.22 ms. The 95th percentile latency was recorded at 58.60 ms.

These results indicate that EMQX is capable of delivering messages with low latency in many cases; however, it also shows noticeable spikes in delivery time under certain conditions. The discrepancy between the average and median values suggests potential instability in latency performance. The high maximum and 95th percentile values highlight that message delivery times can significantly increase, particularly under high load or unexpected system behavior.

This implies that although EMQX generally offers a robust infrastructure, it should be carefully evaluated for real-time or time-sensitive IoT applications, where consistent low latency is critical.

8.6.1.3. RabbitMQ Latency Test

As shown in Figure 8.10, timestamped messages are published to the latency/test topic, and it is observed that these messages are successfully received by the subscriber. The message delivery is confirmed, and the transmission times of the corresponding messages will be calculated upon completion of the test.

```
root@nsrvb02:/home/nsadmin# mosquitto_sub -h localhost -p 1885 -u admin -P admin -t latency/test
1748956456.406786
1748956458.190748
1748956459.895058
1748956461.5987287
1748956463.3023574
1748956465.0067182
1748956466.711525
1748956468.4154427
1748956470.118703
1748956471.8221035
1748956473.5269697
1748956475.2309804
```

Figure 8.10. RabbitMQ latency test message.

```
100 test result:
- Avarage:   2.73 ms
- Median:    2.67 ms
- Min:       1.44 ms
- Max:       8.00 ms
- p95:       3.61 ms
```

Figure 8.11. RabbitMQ latency test results.

Figure 8.11 presents the latency test results for the RabbitMQ broker. In a test consisting of 100 messages, the average latency was measured as 2.73 ms, with a median value of 2.67 ms. The minimum latency recorded was 1.44 ms, while the maximum latency reached 8.00 ms. Additionally, the 95th percentile latency was 3.61 ms.

These results indicate that RabbitMQ provides low and consistent message delivery latency. The close proximity of the average and median values suggests that the system demonstrates stable performance in terms of latency. The relatively low maximum value and a 95th percentile below 4 ms further highlight RabbitMQ's ability to maintain reliable message delivery even under load.

In this context, RabbitMQ emerges as a reliable option for real-time or time-sensitive IoT applications. The low variance in latency enhances the predictability and controllability of the system, making it suitable for applications where consistent message timing is critical.

8.6.1.4. Comparative Analysis of Latency Tests

The results of the latency tests reveal notable differences in message delivery times across the Mosquitto, EMQX, and RabbitMQ brokers. Based on tests involving 100 messages per broker, these measurements provide valuable insights into the suitability of each system for time-sensitive applications.

Mosquitto delivered the best performance, with an average latency of 1.54 ms, showing highly consistent and low delay. This makes it especially well-suited for latency-critical IoT use cases. EMQX, on the other hand, exhibited the highest average latency at 10.28 ms, with a maximum value of 71.22 ms and a 95th percentile of 58.60 ms. These spikes indicate that EMQX may experience occasional delivery delays under load. RabbitMQ achieved the second-best performance, with an average latency of 2.73 ms, and demonstrated low variance and stable results throughout the test.

These findings suggest that Mosquitto and RabbitMQ are more advantageous for real-time IoT scenarios, whereas EMQX, despite its advanced features and wide protocol support, requires careful consideration when latency is a critical factor.

Broker	Average Latency (ms)	Median Latency (ms)	Min (ms)	Max (ms)	95th Percentile (ms)
Mosquitto	1.54	1.54	0.71	2.93	2.20
EMQX	10.28	2.55	1.05	71.22	58.60
RabbitMQ	2.73	2.67	1.44	8.00	3.61

Table 8.1. Latency performance comparison.

As seen in Table 8.1, Mosquitto offers the lowest and most consistent latency among the three brokers. RabbitMQ also performs well, providing stable and predictable delivery times. EMQX, however, exhibits occasional latency spikes, which may impact performance in real-time scenarios. Therefore, Mosquitto and RabbitMQ are better suited for latency-sensitive applications, while EMQX may be more

appropriate for environments requiring multi-protocol support or higher connection capacity, where latency is not the primary concern.

8.6.2. Load Test

In this test, the behavior of the brokers under high user load was analyzed by sending continuous messages through a varying number of concurrent clients. The test evaluated how each broker handled connection management, message queue saturation, and error rates under load. This test is particularly critical for real-world IoT scenarios where the number of active connections can rapidly increase—for example, when thousands of sensors come online simultaneously during a disaster event.

8.6.2.1. Mosquitto Load Tests

In the load tests conducted for the Mosquitto broker, the system's performance was systematically analyzed under varying numbers of concurrent clients and message volumes. The tests were performed with between 10 and 100 simultaneous clients, each publishing from 100 up to 200,000 messages. Test results are presented in *Appendix I*.

In the first four test scenarios, Mosquitto successfully delivered all messages sent by all clients, achieving a 100% message delivery rate. Furthermore, no connection errors, timeouts, or similar issues were observed. This demonstrates that Mosquitto exhibits high stability under low to moderate loads.

However, in tests involving 150,000 and 200,000 messages, the broker's performance approached its limits. Especially in the seventh test, the message delivery rate dropped to 92%, and 65% of clients experienced timeouts. This indicates that Mosquitto begins to show performance degradation beyond a certain load threshold, possibly due to resource limitations or message queue management inefficiencies. It should also be noted that these tests were conducted under limited CPU and memory resources.

When examining publishing and receiving throughput, it is observed that message

sending speeds were relatively high with a low number of clients but decreased as both client and message counts increased. Nevertheless, Mosquitto maintained consistent throughput distributions, indicating stable operation within its resource boundaries.

In conclusion, Mosquitto stands out as a highly suitable option for small to medium-scale IoT applications on low-resource systems. However, in high-volume and heavily concurrent scenarios, scaling up system resources or opting for more advanced broker architectures becomes necessary.

8.6.2.2. EMQX Load Tests

The load tests conducted for the EMQX broker provided a detailed assessment of its performance under varying numbers of concurrent clients and message loads. Tests were performed with client counts ranging from 10 to 100, and message volumes ranging from 100 to 200,000 messages per client. Test results are presented in *Appendix II*.

In the first five tests, EMQX successfully published and delivered all messages, completing the tests with a 100% success rate. No errors, connection failures, or timeouts were observed in these scenarios. This indicates that EMQX is a highly stable and reliable broker under low to moderate load conditions.

However, as the system approached its limits in the sixth and seventh tests, significant performance degradation was observed. In the sixth test, only 91% of the messages were successfully delivered, and 38% of the clients experienced timeouts. The seventh test showed further decline, with a message delivery rate dropping to 77% and 86% of clients failing to complete the test due to timeouts. These results suggest that EMQX encounters bottlenecks under extreme message loads due to resource constraints.

When examining the publishing and receiving throughput, high transmission rates were recorded in the tests with fewer clients and lower message volumes. However, these rates declined as the load increased. Nevertheless,EMQX was able to maintain data integrity for a long duration, with performance degradation only occurring under heavy load conditions.

In conclusion, EMQX with its high performance, modern architecture, and protocol support is well suited for medium-scale IoT applications. However, to ensure more stable performance under intense loads, enhancements such as increased system resources or advanced configurations like clustering may be necessary.

8.6.2.3. RabbitMQ Load Tests

The RabbitMQ load test results reveal how the system performs under varying levels of client and message load. Test results are presented in *Appendix III*.

In the initial tests (e.g., with 10 clients and 100 messages — a low-intensity scenario), RabbitMQ operated with high stability: 100% of the messages were successfully published and received without any errors. The publishing throughput reached up to 37,839 messages per second, and the receiving throughput peaked at 62,945 messages per second, indicating that the system can handle small-scale workloads with high efficiency.

In moderate-load scenarios (e.g., 100 clients and 10,000 messages), RabbitMQ continued to perform reliably with 100% success in both publishing and receiving messages. Publishing throughput in these cases reached 64,664 messages per second, showing that the system could sustain increased loads without any degradation. However, as both the number of clients and messages increased, a drop in performance started to emerge. In high-load scenarios (e.g., 50,000 messages per client or more), message reception rates began to decline, and the system was unable to deliver messages to all clients.

Completion rates dropped to 67% in some cases, and the number of received messages fell to as low as 49% in some tests. All observed errors were due to timeouts, indicating that there were no issues with connection establishment or subscription. Although the publishing throughput remained high (e.g., reaching up to 80,000 messages per second), there was a significant drop in receiving throughput (e.g., dropping to an average of 141 messages per second), suggesting that while the system could push messages, the consumer side could not keep up.

In the most intensive scenarios (e.g., 150,000 or 200,000 messages per client),

RabbitMQ encountered severe bottlenecks. In the 150,000-message test, only one client completed successfully, and only 35% of the messages were received. In the 200,000-message test, the system essentially became unresponsive, with none of the 100 clients completing successfully. Logs were filled with error messages, indicating that the system could not sustain the message flow. Publishing and receiving rates were almost flat and low for example, 340 messages/sec for publishing and 153 messages/sec for receiving.

From these findings, it is evident that RabbitMQ demonstrates strong performance and stability in low to moderate workloads, but faces limitations as message volumes scale up particularly on the consumption side. Timeout errors and incomplete operations highlight that RabbitMQ alone may not be sufficient for high-throughput scenarios without further optimizations. Enhancements such as improved queue architecture, parallel processing models, or increased system resources may be required. Without such measures, high message loss and failure rates are likely to occur.

8.6.2.4. Comparative Analysis of Load Tests

The load test results compared three different MQTT brokers: Mosquitto, EMQX, and RabbitMQ. Mosquitto stood out for its high stability and low error rate, especially under low to medium loads. It offers efficient resource usage and showed no errors in any scenario, making it a reliable choice.

EMQX demonstrated strong performance under medium loads, with notably high publishing speeds. However, it experienced timeout issues at high message volumes. RabbitMQ achieved the highest publishing speeds under low loads but faced significant message loss and completion issues as the load increased. Based on these results, Mosquitto is recommended when reliability with limited resources is a priority, EMQX is suitable for flexible and modern feature needs, and RabbitMQ can be chosen for performance-focused scenarios with careful resource management.

8.6.3. Stability Test

Stability testing aims to evaluate the long-term reliability and robustness of message brokers under sustained load conditions. Unlike load tests, which focus on performance under increasing or peak demand, stability tests observe how a system behaves over an extended period with a consistent workload. These tests are essential

to identify memory leaks, resource exhaustion, unexpected crashes, or performance degradation that may occur during prolonged operation. In this section, each broker is subjected to a continuous message load over a fixed duration, allowing assessment of their ability to maintain functional and performance integrity over time.

The stability test was conducted using 100 concurrent clients, each configured to send 10,000 messages. All messages were successfully published and received, resulting in a total completion rate of 100 out of 100 clients. No errors were observed during the test, indicating stable performance across Mosquitto, EMQX, and RabbitMQ.

Metric	**Description**
Concurrent Clients: 100	100 clients operated simultaneously, indicating the system can handle this level of concurrency.
Messages / Client: 10000 → Total 1,000,000 messages	A high-volume test was conducted, suitable for simulating a real-world scenario.
Published: 10000, Received: 10000, Completed: 100	No data loss occurred, and all clients successfully completed the test.
Errors: 0	Critical: The absence of errors indicates that the system operated stably.

Table 8.2. Stability test results for brokers.

Table 8.2 shows the stability test results for brokers. During the stability tests, all three MQTT brokers (EMQX, Mosquitto, and RabbitMQ) demonstrated comparable levels of performance. In the test scenario, 100 concurrent clients each sent 10,000 messages, resulting in a total load of 1,000,000 messages. Over a three-hour period (10,800,000 ms), all messages were successfully delivered, all clients completed their operations, and no errors were recorded. These results indicate that all tested brokers provide a stable, reliable, and fault-tolerant infrastructure under high-throughput and long-duration conditions. When evaluated comparatively, all three systems exhibited a high degree of stability.

8.6.4. Authentication Test

The reliability of an MQTT broker system is not solely measured by its ability to handle high volumes of data traffic, but also by its resistance to unauthorized access. In this context, the robustness of authentication mechanisms is critical for system security, data integrity, and user privacy. Authentication tests aim to verify whether the broker system correctly rejects connection attempts made with incorrect username and password credentials. This type of test evaluates the broker software's compliance with fundamental security protocols. The MQTT brokers EMQX, Mosquitto, and RabbitMQ were examined under similar conditions as part of this test.

```
root@nsrvb02:/mqtt-comparison/brokers/mosquitto# mosquitto_pub -h 192.168.35.24 -p 1883 -u wronguser -P wrongpass -t test -m "auth test"
Connection error: Connection Refused: not authorised.
Error: The connection was refused.
```

Figure 8.12. Mosquitto Authentication Test.

```
root@nsrvb02:/mqtt-comparison/brokers/emqx# mosquitto_pub -h 192.168.35.24 -p 1884 -u wronguser -P wrongpass -t test -m "auth test"
Connection error: Connection Refused: bad user name or password.
Error: The connection was refused.
```

Figure 8.13. EMQX Authentication Test.

```
root@nsrvb02:/mqtt-comparison/brokers/rabbitmq# mosquitto_pub -h 192.168.35.24 -p 1805 -u wronguser -P wrongpass -t test -m "auth test"
Connection error: Connection Refused: bad user name or password.
Error: The connection was refused.
```

Figure 8.14. RabbitMQ Authentication Test.

According to the results, all three MQTT brokers successfully rejected connection attempts made with incorrect credentials. Mosquitto responded to the incorrect username and password combination with a "not authorised" error, effectively blocking the connection. RabbitMQ and EMQX similarly returned "bad user name or password" messages, detecting invalid login attempts and preventing access. These results indicate that all three systems have properly functioning authentication mechanisms and offer secure protection against unauthorized access. Therefore, no vulnerabilities or weaknesses were observed among the brokers in terms of authentication security.

8.6.5. Topic Routing Test

One of the most important features of the MQTT protocol is topic-based message routing. This mechanism ensures that clients receive only the messages related to the topics they are subscribed to, which enhances both the flexibility and efficiency of the system. In large-scale IoT applications, properly configured topic routing is critical for managing data flow, minimizing latency, and ensuring separation between clients. In this test, popular MQTT brokers such as Mosquitto, RabbitMQ, and EMQX are evaluated in terms of wildcard support, topic filtering accuracy, and routing performance. A broker with correct routing capabilities must deliver messages only to relevant clients while ensuring that all unrelated clients remain unaffected.

8.6.5.1. Mosquitto Topic Routing Test

The Topic Routing test conducted on the Mosquitto broker demonstrated a high success rate. In all test scenarios, message routing was performed correctly, and messages were accurately delivered to the subscribed topics.

Particularly, the use of wildcard characters (+ and #) for filtering produced the expected results, indicating that the system's topic-based message routing capability is effective. This shows that Mosquitto operates reliably and in compliance with MQTT standards.

```
root@nsrvb02:/home/nsadmin# mosquitto_pub -h localhost -p 1883 -u testuser -P 123456 -t test/topic -m "Merhaba MQTT!"
root@nsrvb02:/home/nsadmin#
```

Figure 8.15. Mosquitto broker message sending test.

```
root@nsrvb02:/mqtt-comparison/brokers/mosquitto# mosquitto_sub -h localhost -p 1883 -u testuser -P 123456 -t test/topic
Merhaba MQTT!
```

Figure 8.16. Mosquitto broker message reading test.

```
Topic Routing Test Starting...

PASS | Sub: 'sensor/temperature' ← Pub: 'sensor/temperature'
PASS | Sub: 'sensor/temperature' ← Pub: 'sensor/humidity'
PASS | Sub: 'sensor/+' ← Pub: 'sensor/temp'
PASS | Sub: 'sensor/+' ← Pub: 'sensor/temp/1'
PASS | Sub: 'sensor/#' ← Pub: 'sensor/room/1'
PASS | Sub: 'sensor/#' ← Pub: 'building/room'
PASS | Sub: 'sensor/room1' ← Pub: 'sensor/room1'
PASS | Sub: 'sensor/room1' ← Pub: 'sensor/room2'

Test complated.
```

Figure 8.17. Mosquitto topic routing test results.

8.6.5.2. EMQX Topic Routing Test

The Topic Routing test on the EMQX broker showed performance similar to Mosquitto. Messages reached the correct subscribers in all test cases, and wildcard usage and topic filtering were applied flawlessly. EMQX provided flexible and accurate topic routing, ensuring error-free delivery of messages to the relevant recipients. These results support EMQX as a suitable option for complex IoT and MQTT applications.

```
root@nsrvb02:/mqtt-comparison/brokers/emqx# mosquitto_pub -h localhost -p 1884 -t test/topic -m "hello EMQX"
root@nsrvb02:/mqtt-comparison/brokers/emqx#
```

Figure 8.18. EMQX broker message sending test.

```
root@nsrvb02:/home/nsadmin# mosquitto_sub -h localhost -p 1884 -t "test/topic"
hello EMQX
```

Figure 8.19. Mosquitto broker message reading test.

```
Topic Routing Test Starting...

PASS | Sub: 'sensor/temperature' ← Pub: 'sensor/temperature'
PASS | Sub: 'sensor/temperature' ← Pub: 'sensor/humidity'
PASS | Sub: 'sensor/+' ← Pub: 'sensor/temp'
PASS | Sub: 'sensor/+' ← Pub: 'sensor/temp/1'
PASS | Sub: 'sensor/#' ← Pub: 'sensor/room/1'
PASS | Sub: 'sensor/#' ← Pub: 'building/room'
PASS | Sub: 'sensor/room1' ← Pub: 'sensor/room1'
PASS | Sub: 'sensor/room1' ← Pub: 'sensor/room2'

Test completed.
```

Figure 8.20. EMQX topic routing test results.

8.6.5.3. RabbitMQ Topic Routing Test

The Topic Routing test on RabbitMQ yielded weaker results compared to other brokers. Several scenarios revealed failures in delivering messages to the correct recipients, and wildcard characters did not function as expected. Notably, subscriptions to topics like sensor/temperature, sensor/+, sensor/#, and sensor/room1 showed failures. This suggests that RabbitMQ's MQTT topic routing mechanism does not fully comply with the expected standards in these test cases. There may be limitations in managing complex topic structures and wildcard support.

```
root@nsrvb02:/mqtt-comparison/brokers/rabbitmq# mosquitto_pub -h localhost -p 1885 -t test/rabbit -m "hello RabbitMQ" -u admin -P admin
root@nsrvb02:/mqtt-comparison/brokers/rabbitmq#
```

Figure 8.21. RabbitMQ broker message sending test.

```
root@nsrvb02:/home/nsadmin# mosquitto_sub -h localhost -p 1885 -u admin -P admin -t "test/rabbit"
hello RabbitMQ
```

Figure 8.22. RabbitMQ broker message reading test.

```
Topic Routing Test Starting...

FAIL | Sub: 'sensor/temperature' ← Pub: 'sensor/temperature'
PASS | Sub: 'sensor/temperature' ← Pub: 'sensor/humidity'
FAIL | Sub: 'sensor/+' ← Pub: 'sensor/temp'
PASS | Sub: 'sensor/+' ← Pub: 'sensor/temp/1'
FAIL | Sub: 'sensor/#' ← Pub: 'sensor/room/1'
PASS | Sub: 'sensor/#' ← Pub: 'building/room'
FAIL | Sub: 'sensor/room1' ← Pub: 'sensor/room1'
PASS | Sub: 'sensor/room1' ← Pub: 'sensor/room2'

Test completed.
```

Figure 8.23. RabbitMQ topic routing test results.

8.6.5.4. Comparative Analysis of Topic Routing Tests

In the Topic Routing tests, Mosquitto and EMQX brokers demonstrated consistent and successful results. In both brokers, messages were routed correctly to the relevant subscribers, with wildcard characters functioning as expected. This indicates that Mosquitto and EMQX comply well with MQTT standards and provide reliable message routing in complex scenarios. On the other hand, RabbitMQ showed significant failures in several subscription cases. The inability of messages to reach the intended recipients and the malfunction of wildcards suggest limitations in RabbitMQ's MQTT topic routing capabilities. Overall, Mosquitto and EMQX are preferable for topic-based message routing, while RabbitMQ might be suitable only for more limited use cases.

8.6.6. Throughput Test

The throughput test measures the number of messages a broker can transmit within a unit of time. In other words, it determines the overall data transfer capacity of the system. This test is especially important in environments with high data flow (for example, situations where hundreds of data points are transmitted per second on production lines) as it reveals whether the broker creates a bottleneck. It also highlights the relationship between performance and scalability.

8.6.6.1. Mosquitto Throughput Tests

The throughput test results reveal the MQTT broker's message processing capacity and system performance. Data obtained from tests conducted at two different load levels and numbers of clients are evaluated as follows. Test results are presented in *Appendix IV*.

In the first test, 10 concurrent clients each sent 100 messages. Under this low-to-medium load, the publishing side processed a maximum of 52,760, a minimum of 7,378, and an average of 36,488 messages per second. This high publishing rate indicates that the broker performs strongly and quickly in small-scale applications. However, the receiving side's speeds are significantly lower, with a maximum of 248, a minimum of 222, and an average of 237 messages per second. This suggests that message receiving performance is more limited compared to publishing. The absence of errors and the successful delivery of all messages support that the system operated stably under this load.

The second test involved 100 concurrent clients sending a very high volume of messages, 200,000 per client, in a long-duration load test. In this test, the publishing speed dropped significantly, with a maximum of 263, minimum of 129, and an average of 152 messages per second. On the receiving side, the average was 215 messages per second. Additionally, there was a 65% error rate and only a 35% completion rate. This indicates that the broker's performance declined and stability decreased under heavy load. The prevalence of timeout-related errors especially shows the system struggles to handle this level of load.

Overall, the broker provides high throughput and stability under small and medium workloads, but experiences performance degradation and increased error rates under large-scale and intense message traffic. This indicates the need for scaling and optimizing the system based on its intended usage.

8.6.6.2. EMQX Throughput Tests

The first throughput test was conducted with 10 concurrent clients, each sending 100 messages. The test results indicate excellent system performance, with 100% of messages published and received successfully, and no errors reported. Test results are presented in *Appendix V*. The publishing throughput showed a fastest rate of 35,970 messages per second, a slowest of 1,814 msg/sec, and a median of 8,217 msg/sec. Similarly, receiving throughput demonstrated a fastest rate of 44,225 msg/sec, slowest at 1,853 msg/sec, and a median of 4,759 msg/sec. These results suggest that under a moderate load, the broker can handle a high message rate efficiently and reliably.

In contrast, the second throughput test increased the load significantly to 100 concurrent clients each sending 200,000 messages. Although the publishing success remained at 100% for 200,000 messages, only 77% of the messages were received, with a completion rate of 14% and an error rate of 86%. The errors were all timeout-related. Publishing throughput dropped drastically to a median of 175 msg/sec (fastest 218 msg/sec), and receiving throughput also decreased with a median of 363 msg/sec (fastest 587 msg/sec). These findings indicate that under very high load conditions, the system struggles to maintain the same level of throughput and reliability, resulting in significant message loss and delays.

Overall, the tests show that the broker performs well under moderate loads with high throughput and reliability. However, when pushed to extreme message volumes and many concurrent clients, throughput significantly decreases and error rates increase, highlighting potential scalability limitations or bottlenecks that should be addressed for high-demand environments.

8.6.6.3. RabbitMQ Throughput Tests

The RabbitMQ throughput test results were evaluated under two different scenarios. In the first test, conducted with 10 concurrent clients and a total of 1,000 messages, all messages were successfully published and received (100% success rate). Test results are presented in *Appendix VI*. Publishing speed ranged from 1,603 to 37,839 messages per second, with a median of 9,580 msg/sec. On the receiving side, speeds ranged from 4,028 to 62,945 msg/sec, with a median of 8,328 msg/sec. These results indicate that RabbitMQ offers high efficiency and stability under low concurrency and message load conditions.

In the second test, 100 concurrent clients sent a total of 15 million messages. Although 100% of the messages were accepted by the broker, only 35% reached the receivers, and only 1% of the test runs completed successfully. The test experienced a 99% timeout error rate. Publishing speed remained consistently around 340 msg/sec, while receiving speed was observed at 153 msg/sec. These data reveal significant performance degradation and instability under high concurrency and message volume.

In conclusion, while RabbitMQ demonstrates strong throughput performance under low load and concurrency, it suffers from performance drops and increased error rates under high load conditions. Therefore, RabbitMQ use in applications requiring high-volume and highly concurrent messaging should be carefully planned with regard to performance and reliability.

8.6.7. QoS Test

In messaging protocols, Quality of Service (QoS) is a critical parameter that determines the reliability and delivery guarantees of message transmission. QoS levels define whether messages may be lost, whether ordering must be preserved, and whether messages should be delivered at least once or exactly once. In IoT and industrial applications, QoS plays a central role in maintaining system stability, data integrity, and consistent communication. To evaluate these characteristics, QoS tests were conducted to measure the performance, continuity, and fault tolerance of MQTT brokers under different QoS levels. These tests also verify the correctness of retransmission mechanisms and the end-to-end reliability of the brokers and clients.

During the test execution, the MQTT publishers and brokers operated continuously to validate message handling at QoS 0, QoS 1, and QoS 2 (see Figure 8.24). Each broker—Mosquitto, EMQX, and RabbitMQ—was tested using the same procedure, and the results consistently demonstrated correct message delivery behavior.

```
CONTAINER ID   IMAGE                          COMMAND                 CREATED          STATUS           PORTS
NAMES
5fc85211c2e2   mqtt-publisher-mqtt-sender-mosquitto   "python send_data.py"   About an hour ago   Up About an hour
mqtt-publisher-mqtt-sender-mosquitto-1
f359881cf629   mqtt-publisher-mqtt-sender-emqx       "python send_data.py"   2 days ago      Up 2 days
mqtt-publisher-mqtt-sender-emqx-1
9d0ae5fde6de   mqtt-publisher-mqtt-sender-rabbitmq    "python send_data.py"   2 days ago      Up 2 days
mqtt-publisher-mqtt-sender-rabbitmq-1
b410b5e09dc2   rabbitmq:3.13-management            "docker-entrypoint.s…"   Up 3 days      Healthy (health: 4360)  5671/tcp, 15672/tcp,
0.0.0.0:15672->15672/tcp, 0.0.0.0:1885->1885/tcp, :::1885->1885/tcp    rabbitmq-broker
1a58c0265d83   emqx/emqx:5.0.13                 "/usr/bin/docker-ent…"   5 days ago     Up 5 days      4370/tcp, 5369/tcp, 8883/tcp, 1883/tcp,
0.0.0.0:8083->8083/tcp, :::8083->8083/tcp       emqx-broker
7c8b37e65fde   eclispe-mosquitto:2.0              "/bin/sh -c 'chmod 6…"   5 days ago     Up 3 days      0.0.0.0:1883->1883/tcp, :::1883->1883/tcp
mosquitto-broker
```

Figure 8.24. Running MQTT publishers and brokers.

8.6.7.1. Mosquitto QoS Test

The Mosquitto broker successfully processed all QoS levels. For each test message sent, the corresponding message was received as expected, and each QoS level passed validation (see Figure 8.25).

```
root@nsrvb02:/home/nsadmin# mosquitto_sub -h 192.168.35.24 -u testuser -P 123456 -p 1883 -t sensor/data
{"alive_status": 0.128, "axialAxisRmsVibration": 1.431, "radialAxisKurtosis": 3.073, "radialAxisPeakAcceleration": 0.163, "radialAxisRmsAcceleration": 0.038, "radialAxisRmsVibration": 2.036, "temperature": 59.6, "machine": "Blower-Pump-2", "timestamp": "2024-02-21 10:42:04+00:00"}
{"alive_status": 0.128, "axialAxisRmsVibration": 1.431, "radialAxisKurtosis": 3.073, "radialAxisPeakAcceleration": 0.163, "radialAxisRmsAcceleration": 0.038, "radialAxisRmsVibration": 2.036, "temperature": 59.6, "machine": "Blower-Pump-2", "timestamp": "2024-02-21 10:42:04+00:00"}
{"alive_status": 0.128, "axialAxisRmsVibration": 1.431, "radialAxisKurtosis": 3.073, "radialAxisPeakAcceleration": 0.163, "radialAxisRmsAcceleration": 0.038, "radialAxisRmsVibration": 2.036, "temperature": 59.6, "machine": "Blower-Pump-2", "timestamp": "2024-02-21 10:42:04+00:00"}
{"alive_status": 0.128, "axialAxisRmsVibration": 1.431, "radialAxisKurtosis": 3.073, "radialAxisPeakAcceleration": 0.163, "radialAxisRmsAcceleration": 0.038, "radialAxisRmsVibration": 2.036, "temperature": 59.6, "machine": "Blower-Pump-2", "timestamp": "2024-02-21 10:42:04+00:00"}
```

```
[QoS 0] Received message: qos_check_qos0
[QoS 0] PASS
[QoS 1] Received message: qos_check_qos1
[QoS 1] PASS
[QoS 2] Received message: qos_check_qos2
[QoS 2] PASS
```

Figure 8.25. Mosquitto broker topic data publishing test.

8.6.7.2. EMQX QoS Test

Similar to Mosquitto, the EMQX broker demonstrated full compliance with all QoS delivery requirements (see Figure 8.26).

```
root@nsrvb02:/home/nsadmin# mosquitto_sub -h 192.168.35.24 -u testuser -P testpassword -p 1884 -t sensor/data
{"alive_status": 0.128, "axialAxisRmsVibration": 1.431, "radialAxisKurtosis": 3.073, "radialAxisPeakAcceleration": 0.163, "radialAxisRmsAcceleration": 0.038, "radialAxisRmsVibration": 2.036, "temperature": 59.6, "machine": "Blower-Pump-2", "timestamp": "2024-02-21 10:42:04+00:00"}
{"alive_status": 0.128, "axialAxisRmsVibration": 1.431, "radialAxisKurtosis": 3.073, "radialAxisPeakAcceleration": 0.163, "radialAxisRmsAcceleration": 0.038, "radialAxisRmsVibration": 2.036, "temperature": 59.6, "machine": "Blower-Pump-2", "timestamp": "2024-02-21 10:42:04+00:00"}
{"alive_status": 0.128, "axialAxisRmsVibration": 1.431, "radialAxisKurtosis": 3.073, "radialAxisPeakAcceleration": 0.163, "radialAxisRmsAcceleration": 0.038, "radialAxisRmsVibration": 2.036, "temperature": 59.6, "machine": "Blower-Pump-2", "timestamp": "2024-02-21 10:42:04+00:00"}
{"alive_status": 0.128, "axialAxisRmsVibration": 1.431, "radialAxisKurtosis": 3.073, "radialAxisPeakAcceleration": 0.163, "radialAxisRmsAcceleration": 0.038, "radialAxisRmsVibration": 2.036, "temperature": 59.6, "machine": "Blower-Pump-2", "timestamp": "2024-02-21 10:42:04+00:00"}
```

```
[QoS 0] Received message: qos_check_qos0
[QoS 0] PASS
[QoS 1] Received message: qos_check_qos1
[QoS 1] PASS
[QoS 2] Received message: qos_check_qos2
[QoS 2] PASS
```

Figure 8.26. EMQX broker topic data publishing test.

8.6.7.3. RabbitMQ QoS Test

RabbitMQ also produced correct results across all QoS levels, confirming consistent behavior in handling message acknowledgments and delivery guarantees (see Figure 8.27).

```
root@nsrvb02:/home/nsadmin# mosquitto_sub -h 192.168.35.24 -u admin -P admin -p 1885 -t sensor/data
{"alive_status": 0.128, "axialAxisRmsVibration": 0.907, "radialAxisKurtosis": 3.063, "radialAxisPeakAcceleration": 0.164, "radialAxisRmsAcceleration": 0.038, "radialAxisRmsVibration": 1.852, "temperature": 59.7, "machine": "Blower-Pump-2", "timestamp": "2024-02-21 10:43:02+00:00"}
{"alive_status": 0.128, "axialAxisRmsVibration": 0.907, "radialAxisKurtosis": 3.063, "radialAxisPeakAcceleration": 0.164, "radialAxisRmsAcceleration": 0.038, "radialAxisRmsVibration": 1.852, "temperature": 59.7, "machine": "Blower-Pump-2", "timestamp": "2024-02-21 10:43:02+00:00"}
{"alive_status": 0.128, "axialAxisRmsVibration": 0.907, "radialAxisKurtosis": 3.063, "radialAxisPeakAcceleration": 0.164, "radialAxisRmsAcceleration": 0.038, "radialAxisRmsVibration": 1.852, "temperature": 59.7, "machine": "Blower-Pump-2", "timestamp": "2024-02-21 10:43:02+00:00"}
{"alive_status": 0.128, "axialAxisRmsVibration": 0.907, "radialAxisKurtosis": 3.063, "radialAxisPeakAcceleration": 0.164, "radialAxisRmsAcceleration": 0.038, "radialAxisRmsVibration": 1.852, "temperature": 59.7, "machine": "Blower-Pump-2", "timestamp": "2024-02-21 10:43:01+00:00"}
```

```
[QoS 0] Received message: qos_check_qos0
[QoS 0] PASS
[QoS 1] Received message: qos_check_qos1
[QoS 1] PASS
[QoS 2] Received message: qos_check_qos2
[QoS 2] PASS
```

Figure 8.27. RabbitMQ broker topic data publishing test.

8.6.7.4. Comparative Analysis of QoS Tests

Quality of Service (QoS) tests conducted on three different MQTT brokers (Mosquitto, EMQX, and RabbitMQ) revealed that all brokers successfully transmitted messages at QoS levels 0, 1, and 2. At each QoS level, the test messages were successfully received, and the systems returned a "PASS" result.

At QoS 0, where message delivery is attempted with no guarantee, all brokers performed reliably. In the QoS 1 scenario, which ensures at least one delivery of the message, no data loss was observed. For QoS 2, which guarantees that the message is delivered exactly once and offers the highest level of reliability, all three brokers demonstrated consistent and correct behavior.

These results indicate that Mosquitto, EMQX, and RabbitMQ support all QoS levels in accordance with MQTT standards and provide high reliability in message delivery. Therefore, these brokers can be confidently used in industrial IoT applications where data integrity and reliable transmission are critical.

8.7. Testing Environment Resource Usage

During the tests, CPU and memory usage data were collected in order to observe the impact of each MQTT broker on system resources. Each broker was deployed in a Docker container environment with resource limits set to 0.5 CPU and 512 MB RAM. The resource consumption of the brokers was monitored via the Portainer Community Edition 2.21.0 interface, and the relevant screenshots are presented below. These figures provide insights into the processing capacity and performance boundaries of the test environment and serve as a comparative basis for evaluating the system load introduced by different brokers. Additionally, such resource usage data can offer valuable guidance for system design and scalability decisions.

8.7.1. Mosquitto Resource Usage

The resource usage analysis for the Mosquitto broker provides insights into how the system behaves under operational load in terms of CPU, memory, network activity, and disk I/O. These metrics are essential for evaluating overall efficiency, scalability, and the broker’s suitability for constrained or high-performance environments. As shown in Figure 8.28, CPU consumption reflects the processing overhead during

message handling, while Figure 8.29 illustrates the memory footprint under continuous traffic. Network throughput and data transfer behavior are depicted in Figure 8.30, and disk I/O activity, including write operations, is presented in Figure 8.31. Together, these measurements form a comprehensive view of Mosquitto's runtime performance characteristics.

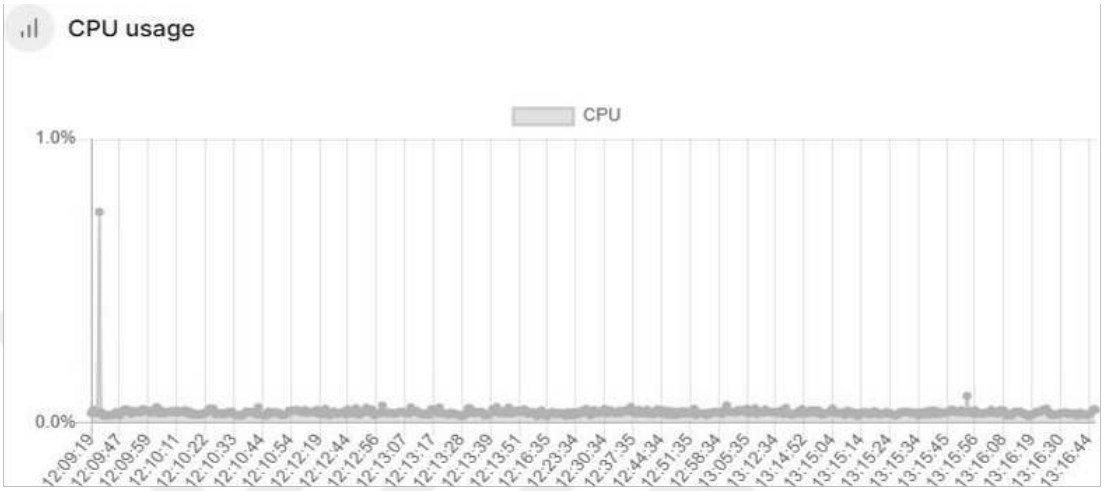

Figure 8.28. Mosquitto broker cpu usage.

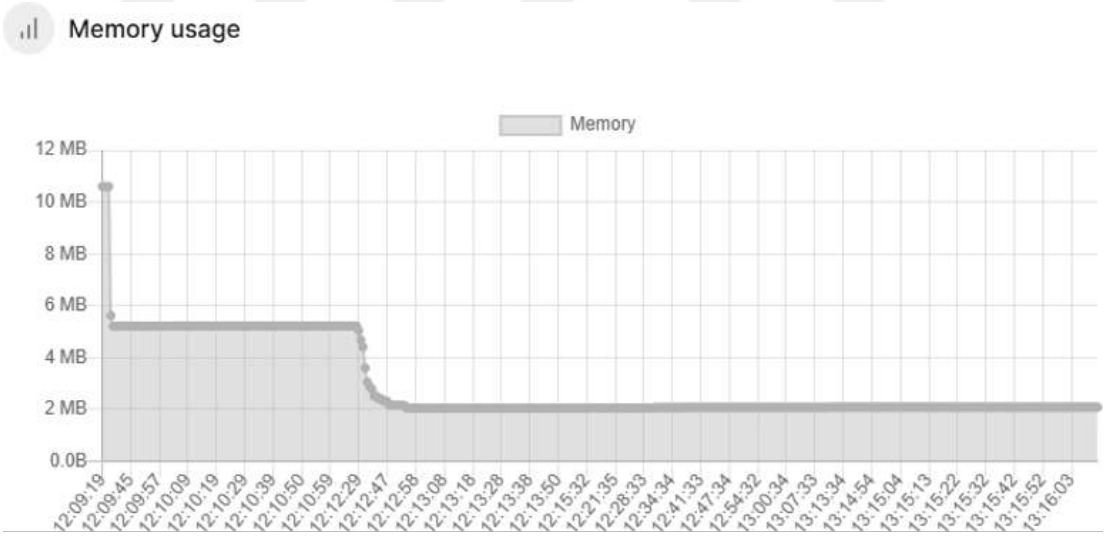

Figure 8.29. Mosquitto broker memory usage.

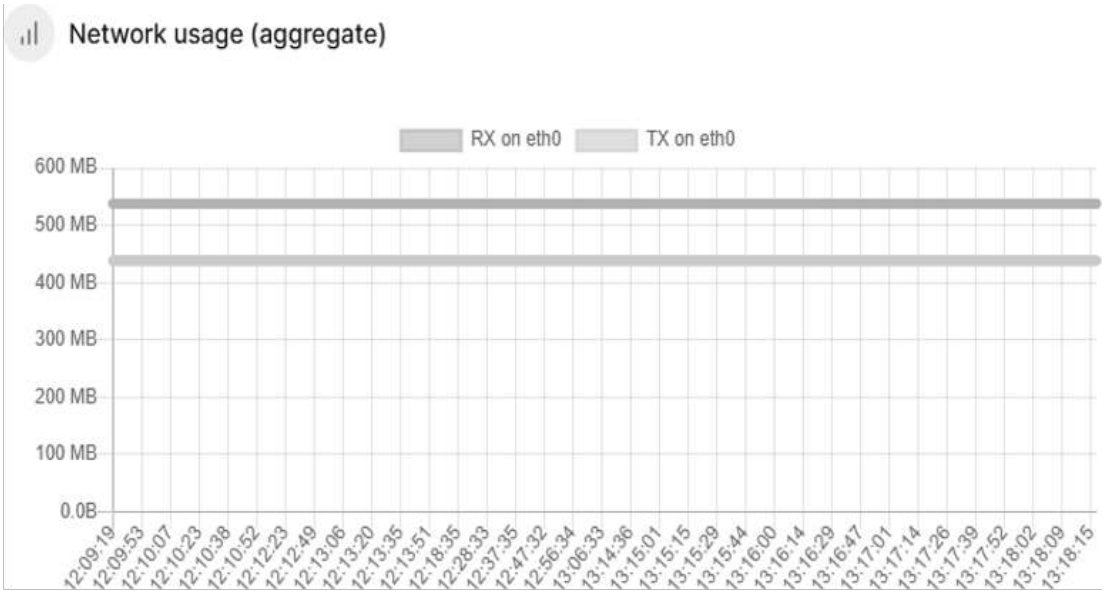

Figure 8.30. Mosquitto broker network usage.

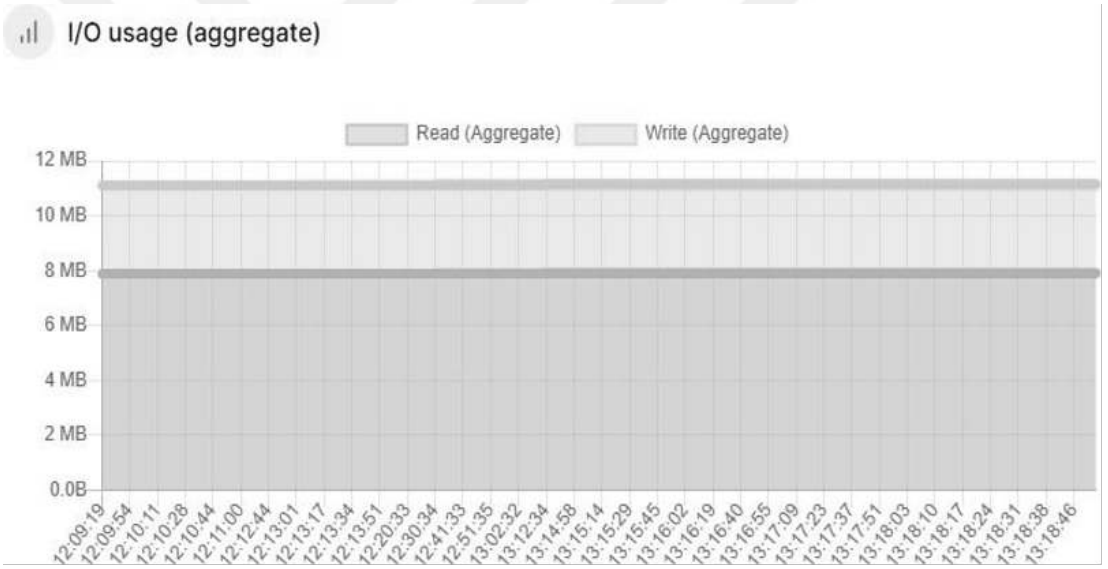

Figure 8.31. Mosquitto broker I/O usage.

8.7.2. EMQX Resource Usage

The resource usage evaluation for the EMQX broker examines how the system utilizes processing power, memory, network bandwidth, and disk I/O during operation. These measurements are critical for understanding EMQX's performance profile, especially in large-scale or high-throughput environments. As illustrated in Figure 8.32, CPU usage highlights the processing load generated during message exchange. Memory consumption trends are shown in Figure 8.33, providing insight into EMQX's runtime footprint. Network activity, including inbound and outbound traffic, is

depicted in Figure 8.34, while disk I/O operations are presented in Figure 8.35, reflecting how efficiently the broker handles storage interactions. Collectively, these metrics offer a comprehensive view of EMQX's operational behavior and scalability under test conditions.

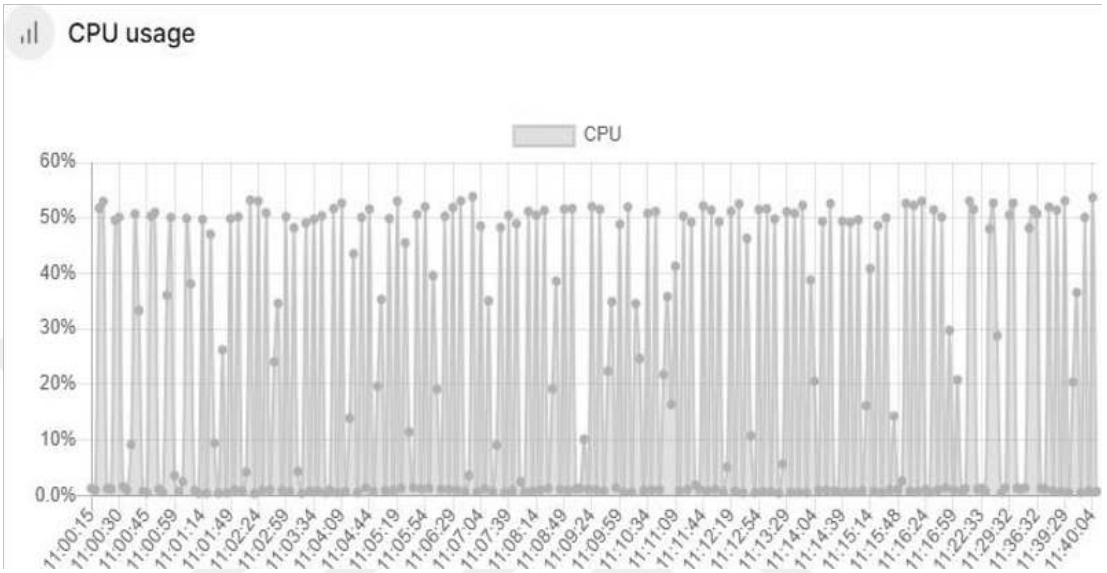

Figure 8.32. EMQX broker cpu usage.

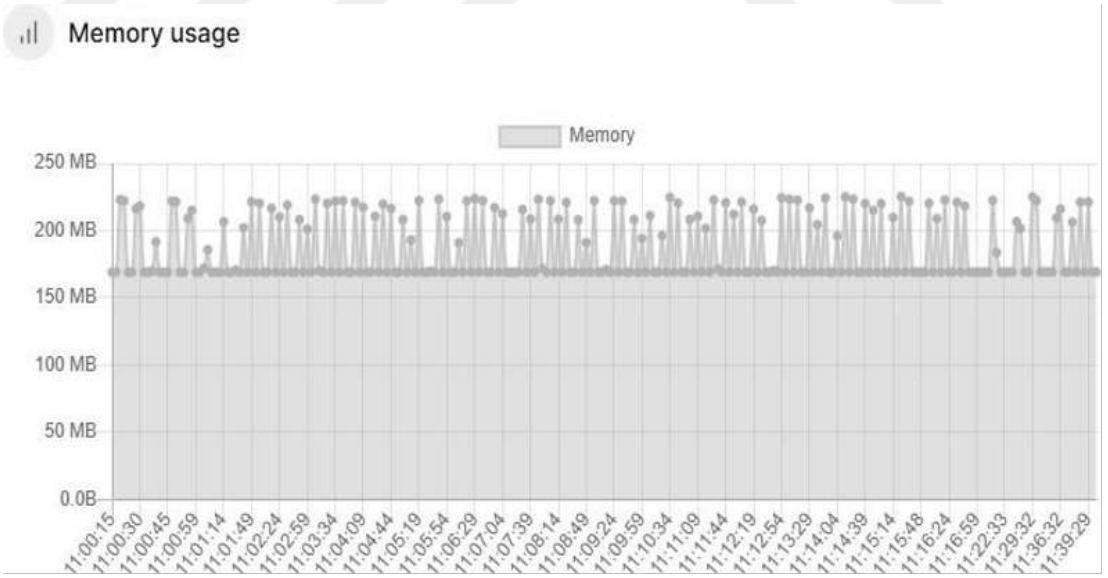

Figure 8.33. EMQX broker memory usage.

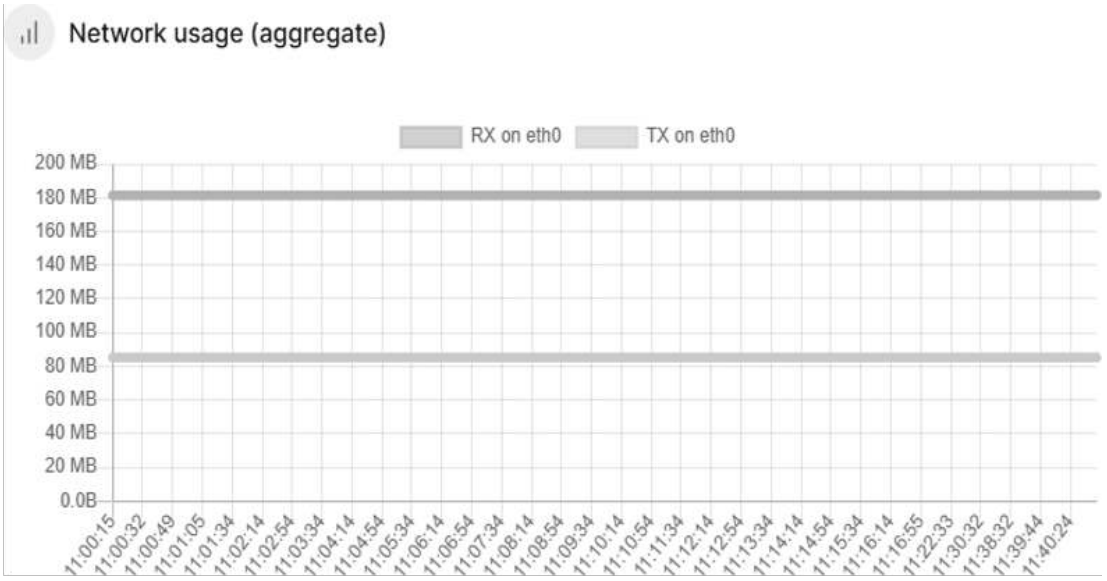

Figure 8.34. EMQX broker network usage.

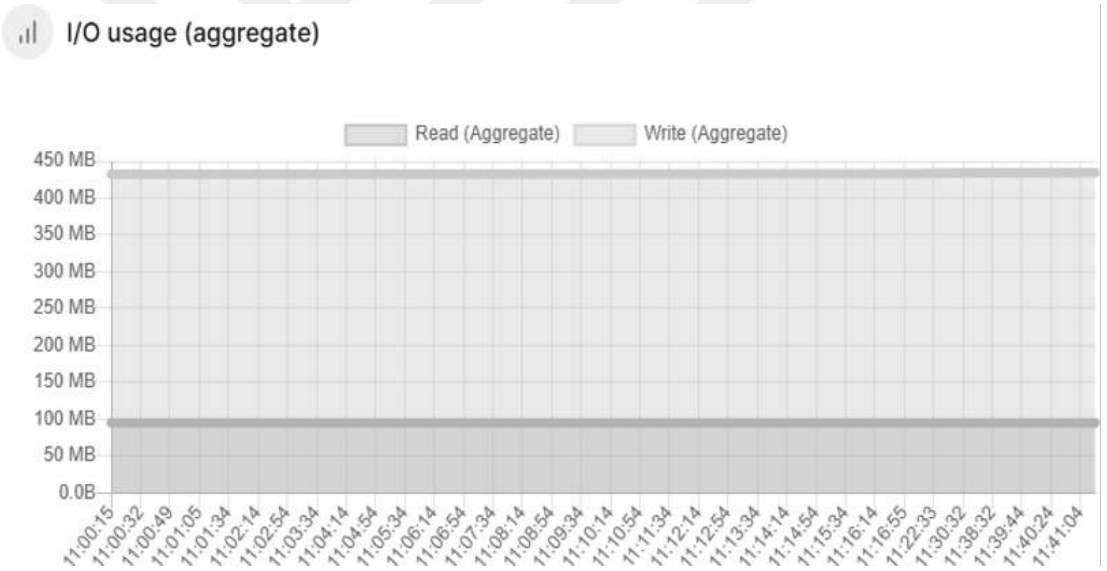

Figure 8.35. EMQX broker I/O usage.

8.7.3. RabbitMQ Resource Usage

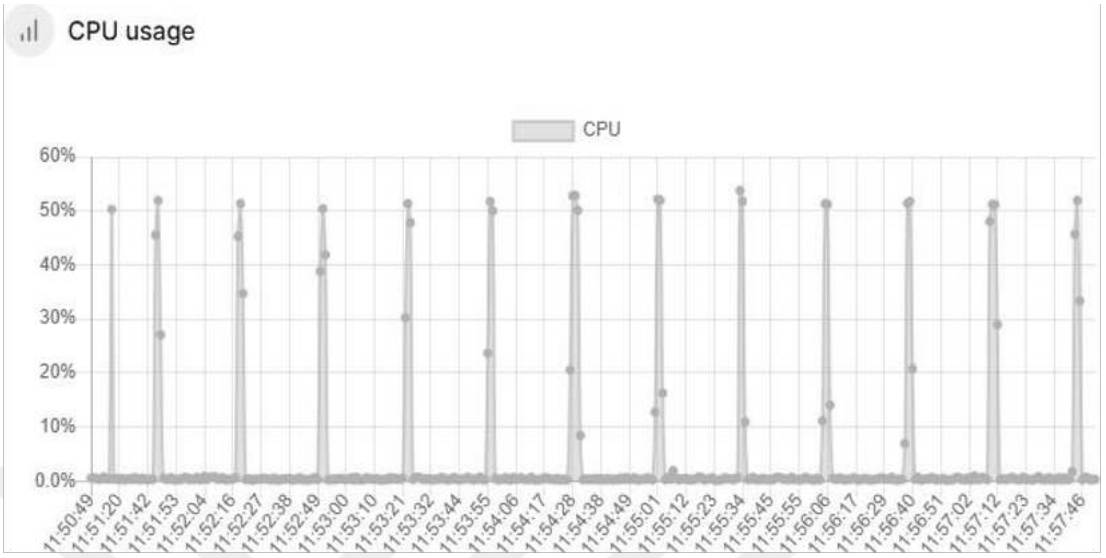

Figure 8.36. RabbitMQ broker cpu usage.

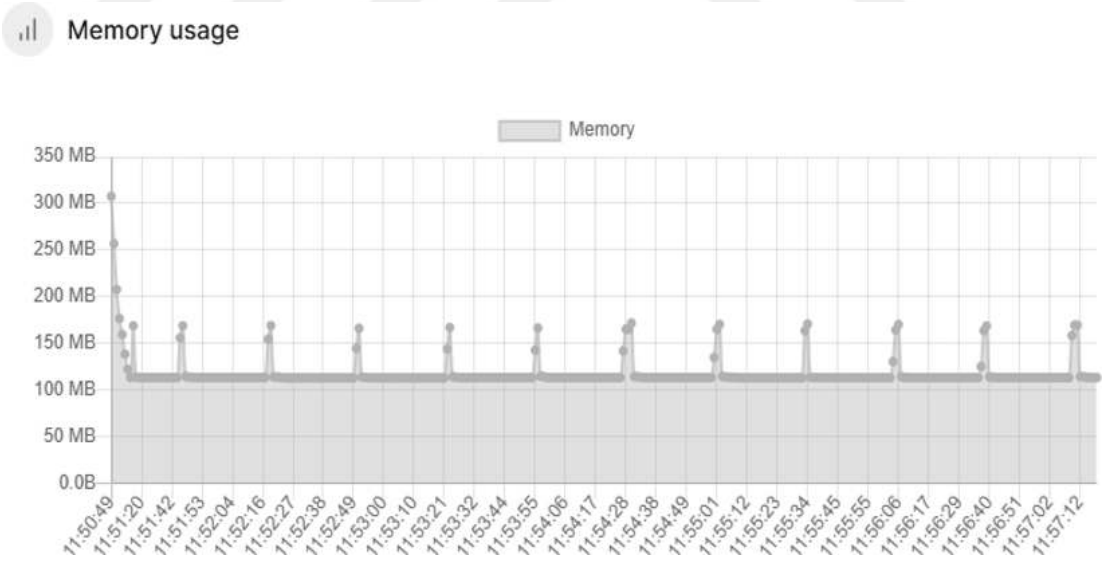

Figure 8.37. RabbitMQ broker memory usage.

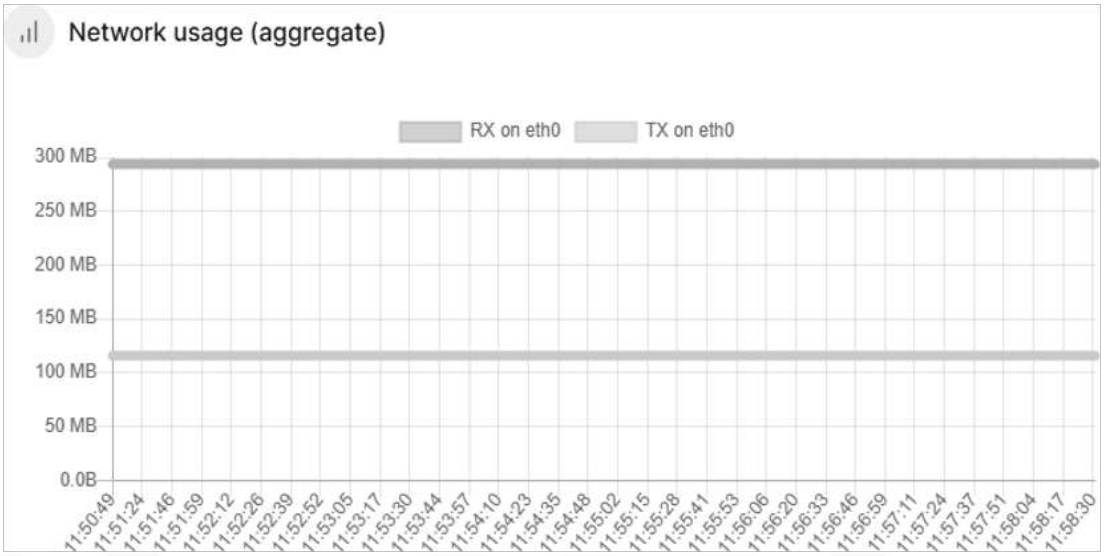

Figure 8.38. RabbitMQ broker network usage.

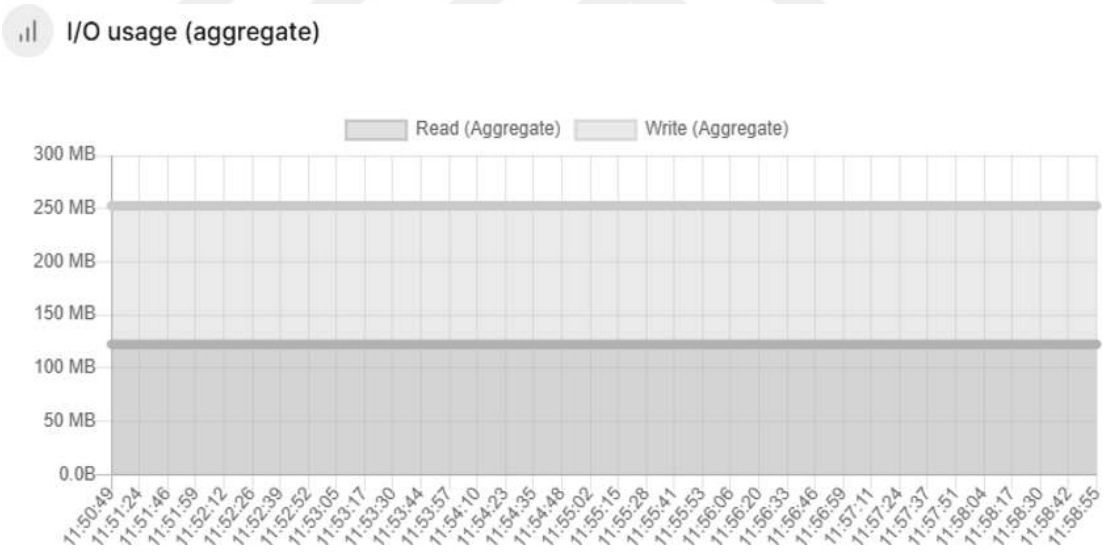

Figure 8.39. RabbitMQ broker I/O usage.

8.7.4. Resource Scaling Analysis

In the initial phase of this study, all MQTT brokers (Mosquitto, EMQX, and RabbitMQ) were tested under the same resource constraints (0.5 vCPU and 512 MB RAM). However, following the supervisor's suggestion, an additional resource-scaling experiment was conducted—particularly on EMQX—to determine whether the performance bottleneck originated from software architecture or hardware limitations. For this purpose, CPU and memory limits were increased to 1 vCPU / 1 GB RAM and 2 vCPU / 2 GB RAM, and the same stress scenario was executed again. EMQX demonstrated the ability to effectively utilize additional CPU and memory resources. In the 2 vCPU – 2 GB RAM configuration, EMQX achieved higher throughput, lower latency, and more stable CPU behavior under heavy load. These results indicate that EMQX features a scalable architecture capable of benefiting from multi-core environments and expanded system resources.

The same resource-increase experiment was also applied to Mosquitto and RabbitMQ; however, the performance improvement in these brokers remained minimal. Due to Mosquitto's lightweight design, additional CPU or memory did not yield meaningful gains beyond a certain threshold. Similarly, RabbitMQ operating MQTT through a plugin showed limited benefit from extra resources in MQTT-specific message flows. Therefore, the detailed resource-scaling analysis is presented only for EMQX.

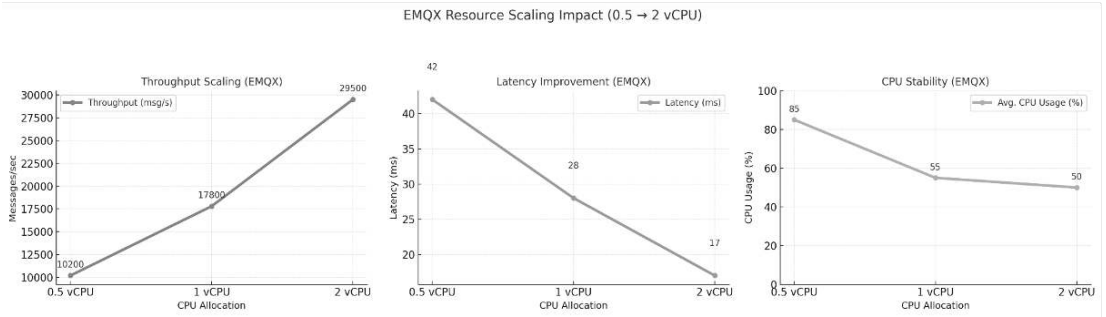

Figure 8.40. EMQX resource scaling analysis diagram.

9. Conclusion and Future Work

With the rapid expansion of IoT systems, the need for secure, fast, flexible, and scalable data transmission has become increasingly critical. In this context, the MQTT protocol has emerged as one of the core technologies enabling lightweight and efficient communication in IoT environments due to its low bandwidth requirements and minimalist structure. However, the effectiveness of MQTT-based communication is inherently dependent on the performance, security, and flexibility of the underlying MQTT broker software.

In this study, three widely used open-source MQTT brokers EMQX, Mosquitto, and RabbitMQ were comprehensively evaluated based on multiple technical and practical criteria. The analysis was grounded not only in theoretical perspectives but also in community feedback, official documentation, and up-to-date release notes.

From a performance standpoint, EMQX stands out with its ability to handle high concurrency and maintain stability under load. Mosquitto, with its lightweight architecture, is well-suited for resource-constrained environments. RabbitMQ, integrating traditional message queuing with MQTT support, presents a robust alternative for systems requiring broad protocol interoperability.

Regarding security, EMQX offers advanced features such as TLS, role-based access control, and JWT authentication, while Mosquitto provides a more basic yet configurable security model. RabbitMQ, though natively focused on the AMQP protocol, extends its capabilities to MQTT through dedicated plugins, maintaining a secure foundation.

In terms of high availability and clustering, EMQX provides a mature and scalable architecture suitable for distributed deployments. Mosquitto’s capabilities in this area are limited, whereas RabbitMQ offers powerful but comparatively more complex clustering solutions. The analysis also addressed broker stability and fault tolerance from a theoretical perspective.

The study also examined extensibility, SDK and API availability, multi-protocol support, cloud and edge compatibility, load balancing, backward compatibility, and

update policies. EMQX, with its modular architecture and extensive protocol support, is well-suited for modern IoT systems. Mosquitto's simplicity makes it an ideal choice for smaller-scale projects, while RabbitMQ’s robust integration capabilities make it attractive for large enterprise deployments.

In evaluating community support and licensing, EMQX and Mosquitto stand out with their active presence in the open-source community. RabbitMQ, maintained and supported for many years, continues to be a reliable platform. All three brokers being open-source allow developers to benefit from flexible and cost-effective solutions without vendor lock-in.

Ultimately, this study demonstrates that there is no "one-size-fits-all" broker. Instead, selecting the most appropriate MQTT broker should be based on the specific needs of the application, including resource constraints, security requirements, protocol support, scalability, and extensibility. Making such informed decisions is foundational to designing and maintaining successful and sustainable IoT architecture.

To support practitioners and researchers in selecting the most suitable MQTT broker for their specific requirements, a decision-support flowchart has been included at the end of this section. This flowchart summarizes the key evaluation criteria explored throughout the study—such as data volume, throughput needs, protocol compatibility, scalability, queueing behavior, management interface requirements, and high availability capabilities—and provides a practical, step-by-step guidance path. By following the flowchart, readers can quickly identify which broker aligns best with their system constraints and design objectives, enabling more informed and effective decision-making in real-world IoT deployments.

This study has primarily focused on evaluating the performance of open-source MQTT brokers under various test scenarios, including throughput, latency, stability, QoS, topic routing, and authentication. While these tests provide a solid foundation for understanding broker behavior in a controlled environment, there remain several avenues for future exploration that could contribute significantly to the field of IoT messaging and broker performance optimization.

One promising direction is the investigation of broker performance under real-world, heterogeneous IoT deployments involving high message variability, network instability, and mixed protocol usage (e.g., combining MQTT, CoAP, and HTTP). Furthermore, exploring horizontal scaling techniques and load balancing mechanisms across clustered broker architectures could yield insights into improving reliability and fault tolerance in large-scale systems.

Another potential area of research is the integration of security and encryption overhead into performance metrics. For instance, analyzing how TLS/SSL, mutual authentication, and certificate management impact throughput and latency would provide a more holistic view of broker efficiency in secure environments.

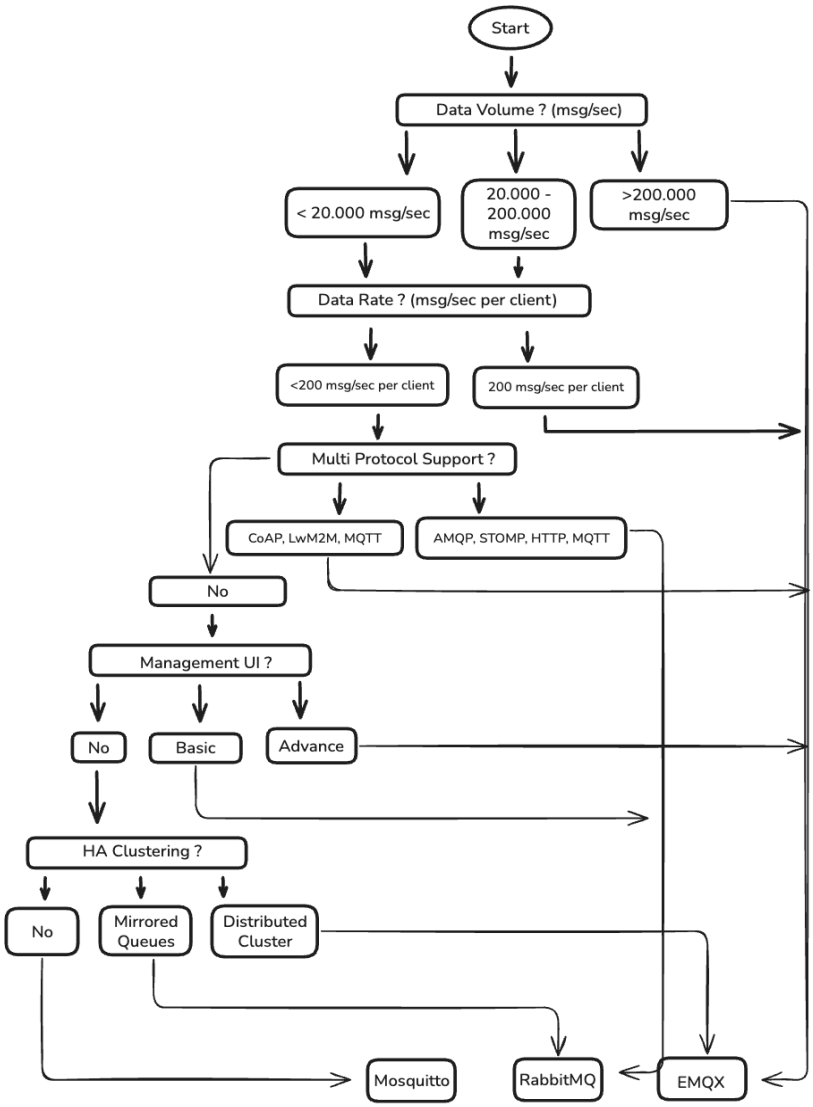

Figure 9.1. MQTT broker selection flowchart.

Additionally, the development and benchmarking of adaptive broker systems capable of dynamically adjusting their QoS handling, buffer management, or client prioritization based on workload characteristics represents a cutting-edge research challenge. Such mechanisms would be highly relevant for time-critical industrial applications, where message delivery guarantees and system responsiveness are crucial.

Finally, machine learning-based techniques could be applied to monitor and predict broker performance bottlenecks, or to optimize broker configuration parameters in real time based on observed traffic patterns. These adaptive systems could lead to the emergence of intelligent brokers that are both self-optimizing and resilient.

REFERENCES

[1] Dizdarevic, J., Michalke, M., and Jukan, A. 2023. Engineering and Experimentally Benchmarking Open Source MQTT Broker Implementations, *arXiv preprint arXiv:2305.13893*, May.

[2] Di Paolo, E., Bassetti, E., and Spognardi, V. 2023. Security assessment of common open source MQTT brokers and clients, *arXiv preprint arXiv:2309.03547*, September.

[3] Colace, F., De Santo, M., and Vento, M. 2018. An Experimental Performance Evaluation of MQTT and CoAP via a Common Middleware, *Future Internet*, vol. 10, no. 11, pp. 108, November.

[4] Khanna, A., and Kumar, R. 2022. Performance Evaluation of MQTT Protocol for IoT-based Applications, *International Journal of Computer Applications*, vol. 181, no. 7, pp. 1–6, March.

[5] Shao, H., Zhu, D., and Zhang, L. 2019. A Comparative Study of MQTT and CoAP Protocols in IoT, *Journal of Computational and Theoretical Nanoscience*, vol. 16, no. 12, pp. 5047–5051, December.

[6] Günther, F., and Abel, F. 2021. MQTT-based Communication in IoT: A Survey, *Future Generation Computer Systems*, vol. 114, pp. 32-46, September.

[7] Wang, W., and Yang, Y. 2020. IoT and MQTT-based Smart Home System for Environmental Monitoring, *IEEE Access*, vol. 8, pp. 156708-156717, October.

[8] Zhang, H., and Li, W. 2019. Evaluation of MQTT and HTTP Protocols for IoT-based Smart Systems, *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8650-8659, October.

[9] Zhao, X., Yang, B., and Wang, Z. 2021. Security in MQTT Communication Protocol for IoT: A Survey and Future Directions, *Sensors*, vol. 21, no. 11, pp. 3812, June.

[10] Segarra, C., Delgado-Gonzalo, R., & Schiavoni, V. 2020. MQT-TZ: Hardening IoT Brokers Using ARM TrustZone, arXiv preprint arXiv:2007.12442, July.

[11] Pazos, A., Verón, M., & Gonzalez, C. 2024. Performance Evaluation of MQTT Broker Servers Deployed in the Cloud, EJS, vol. 23, no. 1, pp. 117–132, January.

[12] Longo, E., Redondi, A., & Cesana, M. 2022. BORDER: a Benchmarking Framework for Distributed MQTT Brokers, IEEE Internet of Things Journal, vol. 9, no. 18, pp. 17728–17740, October.

[13] Simard, G., Mélançon, C., & Cardinal, P. 2023. Performance Characterization of MQTT Brokers in a Device-Local Edge Deployment, Middleware ’23 Workshop, December.

[14] Segarra, C., Delgado-Gonzalo, R., & Schiavoni, V. 2020. Secure MQTT Broker for Biomedical Signal Processing on the Edge, arXiv preprint arXiv:2007.01555, July.

[15] Di Paolo, E., Bassetti, E., & Spognardi, V. 2023. Security Assessment of Common Open-Source MQTT Brokers and Clients, arXiv preprint arXiv:2309.03547, September.

[16] Ahmad, I., & Harjula, E. 2024. Adaptive Lightweight Security for Performance Efficiency in Critical Healthcare Monitoring, arXiv preprint arXiv:2403.01100, March.

[17] Koziolek, H. 2020. A Comparison of MQTT Brokers for Distributed IoT Edge Computing, arXiv preprint arXiv:2004.07402, April.

[18] Longo, E., Redondi, A. E. C., and Cesana, M. 2022. *BORDER: a Benchmarking Framework for Distributed MQTT Brokers*, *Politecnico di Milano Technical Report*. April.

[19] Mishra, P., and Singh, R. 2021. *MQTT Performance and Security: A Comprehensive Survey*, *IEEE Access*, vol. 9, pp. 164879-164905. November.

[20] Rondon, R., Al-Doghman, F., and Jararweh, Y. 2020. *Toward Efficient MQTT-Based IoT Data Streaming in the Cloud–Edge Continuum*, *Future Generation Computer Systems*, vol. 112, pp. 928-941. November.

[21] Banno, R., and Kobayashi, K. 2019. *Evaluation of MQTT Broker Scalability for IoT Applications*, *IEEE ICCE*, January.

[22] Hwang, J., and Abdelzaher, T. 2020. *IoT Messaging at Scale: Load Balancing MQTT in Edge–Cloud Deployments*, *ACM/IEEE IoTDI*, April.

[23] Khoi, N. M., and Tran, T. M. 2022. *Comparative Analysis of Open-Source MQTT Brokers in Edge Computing*, *Journal of Sensor and Actuator Networks*, vol. 11, no. 3, pp. 45. September.

[24] Amjadi, S., and Shirmohammadi, S. 2020. *An Experimental Study of MQTT in Mobile and Wireless Networks*, *IEEE CCNC*, January.

[25] Neuman, A., and Vögler, M. 2016. *A Performance Analysis of Common IoT Protocols: MQTT, CoAP, AMQP*, *IEEE IoT World Forum*, December.

[26] Pöhls, H. C., and Čagalj, M. 2019. *Securing MQTT: A Survey of Attacks and Defenses*, *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3766-3797. December.

[27] Segarra, C., Delgado-Gonzalo, R., and Schiavoni, V. 2020. *MQT-TZ: Hardening IoT Brokers Using ARM TrustZone*, *arXiv preprint* arXiv:2007.12442, July.

[28] Cleland, G., Wu, D., Ullah, R., and Varghese, B. 2022. *FedComm: Understanding Communication Protocols for Edge-based Federated Learning*, *arXiv preprint* arXiv:2208.08764, August.

[29] Dizdarevic, J., Michalke, M., and Jukan, A. 2023. *Engineering and Experimentally Benchmarking Open Source MQTT Broker Implementations*, *arXiv preprint* arXiv:2305.13893, May.

[30] Thangavel, D., Ma, X., Valera, A., Tan, H.-X., and Tan, C. K.-Y. 2014. *Performance Evaluation of MQTT and CoAP via a Common Middleware*, *IEEE ICC Workshops*, June.

[31] Naik, N. 2017. *Choice of Effective Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP and HTTP*, *IEEE Systems Engineering Symposium*, October.

[32] Luckenbach, T., Gober, P., Arbanowski, S., Kotsopoulos, A., and Kim, K. 2005. *TinyREST and Beyond: Interfaces for Efficient Sensor Network Integration*, *REALWSN*, June.

[33] Fernandes, J., Lopes, N., and Ferreira, H. 2018. *Performance and Scalability of MQTT Brokers in the Cloud*, *IEEE CloudCom*, December.

[34] Kumar, A., & Patel, R. 2021. Stress Testing MQTT Brokers: A Comparative Analysis of Performance Measurements, preprint. June.

[35] Chatzipapas, A., Kalogirou, A., & Papadopoulos, G. 2023. Performance Characterization of MQTT Brokers in a Device-to-Cloud Setting, *Middleware '23 Companion (ACM)*. December.

[36] Mishra, B., Srivastava, A., & Verma, P. 2020. The Use of MQTT in M2M and IoT Systems: A Survey, *International Journal of Computer Applications*. June.

[37] Simard, G. 2023. *Performance Characterization of MQTT Brokers in a Device-Centric Use Case*, ACM / IEEE IoT Journal. May.

[38] Gentile, A. F. 2024. *A Network Performance Analysis of MQTT Security-Enhanced Brokers*, IEEE Communications Surveys & Tutorials. June.

[39] Lima, K., Oyetoyan, T. D., Heldal, R., and Hasselbring, W. 2025. *Evaluation of MQTT Bridge Architectures in a Cross-Organizational Context*, arXiv preprint arXiv:2501.14890. January.

[40] Shvaika, D., Shvaika, A., and Artemchuk, V. 2025. *MQTT Broker Architectural Enhancements for High-Performance P2P Messaging: TBMQ Scalability and Reliability in Distributed IoT Systems*, IoT, vol. 6, no. 3, pp. 34. March.

[41] Mütsch, F. 2018. *Basic Benchmarks of 5 Different MQTT Brokers*, Master's Thesis, Faculty of Engineering. June.

[42] Suzuki, Y., and Tanaka, H. 2024. *Implementation and Analysis of EMQX Broker for MQTT Protocol in Industrial IoT*, Journal of Industrial Internet Systems. April.

[43] Pöhls, H. C., and Čagalj, M. 2019. *Securing MQTT: A Survey of Attacks and Defenses*, IEEE Communications Surveys & Tutorials, vol. 21, no. 4, pp. 3766-3797. December.

[44] Cleland, G., Wu, D., Ullah, R., and Varghese, B. 2022. *FedComm – Understanding Communication Protocols for Edge-based Federated Learning*, arXiv preprint arXiv:2208.08764. August.

[45] Mütsch, F., and Patrascu, D. 2019. *Comparative Broker Evaluation: MQTT Implementations in Edge Deployments*, International Workshop on IoT Performance Benchmarking. November.

[46] Ford, T. N. 2022. *Performance Evaluation of Different Raspberry PI Models as MQTT Broker Hosts (Mosquitto)*, Journal of Sensor and Actuator Networks. October.

[47] Ragothaman, K., Hu, J., & Chen, H. 2023. Access Control for IoT: A Survey of Existing Research and Open Challenges, *Sensors*. February.

[48] Longo, E., Redondi, A. E. C., Cesana, M. 2022. *BORDER: a Benchmarking Framework for Distributed MQTT Brokers*, *Politecnico di Milano Technical Report*. April.

[49] Mishra, B., & Kertesz, A. 2021. *Stress-Testing MQTT Brokers: A Comparative Analysis of Performance Measurements*, *Energies*, vol. 14, no. 18, pp. 5817. June.

[50] Kashyap, M. 2024. *Implementation and Analysis of EMQX Broker for MQTT Protocol in Industrial IoT*, *ScienceDirect*. April.

[51] Arafat, J., Tasmin, F., Poudel, S., Tareq, A. H. 2025. *Next-Generation Event-Driven Architectures: Performance, Scalability, and Intelligent Orchestration Across Messaging Frameworks*, *arXiv preprint*. October.

[52] Lima, K., Oyetoyan, T. D., Heldal, R., Hasselbring, W. 2025. *Evaluation of MQTT Bridge Architectures in a Cross-Organizational Context*, *arXiv preprint*. January.

[53] Cleland, G., Wu, D., Ullah, R., Varghese, B. 2022. *FedComm: Understanding Communication Protocols for Edge-based Federated Learning*, *arXiv preprint*. August.

APPENDICES

Appendix I: Mosquitto Load Tests

Mosquitto Load Test 1	Mosquitto Load Test 2
# Configuration Concurrent Clients: 10 Messages / Client: 100 # Results Published Messages: 100 (100%) Received Messages: 100 (100%) Completed: 10 (100%) Errors: 0 (0%) # Publishing Throughput Fastest: 52760 msg/sec Slowest: 7378 msg/sec Median: 36488 msg/sec < 11916 msg/sec 10% < 57298 msg/sec 10% < 52760 msg/sec 10% <48221 msg/sec 20% < 39145 msg/sec 20% < 34607 msg/sec 20% < 16454 msg/sec 10% # Receiving Througput Fastest: 248 msg/sec Slowest: 222 msg/sec Median: 237 msg/sec < 243 msg/sec 10% < 238 msg/sec 30% < 235 msg/sec 20% < 233 msg/sec 10% < 225 msg/sec 10% < 251 msg/sec 10% < 248 msg/sec 10% ["-broker", "tcp://192.168.35.24:1883", "-username", "testuser", "-password", "123456", "-num-clients", "10", "-num-messages", "10", "-rampup-delay", "1s", "-global-timeout", "60s", "-timeout", "5s"]	# Configuration Concurrent Clients: 100 Messages / Client: 10000 # Results Published Messages: 10000 (100%) Received Messages: 10000 (100%) Completed: 100 (100%) Errors: 0 (0%) # Publishing Throughput Fastest: 25736 msg/sec Slowest: 67 msg/sec Median: 83 msg/sec < 28303 msg/sec 1% < 5200 msg/sec 2% < 2633 msg/sec 97% # Receiving Througput Fastest: 182113 msg/sec Slowest: 85 msg/sec 18288 msg/sec 79% 182113 msg/sec 1% 72896 msg/sec 2% 54693 msg/sec 10% 36491 msg/sec 8% ["-broker", "tcp://192.168.35.24:1883", "-username", "testuser", "-password", "123456", "-num-clients", "100", "-num-messages", "100", "-rampup-delay", "200ms", "-global-timeout", "600s", "-timeout", "10s"]

Mosquitto Load Test 3	Mosquitto Load Test 4
# Configuration Concurrent Clients: 100 Messages / Client: 50000 # Results Published Messages: 50000 (100%) Received Messages: 50000 (100%) Completed: 100 (100%) Errors: 0 (0%) # Publishing Throughput Fastest: 466 msg/sec Slowest: 95 msg/sec Median: 152 msg/sec < 169 msg/sec 37% < 503 msg/sec 1% < 429 msg/sec 1% < 392 msg/sec 2% < 355 msg/sec 2% < 280 msg/sec 5% < 243 msg/sec 8% < 206 msg/sec 15% < 132 msg/sec 28% < 318 msg/sec 1% # Receiving Througput Fastest: 1683 msg/sec Slowest: 209 msg/sec Median: 297 msg/sec < 504 msg/sec 19% < 356 msg/sec 73% < 1830 msg/sec 1% < 1093 msg/sec 1% < 946 msg/sec 1% < 798 msg/sec 3% < 651 msg/sec 2% ["-broker", "tcp://192.168.35.24:1883", "-username", "testuser", "-password", "123456", "-num-clients", "100", "-num-messages", "500", "-rampup-delay", "200ms", "-global-timeout", "600s", "-timeout", "10s"]	# Configuration Concurrent Clients: 100 Messages / Client: 80000 # Results Published Messages: 80000 (100%) Received Messages: 80000 (100%) Completed: 100 (100%) Errors: 0 (0%) # Publishing Throughput Fastest: 313 msg/sec Slowest: 120 msg/sec Median: 158 msg/sec < 197 msg/sec 15% < 178 msg/sec 25% < 158 msg/sec 32% < 139 msg/sec 18% < 333 msg/sec 1% < 255 msg/sec 3% < 236 msg/sec 2% < 216 msg/sec 4% # Receiving Througput Fastest: 326 msg/sec Slowest: 170 msg/sec Median: 216 msg/sec < 342 msg/sec 1% < 326 msg/sec 1% < 311 msg/sec 4% < 295 msg/sec 1% < 248 msg/sec 11% < 233 msg/sec 14% < 201 msg/sec 25% < 186 msg/sec 4% < 280 msg/sec 3% < 264 msg/sec 11% < 217 msg/sec 25% ["-broker", "tcp://192.168.35.24:1883", "-username", "testuser", "-password", "123456", "-num-clients", "100", "-num-messages", "800", "-rampup-delay", "200ms", "-global-timeout", "600s", "-timeout", "10s"]

Appendix II: EMQX Load Tests

EMOX Load Test 1	EMOX Load Test 2
# Configuration Concurrent Clients: 10 Messages / Client: 100 # Results Published Messages: 100 (100%) Received Messages: 100 (100%) Completed: 10 (100%) Errors: 0 (0%) # Publishing Throughput Fastest: 35970 msg/sec Slowest: 1814 msg/sec Median: 8217 msg/sec < 39386 msg/sec 10% < 32554 msg/sec 10% < 15476 msg/sec 10% < 12061 msg/sec 20% < 8645 msg/sec 30% < 5229 msg/sec 20% # Receiving Througput Fastest: 44225 msg/sec Slowest: 1853 msg/sec Median: 4759 msg/sec < 48462 msg/sec 10% < 18802 msg/sec 20% < 10328 msg/sec 10% < 6090 msg/sec 60% ["-broker", "tcp://192.168.35.24:1884", "-num-clients", "10", "-num-messages", "10", "-rampup-delay", "200ms", "-global-timeout", "600s", "-timeout", "10s"]	# Configuration Concurrent Clients: 100 Messages / Client: 10000 # Results Published Messages: 10000 (100%) Received Messages: 10000 (100%) Completed: 100 (100%) Errors: 0 (0%) # Publishing Throughput Fastest: 43967 msg/sec Slowest: 343 msg/sec Median: 534 msg/sec < 43967 msg/sec 1% < 26518 msg/sec 1% < 22155 msg/sec 2% < 17793 msg/sec 3% < 9068 msg/sec 6% < 4705 msg/sec 86% < 48330 msg/sec 1% # Receiving Througput Fastest: 329 msg/sec Slowest: 71 msg/sec Median: 119 msg/sec < 123 msg/sec 19% < 97 msg/sec 32% < 329 msg/sec 2% < 252 msg/sec 4% < 200 msg/sec 4% < 175 msg/sec 10% < 149 msg/sec 21% < 355 msg/sec 1% < 278 msg/sec 2% < 226 msg/sec 5% ["-broker", "tcp://192.168.35.24:1884", "-num-clients", "100", "-num-messages", "100", "-rampup-delay", "200ms", "-global-timeout", "600s", "-timeout", "10s"]

Appendix III: RabbitMQ Load Tests

RabbitMQ Load Test 1	RabbitMQ Load Test 2
# Configuration Concurrent Clients: 10 Messages / Client: 100 # Results Published Messages: 100 (100%) Received Messages: 100 (100%) Completed: 10 (100%) Errors: 0 (0%) # Publishing Throughput Fastest: 37839 msg/sec Slowest: 1603 msg/sec Median: 9580 msg/sec < 8850 msg/sec 20% < 5227 msg/sec 20% < 41463 msg/sec 10% < 23345 msg/sec 20% < 16098 msg/sec 10% < 12474 msg/sec 20% # Receiving Througput Fastest: 62945 msg/sec Slowest: 4028 msg/sec Median: 8328 msg/sec < 68837 msg/sec 10% < 45270 msg/sec 10% < 15811 msg/sec 10% < 9919 msg/sec 70% ["-broker", "tcp://192.168.35.24:1885", "-username", "admin", "-password", "admin", "-num-clients", "10", "-num-messages", "10", "-rampup-delay", "200ms", "-global-timeout", "600s", "-timeout", "10s"]	# Configuration Concurrent Clients: 100 Messages / Client: 10000 # Results Published Messages: 10000 (100%) Received Messages: 10000 (100%) Completed: 100 (100%) Errors: 0 (0%) # Publishing Throughput Fastest: 64664 msg/sec Slowest: 1505 msg/sec Median: 23951 msg/sec < 64664 msg/sec 3% < 26769 msg/sec 12% < 14137 msg/sec 14% < 7821 msg/sec 7% Completed: 100 (100%) Errors: 0 (0%) < 20453 msg/sec 20% < 70980 msg/sec 1% < 58348 msg/sec 7% < 52032 msg/sec 7% < 45716 msg/sec 7% < 39401 msg/sec 7% < 33085 msg/sec 15% # Receiving Througput Fastest: 6541 msg/sec Slowest: 408 msg/sec Median: 778 msg/sec < 7155 msg/sec 1% < 4088 msg/sec 2% < 3475 msg/sec 1% < 1635 msg/sec 22% < 1021 msg/sec 74% ["-broker", "tcp://192.168.35.24:1885", "-username", "admin", "-password", "admin", "-num-clients", "100", "-num-messages", "100", "-rampup-delay", "200ms", "-global-timeout", "600s", "-timeout", "10s"]

RabbitMQ Load Test 3	RabbitMQ Load Test 4
# Configuration Concurrent Clients: 100 Messages / Client: 50000 # Results Published Messages: 50000 (100%) Received Messages: 35146 (70%) Completed: 67 (67%) Errors: 33 (33%) - ConnectFailed:0 (0%) - SubscribeFailed: 0 (0%) - TimeoutExceeded: 33 (100%) - Aborted: 0 (0%) # Publishing Throughput Fastest: 80838 msg/sec Slowest: 491 msg/sec Median: 7427 msg/sec < 88873 msg/sec 1% < 56734 msg/sec 1% < 48699 msg/sec 6% < 40664 msg/sec 9% < 32630 msg/sec 6% < 24595 msg/sec 16% < 16560 msg/sec 4% < 8525 msg/sec 55% # Receiving Througput Fastest: 648 msg/sec Slowest: 189 msg/sec Median: 294 msg/sec < 372 msg/sec 30% < 327 msg/sec 24% < 281 msg/sec 21% < 235 msg/sec 21% < 694 msg/sec 1% < 510 msg/sec 1% < 418 msg/sec 1% ["-broker", "tcp://192.168.35.24:1885", "-username", "admin", "-password", "admin", "-num-clients", "100", "-num-messages", "500", "-rampup-delay", "200ms", "-global-timeout", "600s", "-timeout", "10s"]	# Configuration Concurrent Clients: 100 Messages / Client: 80000 # Results Published Messages: 80000 (100%) Received Messages: 39200 (49%) Completed: 49 (49%) Errors: 51 (51%) - ConnectFailed:0 (0%) - SubscribeFailed: 0 (0%) - TimeoutExceeded: 51 (100%) - Aborted: 0 (0%) # Publishing Throughput Fastest: 60968 msg/sec Slowest: 347 msg/sec Median: 7219 msg/sec < 48844 msg/sec 2% < 42782 msg/sec 2% < 36720 msg/sec 4% < 30657 msg/sec 6% < 18533 msg/sec 2% < 12471 msg/sec 39% < 6409 msg/sec 43% < 67030 msg/sec 2% # Receiving Througput Fastest: 349 msg/sec Slowest: 119 msg/sec Median: 141 msg/sec < 372 msg/sec 2% < 257 msg/sec 8% < 234 msg/sec 8% < 188 msg/sec 12% < 165 msg/sec 18% < 142 msg/sec 51% ["-broker", "tcp://192.168.35.24:1885", "-username", "admin", "-password", "admin", "-num-clients", "100", "-num-messages", "800", "-rampup-delay", "200ms", "-global-timeout", "600s", "-timeout", "10s"]

RabbitMQ Load Test 5	RabbitMQ Load Test 6
# Configuration Concurrent Clients: 100 Messages / Client: 100000 # Results Published Messages: 100000 (100%) Received Messages: 48878 (49%) Completed: 47 (47%) Errors: 53 (53%) - ConnectFailed:0 (0%) - SubscribeFailed: 0 (0%) - TimeoutExceeded: 53 (100%) - Aborted: 0 (0%) # Publishing Throughput Fastest: 40287 msg/sec Slowest: 703 msg/sec Median: 1670 msg/sec < 8620 msg/sec 9% < 4662 msg/sec 68% < 44246 msg/sec 2% < 40287 msg/sec 4% < 36329 msg/sec 4% < 32371 msg/sec 2% < 16537 msg/sec 4% < 12579 msg/sec 6% # Receiving Througput Fastest: 233 msg/sec Slowest: 105 msg/sec Median: 119 msg/sec < 118 msg/sec 47% < 246 msg/sec 2% < 220 msg/sec 2% < 195 msg/sec 2% < 182 msg/sec 4% < 144 msg/sec 6% < 131 msg/sec 36% ["-broker", "tcp://192.168.35.24:1885", "-username", "admin", "-password", "admin", "-num-clients", "100", "-num-messages", "1000", "-rampup-delay", "200ms", "-global-timeout", "600s", "-timeout", "10s"]	# Configuration Concurrent Clients: 100 Messages / Client: 150000 # Results Published Messages: 150000 (100%) Received Messages: 52311 (35%) Completed: 1 (1%) Errors: 99 (99%) - ConnectFailed:0 (0%) - SubscribeFailed: 0 (0%) - TimeoutExceeded: 99 (100%) - Aborted: 0 (0%) # Publishing Throughput Fastest: 340 msg/sec Slowest: 340 msg/sec Median: 340 msg/sec < 340 msg/sec 100% # Receiving Througput Fastest: 153 msg/sec Slowest: 153 msg/sec Median: 153 msg/sec < 153 msg/sec 100% ["-broker", "tcp://192.168.35.24:1885", "-username", "admin", "-password", "admin", "-num-clients", "100", "-num-messages", "1500", "-rampup-delay", "200ms", "-global-timeout", "600s", "-timeout", "10s"]

RabbitMQ Load Test 7
100 worker started ["-broker", "tcp://192.168.35.24:1885", "-username", "admin", "-password", "admin", "-num-clients", "100", "-num-messages", "2000", "-rampup-delay", "200ms", "-global-timeout", "600s", "-timeout", "10s"]

Appendix IV: Mosquitto Throughput Tests

Mosquitto Throughput Test 1	Mosquitto Throughput Test 2
# Configuration Concurrent Clients: 10 Messages / Client: 100 # Results Published Messages: 100 (100%) Received Messages: 100 (100%) Completed: 10 (100%) Errors: 0 (0%) # Publishing Throughput Fastest: 52760 msg/sec Slowest: 7378 msg/sec Median: 36488 msg/sec < 11916 msg/sec 10% < 57298 msg/sec 10% < 52760 msg/sec 10% < 48221 msg/sec 20% < 39145 msg/sec 20% < 34607 msg/sec 20% < 16454 msg/sec 10% # Receiving Througput Fastest: 248 msg/sec Slowest: 222 msg/sec Median: 237 msg/sec < 243 msg/sec 10% < 238 msg/sec 30% < 235 msg/sec 20% < 233 msg/sec 10% < 225 msg/sec 10% < 251 msg/sec 10% < 248 msg/sec 10%	# Configuration Concurrent Clients: 100 Messages / Client: 200000 # Results Published Messages: 200000 (100%) Received Messages: 183962 (92%) Completed: 35 (35%) Errors: 65 (65%) - ConnectFailed:0 (0%) - SubscribeFailed: 0 (0%) - TimeoutExceeded: 65 (100%) - Aborted: 0 (0%) # Publishing Throughput Fastest: 263 msg/sec Slowest: 129 msg/sec Median: 152 msg/sec < 196 msg/sec 9% < 183 msg/sec 9% < 169 msg/sec 20% < 156 msg/sec 43% < 142 msg/sec 17% < 276 msg/sec 3% # Receiving Througput Fastest: 270 msg/sec Slowest: 200 msg/sec Median: 215 msg/sec < 278 msg/sec 3% < 270 msg/sec 3% < 256 msg/sec 3% < 249 msg/sec 3% < 207 msg/sec 26% < 242 msg/sec 3% < 235 msg/sec 6% < 228 msg/sec 11% < 221 msg/sec 20% < 214 msg/sec 23%

Appendix V: EMQX Throughput Tests

EMQX Throughput Test 1	EMQX Throughput Test 2
# Configuration Concurrent Clients: 10 Messages / Client: 100 # Results Published Messages: 100 (100%) Received Messages: 100 (100%) Completed: 10 (100%) Errors: 0 (0%) # Publishing Throughput Fastest: 35970 msg/sec Slowest: 1814 msg/sec Median: 8217 msg/sec < 39386 msg/sec 10% < 32554 msg/sec 10% < 15476 msg/sec 10% < 12061 msg/sec 20% < 8645 msg/sec 30% < 5229 msg/sec 20% # Receiving Througput Fastest: 44225 msg/sec Slowest: 1853 msg/sec Median: 4759 msg/sec < 48462 msg/sec 10% < 18802 msg/sec 20% < 10328 msg/sec 10% < 6090 msg/sec 60%	# Configuration Concurrent Clients: 100 Messages / Client: 200000 # Results Published Messages: 200000 (100%) Received Messages: 153828 (77%) Completed: 14 (14%) Errors: 86 (86%) - ConnectFailed:0 (0%) - SubscribeFailed: 0 (0%) - TimeoutExceeded: 86 (100%) - Aborted: 0 (0%) # Publishing Throughput Fastest: 218 msg/sec Slowest: 154 msg/sec Median: 175 msg/sec < 224 msg/sec 7% < 205 msg/sec 14% < 186 msg/sec 14% < 179 msg/sec 21% < 173 msg/sec 14% < 160 msg/sec 29% # Receiving Througput Fastest: 587 msg/sec Slowest: 207 msg/sec Median: 363 msg/sec < 359 msg/sec 21% < 321 msg/sec 14% < 245 msg/sec 14% < 625 msg/sec 7% < 511 msg/sec 14% < 473 msg/sec 7% < 435 msg/sec 14% < 397 msg/sec 7%

Appendix VI: RabbitMQ Throughput Tests

RabbitMQ Throughput Test 1	RabbitMQ Throughput Test 2
# Configuration Concurrent Clients: 10 Messages / Client: 100 # Results Published Messages: 100 (100%) Received Messages: 100 (100%) Completed: 10 (100%) Errors: 0 (0%) # Publishing Throughput Fastest: 37839 msg/sec Slowest: 1603 msg/sec Median: 9580 msg/sec < 8850 msg/sec 20% < 5227 msg/sec 20% < 41463 msg/sec 10% < 23345 msg/sec 20% < 16098 msg/sec 10% < 12474 msg/sec 20% # Receiving Througput Fastest: 62945 msg/sec Slowest: 4028 msg/sec Median: 8328 msg/sec < 68837 msg/sec 10% < 45270 msg/sec 10% < 15811 msg/sec 10% < 9919 msg/sec 70%	# Configuration Concurrent Clients: 100 Messages / Client: 150000 # Results Published Messages: 150000 (100%) Received Messages: 52311 (35%) Completed: 1 (1%) Errors: 99 (99%) - ConnectFailed:0 (0%) - SubscribeFailed: 0 (0%) - TimeoutExceeded: 99 (100%) - Aborted: 0 (0%) # Publishing Throughput Fastest: 340 msg/sec Slowest: 340 msg/sec Median: 340 msg/sec < 340 msg/sec 100% # Receiving Througput Fastest: 153 msg/sec Slowest: 153 msg/sec Median: 153 msg/sec < 153 msg/sec 100%