

People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
Mohamed El Bachir El Ibrahimi University of Bordj Bou Arréridj
Faculty of Mathematics and Computer Science
Department of Computer Science



THESIS

Submitted in partial fulfillment of the requirements
for the Master's Degree in Computer Science
Specialization: Information and Communication Technologies

TOPIC

Communication in IoT Environments
Decentralization and Resilience via Brokers

Submitted by:

Mr. Benzerroug Yasser

Publicly defended on: 12/06/2025

Before the jury composed of:

President: Dr. Hassina Bensfia

Examiner 1: Dr. Manel Lamri

Examiner 2: Dr. Mourad Nouioua

Supervisor: Dr. Lyazid Sabri

2024/2025

Dedication

*To the **Keepers of Origin** — whose silent watch, relentless resolve, and unseen sacrifices carved the first stones of this path. You taught without words and endured without praise. A father who became the light when all torches dimmed. A mother whose strength ran deeper than the silence she endured. And one sibling — more shield than sister — who stood when few did. In the shadows, you shaped a sovereign.*

*To the **Proven Few** — not bound by name, but by presence. The true kin, the ones who stayed when the storm bared its teeth. Your loyalty was never loud, but always there. You weren't many — but enough.*

*To the **Architects of Insight** — the ones who dared to see beyond the noise. From chalkboards in dust-lit classrooms to pages marked by late-night questioning, your belief kindled the ascent. To my supervisor your guidance cut through fog like steel. A steady hand, when the path blurred.*

*To the **Ones who left**, and the ones who remained unseen. Your absence taught resilience. Your doubt sharpened resolve. You gave this work its edge.*

*To the **One Who Stayed** — who rose through doubt, walked alone through the fight within.*

*This is no conclusion.
This is a mark. A flag planted in silence. A kingdom born not of inheritance — but intent.*

Tempus est dominus. Qui eum dominatur, imperat.

*Not an ending.
A beginning. Veiled in fire.*

Acknowledgments

I extend my deepest gratitude to my supervisor, **Dr. Lyazid Sabri**, whose trust in my potential often spoke louder than words. From the earliest stages, I felt that he believed I would carry this through — not because it was easy, but because it was mine to bear. His guidance was precise, his expectations clear, and his belief a quiet force that helped define the shape of this work.

I am also sincerely grateful to the president of the jury, **Dr. Hassina Bensfia**, as well as the examiners, **Dr. Manel Lamri** and **Dr. Mourad Nouioua**, for their role in evaluating this thesis. More than just jury members, they have each shaped my academic journey as professors whose teaching left a lasting impact. Their presence in this final chapter is deeply meaningful, and I thank them for the time, insight, and knowledge they generously shared — both in the classroom and through this final assessment.

This work was carried out independently and with resolve, encompassing two distinct yet complementary subjects. The challenge was doubled; so was the growth.

To those who offered passing remarks, silent encouragements, or small moments of clarity — whether they knew it or not — you are part of the architecture of this accomplishment.

And finally, to the quiet allies: the unremarked hours, the late iterations, the cold coffee, and the steady resolve that never wavered — this journey was forged in your silent presence.

Abstract

The rapid expansion of the Internet of Things (IoT) has led to an increasing demand for communication protocols that are efficient, scalable, and reliable—especially in environments with limited resources. Among these, MQTT (Message Queuing Telemetry Transport) has emerged as a lightweight protocol particularly well suited for constrained IoT systems. However, as devices become more distributed geographically, new challenges arise in ensuring communication that is both decentralized and resilient.

This thesis addresses these challenges by exploring two real-world scenarios where MQTT is used to ensure system robustness and flexibility. The first involves a critical healthcare environment, where we implemented a fault-tolerant architecture using a Master/Slave model over Apache ActiveMQ with shared persistent storage. This setup ensures automatic broker failover with no interruption in data transmission or alert delivery, which is vital in medical emergencies. The second scenario focuses on decentralization through broker bridging, using Mosquitto to interconnect remote sites—a retirement home and a hospital. In this case, critical alerts from the retirement home are selectively forwarded to the hospital’s dashboard, reducing traffic while maintaining responsiveness.

Our contribution lies in the concrete design and implementation of these two architectures: a highly available broker system for resilience, and a decentralized communication model for distributed monitoring. We also explore the integration of WebSockets to deliver real-time alerts to web-based dashboards, further enhancing usability. Through this work, we demonstrate how MQTT can support both robust failover mechanisms and scalable, autonomous communication across heterogeneous IoT environments.

Keywords: MQTT, Internet of Things (IoT), Broker Communication, Master-Slave Architecture, High Availability, Real-Time Critical Alerts, Decentralized Architecture, Sensor Networks, WebSockets, Resilient Transmission via MQTT.

Résumé

Le développement rapide de l'Internet des objets (IdO) a accentué la nécessité de protocoles de communication efficaces, évolutifs et fiables, notamment dans des environnements aux ressources limitées. Le protocole MQTT (Message Queuing Telemetry Transport) s'est imposé comme une solution légère et adaptée à ces contraintes. Toutefois, à mesure que les objets connectés se déploient sur des sites géographiquement dispersés, de nouveaux défis apparaissent, en particulier en matière de résilience et de décentralisation des communications.

Ce mémoire aborde ces enjeux à travers deux scénarios concrets mettant en œuvre des courtiers MQTT. Le premier concerne un environnement hospitalier critique, dans lequel nous avons conçu et déployé une architecture tolérante aux pannes fondée sur un modèle maître-esclave, via le broker Apache ActiveMQ et un mécanisme de stockage partagé. Cette solution permet un basculement automatique en cas de panne du courtier principal, assurant ainsi la continuité des transmissions et des alertes, sans intervention humaine. Le second scénario illustre un modèle de communication décentralisée entre deux sites distants — une maison de retraite et un hôpital — à l'aide d'un pont entre courtiers Mosquitto. Les alertes critiques générées localement sont ainsi relayées vers l'hôpital, tout en conservant l'autonomie du site émetteur.

Notre contribution réside dans la conception, l'implémentation et l'évaluation de ces deux architectures : l'une orientée haute disponibilité, l'autre orientée décentralisation. Nous avons également intégré le protocole WebSocket pour la diffusion en temps réel des alertes vers une interface web, offrant ainsi une solution complète de supervision. Ce travail démontre la capacité du protocole MQTT à soutenir des systèmes IoT à la fois robustes, distribués et adaptés aux exigences des environnements critiques.

Mots clés : MQTT, Internet des objets (IoT), Communication Broker, Architecture maître-esclave, Haute disponibilité, Alertes critiques en temps réel, Architecture décentralisée, Réseaux de capteurs, WebSockets, Transmission résiliente via MQTT.

الملخص

أدى التطور السريع لإنترنت الأشياء (IoT) إلى زيادة الحاجة إلى بروتوكولات اتصال تكون فعّالة وقابلة للتوسع وموثوقة، خصوصاً في البيئات ذات الموارد المحدودة. وقد برز بروتوكول MQTT كحل خفيف وفّعال لهذه التحديات. ومع توسع استخدام الأجهزة المتصلة عبر مواقع جغرافية متباعدة، ظهرت تحديات جديدة تتعلق بمرونة الاتصال ولا مركزيته. يتناول هذا البحث هذه التحديات من خلال سيناريوهين واقعيين تم فيهما استخدام وسطاء MQTT. يمثل السيناريو الأول في بيئة طبية حرجة، حيث قننا بتصميم وتنفيذ بنية موثوقة تعتمد على نموذج "رئيسي/تابع" باستخدام الوسيط Apache ActiveMQ وآلية تخزين مشترك. تسمح هذه البنية بانتقال تلقائي بين الوسطاء في حالة فشل الوسيط الرئيسي، مما يضمن استمرار إرسال البيانات والتنبيهات دون انقطاع أو تدخل بشري. أما السيناريو الثاني فيركز على الاتصال اللامركزي بين موقعين متباعدين — دار رعاية ومستشفى — باستخدام تقنية الربط بين وسطاء Mosquitto، حيث يتم تمرير التنبيهات الحيوية فقط من دار الرعاية إلى المستشفى مع الحفاظ على استقلالية النظام المحلي. تتمثل مساهمتنا في هذا العمل في تصميم وتنفيذ وتقييم هاتين المعماريتين: الأولى مخصصة للتوافر العالي، والثانية لتحقيق اللامركزية في الاتصال. كما قننا بدمج بروتوكول WebSocket من أجل عرض التنبيهات في الوقت الحقيقي على لوحة تحكم ويب، مما يوفر نظام مراقبة متكامل. يبرز هذا العمل قدرة بروتوكول MQTT على دعم أنظمة إنترنت الأشياء التي تتمتع بالمرونة، والموثوقية، والقدرة على التكيف مع البيئات الحرجة والموزعة.

الكلمات المفتاحية: بروتوكول MQTT ، إنترنت الأشياء ، (IoT) الاتصال عبر الوسطاء، بنية الوسيط الرئيسي/التابع، التوفر العالي، التنبيهات الحرجة في الوقت الحقيقي، البنية اللامركزية، شبكات المستشعرات، WebSockets ، النقل الموثوق عبر MQTT.

Contents

Dedication	ii
Acknowledgments	iii
Abstract	iv
1 Introduction	1
1.1 Context	1
1.2 Problem Statement	1
1.3 Objectives	2
1.4 Methodology and Results	2
1.5 Structure of the Thesis	3
2 State of the Art	4
2.1 Introduction	4
2.2 MQTT Protocol Fundamentals	4
2.3 Broker Architectures and Implementations	6
2.4 High Availability Approaches in MQTT Brokers	7
2.5 Inter-Broker Communication and Bridging	9
2.6 Comparison of MQTT Brokers	10
2.7 Existing Work and Gaps in Literature	11
2.8 Conclusion of the Chapter	12
3 Resilience and Inter-Broker Communication	13
3.1 Introduction	13
3.2 Broker-Level Fault Tolerance with ActiveMQ	13
3.2.1 Problem Modeling	13
3.2.2 Architecture and Design	16
3.2.3 Algorithmic Overview	17
3.2.4 Scenario Execution	20
3.2.5 Observations and Analysis	22

3.2.6	Discussion	24
3.3	Inter-Broker Communication with Mosquitto	24
3.3.1	Problem Modeling	24
3.3.2	Architecture and Design	27
3.3.3	Algorithmic Overview	30
3.3.4	Scenario Execution	34
3.3.5	Observations and Analysis	37
3.3.6	Discussion	39
3.4	Conclusion of the chapter	39
4	Implementation	40
4.1	Implementation Overview	40
4.2	Environment Setup	40
4.2.1	General Development Environment	40
4.2.2	Dependencies and Toolchain	41
4.3	Scenario 1 – Shared Storage High Availability	43
4.3.1	Installing and Running ActiveMQ	43
4.3.2	Broker Configuration for Shared Storage HA	48
4.3.3	Publisher and Subscriber Client Interfaces	59
4.3.4	Client Interaction Mechanism	65
4.3.5	Failover Execution and Results	67
4.3.6	High Availability Architecture – Conclusion	72
4.4	Scenario 2 – Bridged MQTT Communication	73
4.4.1	Installing and Running Mosquitto	73
4.4.2	Broker Configuration for Communication Bridge	75
4.4.3	Publisher and Subscriber Client Interfaces	78
4.4.4	Client Interaction Mechanism	81
4.4.5	Execution and Results	83
4.4.6	Bridged Communication – Conclusion	87
4.5	Implementation Summary	87
5	Evaluation and System Limitations	88
5.1	Evaluation Overview	88
5.2	Scenario 1: High Availability Evaluation	88
5.3	Scenario 2: Bridged Communication Evaluation	90
5.4	Conclusion and Comparative Summary	92
6	Conclusion and Perspectives	93

1 Introduction

1.1 Context

The Internet of Things (IoT) is transforming how modern infrastructures operate by enabling billions of devices to communicate and collaborate in real time. From smart cities to connected healthcare and precision agriculture, IoT systems allow physical environments to become intelligent and reactive.

These systems often rely on lightweight communication protocols to operate efficiently over constrained networks, where bandwidth, power, and stability are limited. Among these, MQTT has emerged as a practical and widely adopted solution. However, the increasing scale and distribution of IoT deployments introduce new architectural challenges. When devices are geographically dispersed or deployed in sensitive contexts such as healthcare, the need for reliability, fault tolerance, and decentralized communication becomes critical.

In this context, it becomes essential to rethink how MQTT brokers operate, how they ensure service continuity, and how they can be interconnected to support more resilient and scalable communication patterns. This calls for robust architectural models capable of ensuring communication continuity and supporting distributed deployments across dynamic and sometimes fragile environments.

1.2 Problem Statement

Despite its efficiency and simplicity, MQTT presents key architectural limitations when applied to real-world systems that demand robustness and flexibility. By design, MQTT relies on a central broker to route all messages. This makes the broker a single point of failure: if it crashes or becomes unreachable, the entire communication chain is interrupted.

In time-critical systems—such as patient monitoring, industrial control, or public safety—such a failure can lead to serious consequences. Moreover, MQTT does not natively support inter-broker communication, which limits its ability to scale across multiple regions or to support decentralized deployments.

The core challenge lies in architecting an MQTT-based systems capable of maintaining service continuity during broker failures, while enabling seamless communication across distributed brokers without relying on centralized coordination.

✦ Challenges

- One of the main challenges is decentralizing the IoT infrastructure to avoid saturation of central points, improve resilience, and maintain connectivity in the event of partial failure.
- Inter-broker communication (for example, between two villages equipped with sensors) and fault-tolerance mechanisms (such as master-slave brokers) are key approaches in this context.

1.3 Objectives

This Master’s thesis addresses the architectural limitations of MQTT systems through two practical use cases:

- **Ensuring fault tolerance through broker-level High Availability (HA):** In the event of a broker failure, there must be a mechanism for automatic failover that preserves session state and guarantees message continuity.
- **Enabling distributed communication through broker bridging:** In larger systems, brokers must be able to communicate across network boundaries to route messages between clients connected to different locations.

The aim is to design, implement, and evaluate MQTT-based systems that remain operational under broker failure and support cross-broker message exchange.

1.4 Methodology and Results

To address these limitations, two architectural approaches were designed and implemented. The first focuses on achieving fault tolerance through a high-availability broker system based on a Master/Slave model with shared storage. This setup allows seamless broker failover in case of failure, without affecting client connections or data integrity.

The second approach aims at enabling decentralized communication across physically separated sites. By establishing a communication bridge between two MQTT brokers, it becomes possible to forward only critical messages from one site to another—reducing overhead while preserving local autonomy.

These two architectures were implemented and evaluated through simulated real-life scenarios. Broker resilience was tested using Apache ActiveMQ configured for high availability, while inter-broker communication was achieved using Mosquitto’s bridging

functionality. MQTT clients were developed in Python, and real-time visualization was integrated via WebSocket using a web dashboard interface. The evaluation focused on behavior during broker failures, alert continuity, and scalability across distributed environments.

1.5 Structure of the Thesis

This thesis is structured as follows:

- **Chapter 2 – State of the Art:** Reviews MQTT protocol fundamentals and compares broker capabilities, with a focus on HA and bridging.
- **Chapter 3 – Resilience and Inter-Broker Communication:** Presents architecture and configuration for both scenarios.
- **Chapter 4 – Implementation:** Describes technical integration of brokers, clients, and the dashboard interface. Including a full evaluation for system behavior through fault injection and performance tests.
- **Chapter 5 – Conclusion:** Summarizes findings and proposes improvements for future resilient IoT deployments.

2 State of the Art

2.1 Introduction

This chapter presents a comprehensive review of the MQTT protocol, its architectural foundations, and its applicability to distributed IoT systems. Given the growing demand for real-time, fault-tolerant communication in critical environments, it is essential to understand the strengths and limitations of current MQTT broker implementations.

We begin by outlining the core principles and features of the MQTT protocol, including its publish/subscribe model, Quality of Service levels, and WebSocket integration. We then compare major broker implementations—such as Mosquitto, ActiveMQ, EMQX, and HiveMQ—highlighting their capabilities in terms of high availability, scalability, and bridging support.

Subsequently, we examine existing high availability models and inter-broker communication techniques, identifying key design trade-offs. A comparative table is provided to summarize broker capabilities. Finally, we discuss relevant academic and industrial works to highlight gaps in current literature and motivate the practical scenarios addressed in this thesis.

2.2 MQTT Protocol Fundamentals

The Message Queuing Telemetry Transport (MQTT) is a lightweight, open protocol designed for efficient communication in constrained or unreliable network environments. It follows a publish/subscribe messaging model, which decouples data producers (publishers) from consumers (subscribers) through an intermediary known as the broker. This architecture reduces network overhead and enables asynchronous, many-to-many communication, making it ideal for Internet of Things (IoT) applications [1]. In MQTT, all messages are published to *topics*, which form a hierarchical namespace (e.g., `building1/roomA/temperature`). Subscribers express interest in specific topics or wildcard patterns, and the broker routes messages accordingly. This model promotes scalability and flexibility in distributed systems.

1. Levels of Message Delivery Reliability in MQTT

A defining feature of MQTT is its support for multiple **Quality of Service (QoS)** levels:

- **QoS 0** – At most once: messages are delivered on a best-effort basis without acknowledgment.
- **QoS 1** – At least once: messages are guaranteed to arrive but may be duplicated.
- **QoS 2** – Exactly once: messages are delivered once and only once, using a four-step handshake.

2. Additional MQTT Capabilities

Further MQTT features of interest include:

- **Retained messages:** the broker stores the last retained message per topic and delivers it immediately to new subscribers.
- **Last Will and Testament (LWT):** clients can specify a last message to be published if they disconnect unexpectedly.
- **Session persistence:** clients can maintain state between connections if the broker supports persistent sessions.

MQTT has evolved since its inception. Version 3.1.1, which is widely implemented and used in this thesis, is standardized by OASIS and offers a minimal core feature set [1]. Version 5.0 introduces extended metadata, shared subscriptions, and improved error reporting, but is not yet universally supported across all brokers [2].

While MQTT was originally designed to run over TCP/IP, modern implementations also support alternative transport protocols. One widely adopted option is **MQTT over WebSockets**, which enables brokers to communicate with clients over HTTP-compliant channels [3]. This is especially useful in environments with strict firewall rules or where browser-based clients (e.g., dashboards) need to interact with MQTT services. WebSocket transport maintains the publish/subscribe semantics of MQTT while using standard web ports such as 80 and 443.

The combination of lightweight messaging, flexible topic structures, and adaptable transport makes MQTT particularly well-suited for distributed IoT systems. However, its simplicity also introduces architectural limitations—particularly in broker-level reliability and cross-broker communication—which this thesis seeks to address.

2.3 Broker Architectures and Implementations

The MQTT protocol is broker-based, meaning that the broker plays a central role in message routing, session management, and delivery guarantees. While MQTT defines the communication behavior between clients and the broker, it leaves broker implementation details open to interpretation. As a result, multiple brokers exist—each with unique architectures, features, and performance profiles.

- **Apache ActiveMQ Classic** [4] is a mature, multi-protocol message broker developed by the Apache Software Foundation. Although originally built around the Java Message Service (JMS) API and designed for enterprise messaging, it also supports MQTT through a dedicated transport connector. ActiveMQ offers persistent message storage (via KahaDB), broker-level failover mechanisms, and pluggable configurations via XML files. Its architecture is modular, and it supports features such as advisory messaging, topic hierarchies, and various transport protocols (OpenWire, STOMP, MQTT, and WebSockets).
- **Eclipse Mosquitto** [5] is a lightweight and efficient MQTT broker designed specifically for resource-constrained environments. Written in C and built to handle thousands of concurrent connections with minimal resource usage, Mosquitto is ideal for edge computing or embedded use cases. It supports static bridging, TLS encryption, WebSocket transport, and simple authentication. However, it lacks native high availability or clustering features.

Other brokers have emerged to address advanced scalability and fault tolerance needs:

- **EMQX** (Erlang MQTT Broker) [6] is a scalable, cloud-native broker written in Erlang/OTP. It supports millions of concurrent connections, distributed clustering, built-in authentication and authorization, and MQTT v5 features.
- **HiveMQ** [7] is a commercial-grade MQTT broker with a strong focus on scalability, extensibility, and cloud integration. It supports session persistence across clusters, shared subscriptions, WebSockets, and secure transport.
- **VerneMQ** [8] is another scalable MQTT broker written in Erlang, designed for distributed deployments with horizontal scaling, clustering, and plugin extensibility.

This thesis focuses on ActiveMQ and Mosquitto as representative brokers: one offering high availability via shared storage, and the other enabling inter-broker bridging. These two choices strike a practical balance between architectural complexity and ease of implementation.

2.4 High Availability Approaches in MQTT Brokers

In MQTT systems, the broker serves as the core message router between publishers and subscribers. Its availability is essential, as failure can disrupt communication, break client sessions, and cause data loss. To prevent such risks, High Availability (HA) must be considered in the system design. HA strategies, including clustering and failover, help maintain service continuity and ensure message flow during faults. These approaches allow the system to recover quickly, minimizing downtime. In time-sensitive applications, such as smart homes or healthcare monitoring, even brief outages can have serious consequences. Ensuring broker resilience is therefore not only a technical goal but also a user-centered priority. By adopting HA mechanisms, MQTT infrastructures can better support reliability, trust, and system integrity under real-world conditions.

Resilience Requirements for IoT Messaging Systems

Before exploring specific high availability (HA) mechanisms in MQTT brokers, it is essential to understand the resilience expectations that arise in IoT deployments. Resilience in messaging refers to a system's ability to maintain communication flow despite failures, network partitions, or software crashes. As brokers serve as the central communication hubs in MQTT systems, any disruption to their availability can result in data loss, sensor silence, or missed control signals.

The demand for resilient messaging infrastructures is especially critical in mission-driven environments such as healthcare, industrial automation, and energy management. Al-Masri et al. [9] emphasized the significance of robust protocol design in ensuring dependable IoT operation under real-time constraints. Similarly, Shapsough et al. [10] proposed a scalable MQTT-based control system for photovoltaic energy monitoring that explicitly relies on uninterrupted broker service to avoid energy imbalances.

Resilience also extends to the ability to recover from failure states and resume communication with minimal intervention. Mechanisms such as persistent sessions, last-will messages, and replay buffers help achieve this, but without a reliable broker or HA strategy, these features remain vulnerable. Consequently, designing fault-tolerant MQTT systems requires architectural planning at the broker level, not just the protocol level.

The following subsections explore the concrete HA models employed in modern MQTT-capable brokers, evaluating their reliability and implementation complexity.

HA models are employed in MQTT-capable brokers

1. Shared Storage HA

This model allows multiple broker instances to access a shared persistent store. Only one broker is active at a time. ActiveMQ uses this approach via KahaDB, where a slave monitors the storage used by the master [11]. Upon failure, the slave takes control and resumes operation with minimal interruption.

- **Pros:** simple to set up, no replication logic required.
- **Cons:** shared storage becomes a potential single point of failure.

2. Shared-Nothing Clustering

In shared-nothing clustering, each broker maintains its own state and replicates relevant data using consensus protocols. This approach is used by EMQX and HiveMQ to support multi-node clusters with fault tolerance [12].

- **Pros:** true horizontal scaling and fault isolation.
- **Cons:** high configuration complexity and risk of inconsistent replication if not tuned correctly.

3. Database Locking

This strategy uses an external database to store state and coordinate broker roles. Brokers compete for a lock (e.g., JDBC semaphore). Failover occurs when the lock is transferred [13].

- **Pros:** strong durability, no shared filesystem needed.
- **Cons:** slower failover, external DB required.

Trade-offs

Each approach has strengths and weaknesses:

- **Shared storage:** simple but vulnerable.
- **Shared nothing:** resilient but complex.
- **DB-locking:** durable but slower.

MQTT doesn't specify HA behavior, implementations vary significantly. This thesis uses ActiveMQ's shared store HA setup as a testable, real-world method to ensure broker-level availability.

2.5 Inter-Broker Communication and Bridging

MQTT bridging allows multiple brokers to interconnect and exchange messages across different systems. This is vital in systems where brokers are deployed in separate locations (e.g., hospitals, edge devices, cloud nodes).

1. Static vs. Dynamic Bridging

Static bridging, as supported by Mosquitto, is manually defined in the broker config file [14]. Topics can be filtered or forwarded in specific directions. **Dynamic bridging**, less common in lightweight brokers, involves auto-discovery and dynamic link establishment (e.g., in EMQX).

2. Loop Prevention and Topic Mapping

To prevent message loops and distinguish bridged traffic, brokers use:

1. Topic remapping (e.g., prefixing 'remote/').
2. Direction flags (in/out/local).
3. Bridging rules to avoid echoing.

Misconfiguration can lead to message duplication, loops or CPU-intensive forwarding, especially in mesh topologies.

3. When to Use Bridging ?

Bridging is appropriate when:

1. Clients must connect to local brokers (e.g., for latency or firewall rules).
2. HA is not required, but regional routing is needed.
3. Interoperability between MQTT systems is necessary.

It is not a substitute for HA. Bridging provides distribution, not failover.

2.6 Comparison of MQTT Brokers

To better understand the practical trade-offs between different MQTT broker implementations, it is useful to compare their core capabilities side by side. This includes support for MQTT versions, high availability mechanisms, clustering options, and extensibility. Such a comparison not only highlights the diversity in design philosophies—from minimal lightweight brokers like Mosquitto to scalable, cloud-native solutions like EMQX and HiveMQ—but also reflects the architectural decisions that influence reliability and integration in real-world deployments.

Feature	ActiveMQ	Mosquitto	EMQX	HiveMQ	VerneMQ
MQTT Version	3.1.1	3.1.1 / 5.0	3.1.1 / 5.0	3.1.1 / 5.0	3.1.1 / 5.0
HA Support	Shared Storage	None	Clustering	Clustering	Clustering
Clustering	No	No	Yes	Yes	Yes
Bridging	No	Static Only	Dynamic	Dynamic	Static/Dynamic
WebSocket Support	Yes	Yes	Yes	Yes	Yes
Persistence	KahaDB, JDBC	File-based	Mnesia/SQL	Pluggable DB	LevelDB
Extensibility	Moderate	Minimal	High	High	High
License	Apache 2.0	EPL/EDL	Apache 2.0	Commercial (core open)	Apache 2.0

Table 2.1: Comparison of MQTT Brokers Based on Key Features

The table above [2.1](#) shows that MQTT brokers vary significantly in terms of architecture, scalability, and built-in fault tolerance. While all platforms listed are open source or offer open-source editions, this often comes with trade-offs. Feature sets such as clustering, high availability, and plugin extensibility may be limited or require additional configuration in non-commercial versions. These constraints are important to consider when designing systems that demand both resilience and flexibility.

For this thesis, ActiveMQ and Mosquitto were chosen as representative brokers due to their contrasting design goals. One emphasizes lightweight deployment with bridging support, while the other offers persistent failover via shared storage. Together, they enable a practical evaluation of MQTT reliability across different operational contexts.

2.7 Existing Work and Gaps in Literature

While many studies have explored MQTT protocol behavior and optimization [15], relatively few experimental works combine broker-level High Availability (HA) with message bridging in practical, controlled IoT testbeds.

Prior research such as [10] proposes MQTT-based architectures for large-scale solar energy monitoring and demonstrates the scalability of remote control systems. Meanwhile, [16] examines MQTT's use in industrial and retail IoT environments, focusing on low-latency performance but not on cross-broker fault tolerance. Surveys like [17] give broad overviews of distributed message broker architectures but rarely present concrete, integrated solutions combining failover resilience and decentralized bridging.

Most available resources on MQTT HA or bridging come from vendor documentation or informal tutorials, rather than peer-reviewed case studies. Moreover, there is limited academic work analyzing failover latency, session persistence, or cross-broker message consistency.

Research Trends in MQTT-Based Reliability and Bridging

Recent research has increasingly focused on enhancing the reliability of MQTT-based systems by exploring broker-level robustness, message delivery guarantees, and cross-broker synchronization. The convergence of high availability (HA) and inter-broker bridging is a growing area of interest, particularly as IoT infrastructures expand into decentralized and mission-critical contexts.

John and Liu [17] surveyed various message queue systems and highlighted the lack of native HA in lightweight MQTT brokers, noting that many rely on external workarounds or vendor-specific clustering solutions. Similarly, Konieczek et al. [18] performed an empirical comparison of MQTT brokers under edge computing scenarios, revealing notable disparities in failover latency, message persistence, and cluster behavior.

Moreover, studies such as Bandyopadhyay et al. [19] have emphasized the importance of middleware in managing distributed broker communication. Bridging is often treated as a distinct concern from HA; however, combining both capabilities presents a challenge for systems requiring both reliability and scalability. Most open-source brokers still treat these concerns separately, requiring custom configurations or third-party tools to achieve integrated behavior.

The following section illustrates how MQTT operates within mission-critical domains, reinforcing the importance of robust broker infrastructures.

i MQTT in Mission-Critical IoT Applications

MQTT has been widely adopted in domains where reliability, low latency, and distributed resilience are essential. Its lightweight nature and asynchronous publish/subscribe model make it suitable for constrained devices, but in mission-critical systems, these features must be complemented with robust fault-tolerant design.

- **Healthcare monitoring:** In hospital environments, wearable sensors transmit vital signs such as temperature, heart rate, and blood oxygen levels [20]. Any delay or loss in message delivery can compromise patient safety. Therefore, MQTT systems must be designed with high availability and immediate alert propagation in mind.
- **Smart energy systems:** MQTT has been used to monitor photovoltaic systems, smart grids, and microgeneration units [21]. Real-time fault detection and power balancing depend on uninterrupted message routing, even across geographically distributed brokers.
- **Industrial automation:** Manufacturing and industrial environments rely on MQTT for machine-to-machine (M2M) communications [22]. Downtime or broker failure can halt production lines or prevent critical control signals from being received.
- **Disaster response and edge computing:** In decentralized emergency systems and remote deployments, MQTT provides a low-bandwidth solution for coordinating devices [23]. Here, bridging between local and cloud brokers ensures continuity when backhaul links are unstable or unavailable.

These scenarios highlight MQTT's versatility but also expose the urgent need for resilient broker infrastructures.

2.8 Conclusion of the Chapter

Through this review of MQTT fundamentals, broker architectures, and fault-tolerant design models, we identified key limitations in existing implementations—specifically the absence of standardized high availability and native support for broker federation. While solutions like ActiveMQ and Mosquitto offer partial features, no integrated architecture combines both HA and distributed bridging. This gap justifies the dual-scenario approach undertaken in this thesis.

3 Resilience and Inter-Broker Communication

3.1 Introduction

In critical connected environments, such as smart medical infrastructures, the reliability of information exchange must be guided by rigorous management and control. Even the slightest interruption in data flow can compromise essential automated actions or delay vital alerts. This chapter addresses two complementary areas for strengthening the resilience of distributed architectures: ensuring continuity of service in the event of a central node failure and extending communication beyond the physical boundaries of a local system. Together, these mechanisms enable the maintenance of a constant flow of information, both within and outside the main structures, while ensuring seamless supervision of remote devices.

3.2 Broker-Level Fault Tolerance with ActiveMQ

3.2.1 Problem Modeling

In modern healthcare environments—especially hospitals equipped with smart monitoring infrastructure—the timely and uninterrupted flow of patient data is not a luxury; it is a necessity. From wearable sensors tracking vital signs such as body temperature and SpO₂ levels, to room-based monitoring systems that issue visual and audio alerts, these interconnected components rely heavily on continuous and reliable data transmission.

At the center of this communication web lies the MQTT broker, the lightweight but critical message-routing component. When it fails, the results are immediate and potentially life-threatening: sensor readings stop flowing, alerts are delayed, and automated responses—like activating a flashing red lamp outside a patient’s door—might not be executed. In such high-stakes environments, broker reliability is a foundational requirement.

To ensure that the system remains operational even in the face of software or hardware failures, this chapter focuses on implementing a broker-level **High Availability (HA)** mechanism using Apache ActiveMQ and its shared storage failover model. In this configuration, two brokers share access to a persistent message store (KahaDB) [24], but

only one is active at any time. If the active broker fails, the passive one takes over without human intervention. MQTT clients automatically reconnect and continue transmitting and receiving data.

🔑 Use Case Diagram

The following use case diagram outlines the key components involved in the MQTT-based monitoring architecture. It includes the wearable sensor (publisher), the hospital dashboard (subscriber), and the optional doctor for fingerprint-based alert clearance. The diagram highlights core functionalities such as publishing vitals, alert management, and broker failover recovery.

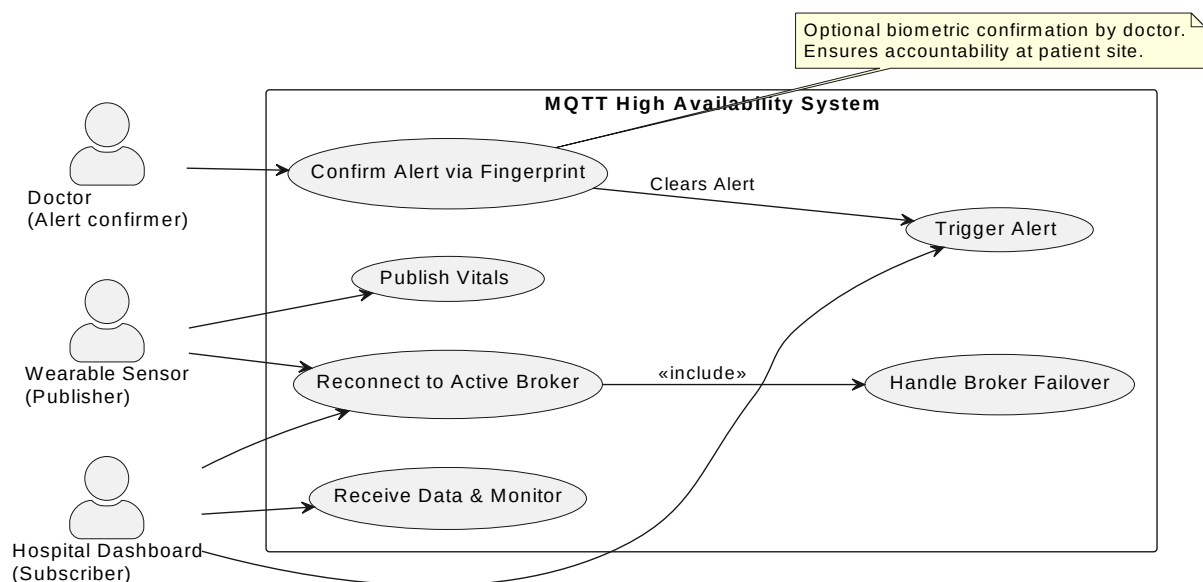


Figure 3.1: Use case diagram for the MQTT HA hospital monitoring system

A patient's temperature suddenly spikes. The wearable sensor detects the anomaly and publishes the data; the broker forwards it to the dashboard, which triggers an alert. A doctor rushes to the room, scans their fingerprint, and silences the alert—while in the background, the broker has already failed over to its passive instance without disrupting the data flow. This compact scenario illustrates how the system handles critical events with resilience and accountability.

⊕ Introducing Physical Confirmation: Fingerprint-Based Alert Clearance

Beyond ensuring message flow, modern patient monitoring systems must also provide accountability—verifiable proof that medical staff responded to an alert. To address this, our design introduces a fingerprint scanner as a means of alert confirmation.

Here’s the scenario: when a wearable sensor detects a dangerous change in a patient’s condition (e.g., sudden fever), it triggers an alert in the form of a flashing red light and a loudspeaker at the patient’s room. The only way to stop the alert is for a qualified staff member—typically a doctor or nurse—to physically go to the room and scan their fingerprint. This interaction serves multiple critical purposes:

- It guarantees that the alert is cleared only by authorized personnel.
- It logs the identity and response time of the responder for accountability.
- It ensures that the alert is not silenced remotely, which could be dangerous or misleading.

This physical confirmation mechanism adds a layer of trust to the alert-handling workflow, while the high availability of the MQTT system ensures that all related messages—sensor readings, alert triggers, and fingerprint events—are reliably transmitted even under failure conditions.

Although this fingerprint alert-clearance logic is not the core subject of this implementation chapter, it is part of the broader system design. If time permits, it may be demonstrated through screenshots or interface flows in the implementation chapter.

🌀 Contextual Continuity: Beyond Hospital Walls

The high-availability design presented here also has relevance beyond in-hospital settings. For example, patients recovering in remote care centers or at home can wear the same IoT devices. Their data can be routed to hospital dashboards through secure inter-broker communication, which is addressed in Chapter 4. In all such cases, continuous, fail-safe communication remains the backbone of patient monitoring.

3.2.2 Architecture and Design

This scenario implements a fault-tolerant MQTT messaging system tailored for a hospital's internal infrastructure. The system ensures that patient vitals are monitored in real time—even in the event of broker failure—and that alerts tied to these vitals remain responsive and recoverable.

1. System Components

- **MQTT Publisher:** A simulated wearable sensors continuously publishes patient temperature and SpO₂ data.
- **MQTT Subscriber:** A dashboard application subscribes to relevant topics and visualizes incoming sensor data, potentially triggering red light and speaker alerts.
- **ActiveMQ Brokers:** Two brokers operate in a master-slave configuration, sharing access to a KahaDB-based message store.
- **Shared KahaDB Storage:** A network-mounted or local file system used to persist messages and session data.

Under normal conditions, the master broker processes all messages while the slave remains on standby. If the master fails, the passive broker automatically acquires the storage lock and becomes the active broker. Clients, unaware of which broker is active, reconnect automatically and resume communication.

2. Transport Clarification

While the brokers support MQTT over TCP and WebSockets, the project architecture uses **TCP transport for broker communication**. Real-time interactions on the client-side dashboard—such as displaying incoming alerts, updating sensor data in real time, and handling fingerprint scan responses—are implemented using Flask and Flask-SocketIO [25, 26]. These components communicate via WebSockets at the application layer, independently of the broker's native transport options.

3. Quality of Service Considerations

All messages use **QoS 1** to ensure that they are delivered at least once. This guarantees delivery even during failover, though it may lead to duplicate messages. Application-level deduplication is recommended.

3.2.3 Algorithmic Overview

The reliability of this design stems from a simple failover algorithm based on file-lock acquisition over the shared KahaDB directory.

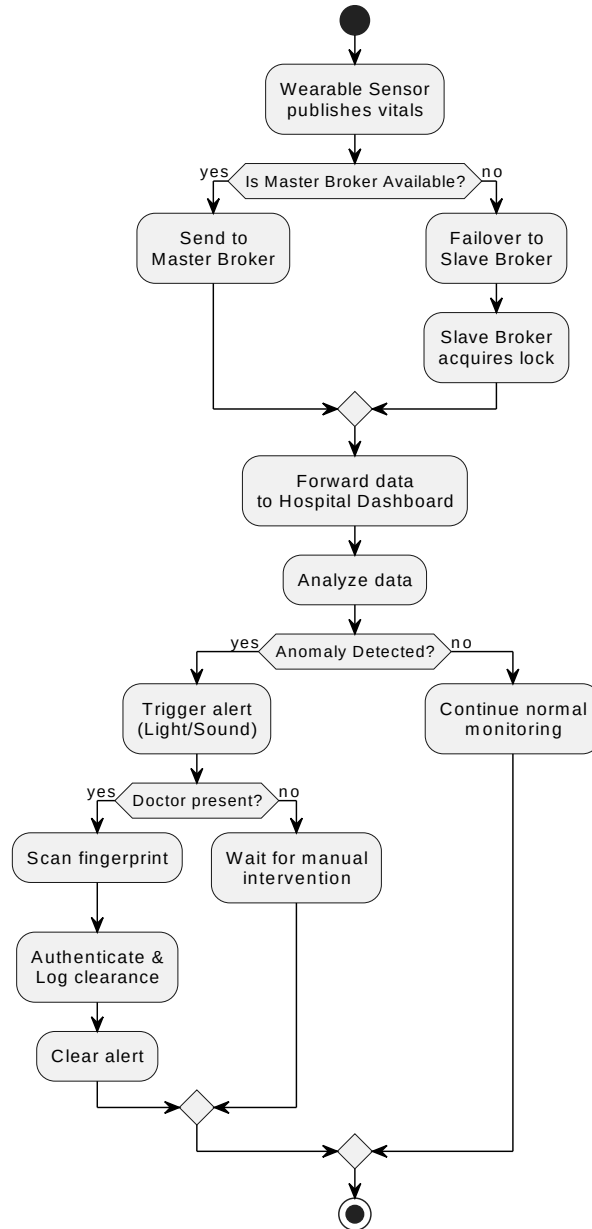


Figure 3.2: Flowchart of broker coordination, client reconnection, and alert lifecycle

The flowchart above provides a simplified yet complete view of the system's control logic. It shows how client connections, failover, message handling and alert verification are managed under both normal and fault conditions.

1. Broker Role Logic

Here's what happens during broker operation:

```
1 On broker startup:
2     If shared storage lock is available:
3         Acquire lock
4         Become ACTIVE broker
5         Accept client connections
6     Else:
7         Enter PASSIVE state
8         Monitor lock periodically
9
10 While running:
11     If lock becomes available:
12         Attempt to acquire lock
13         If successful:
14             Transition to ACTIVE
15             Start handling client sessions
```

Listing 3.1: Illustrating the behavior of an ActiveMQ broker during startup and runtime under a shared KahaDB configuration. The broker attempts to acquire a file-based lock to become active; otherwise, it enters standby mode and monitors the lock.

This logic avoids the need for complex consensus protocols or service discovery. The two brokers coordinate exclusively through the file lock.

2. Discussion: Lock-Based Coordination vs. Consensus Protocols

Lightweight Coordination: Lock-based coordination offers a simple alternative to consensus systems like ZooKeeper or Raft. It relies solely on acquiring a file lock in the shared KahaDB directory, avoiding leader election or quorum handling. This minimizes complexity and suits environments with limited resources or strict latency constraints.

Storage Dependency: The model's reliability depends entirely on the shared storage. If the filesystem becomes slow or inaccessible, brokers may both remain in standby or fail to transition correctly.

Predictable Failover: Despite this dependency, the architecture ensures deterministic failover: only one broker becomes active, message persistence is maintained, and clients remain unaffected. This makes the model highly predictable and ideal for healthcare systems requiring reliability and traceability.

3. Client Reconnection and Message Flow

Clients (publisher and subscriber) use automatic reconnection via the Paho MQTT library. Since the new master uses the same network address, no reconfiguration is needed. Any messages sent during the brief failover window are resent, ensuring data integrity.

🔑 Sequence diagram

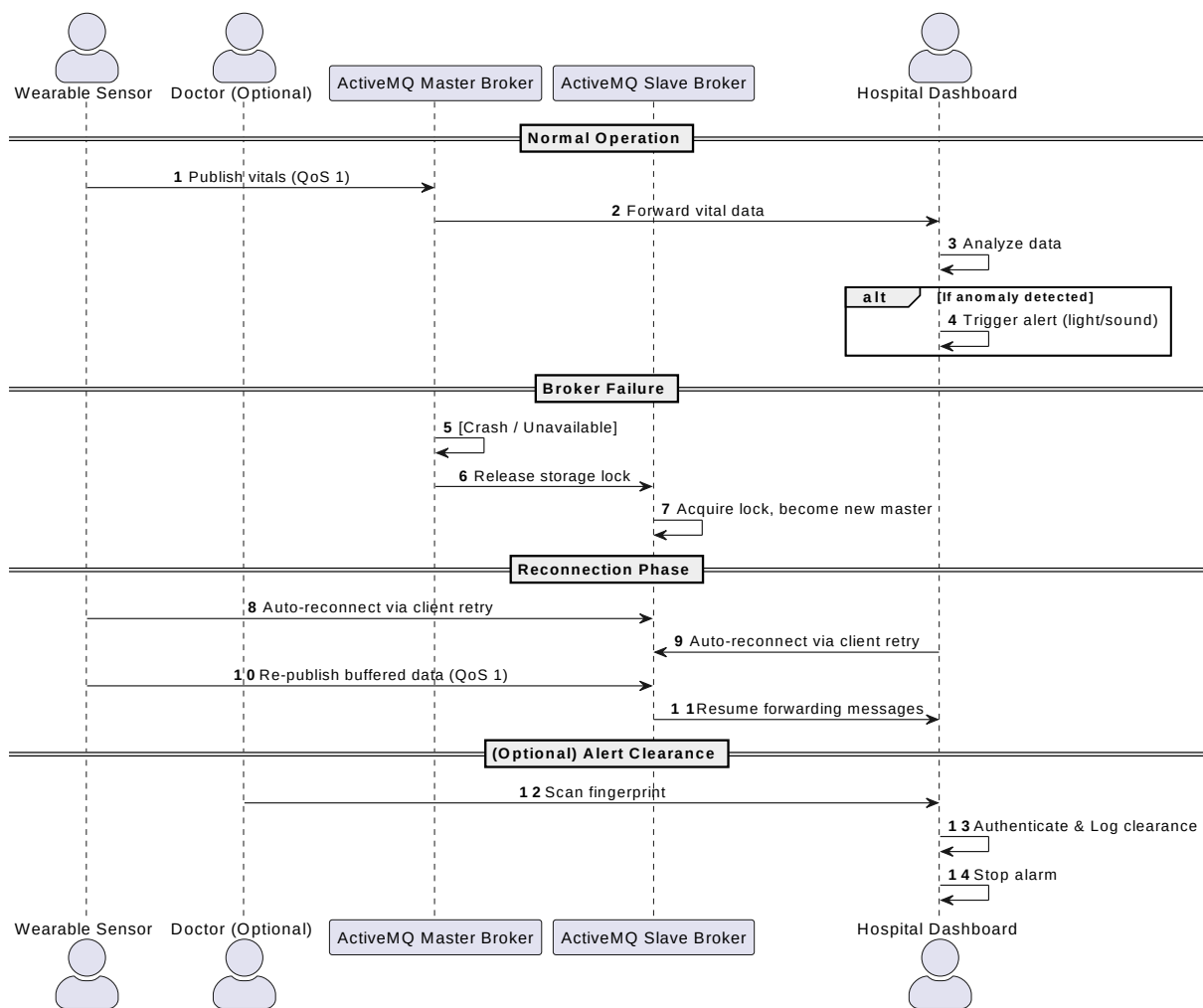


Figure 3.3: Sequence diagram showing normal operation, failover, and alert resolution

The sequence diagram above illustrates the complete flow of interactions in the system—from initial data publication and real-time alerting to broker failover handling and alert clearance via fingerprint. It reflects how failover is transparent to clients due to automatic reconnection and QoS guarantees.

3.2.4 Scenario Execution

This section presents the actual testing sequence conducted using:

- Two ActiveMQ brokers (master/slave with shared KahaDB)
- A Python MQTT publisher (simulated wearable sensor)
- A Python MQTT subscriber (nurse station dashboard)
- Flask dashboard with alert and fingerprint logic (optional)

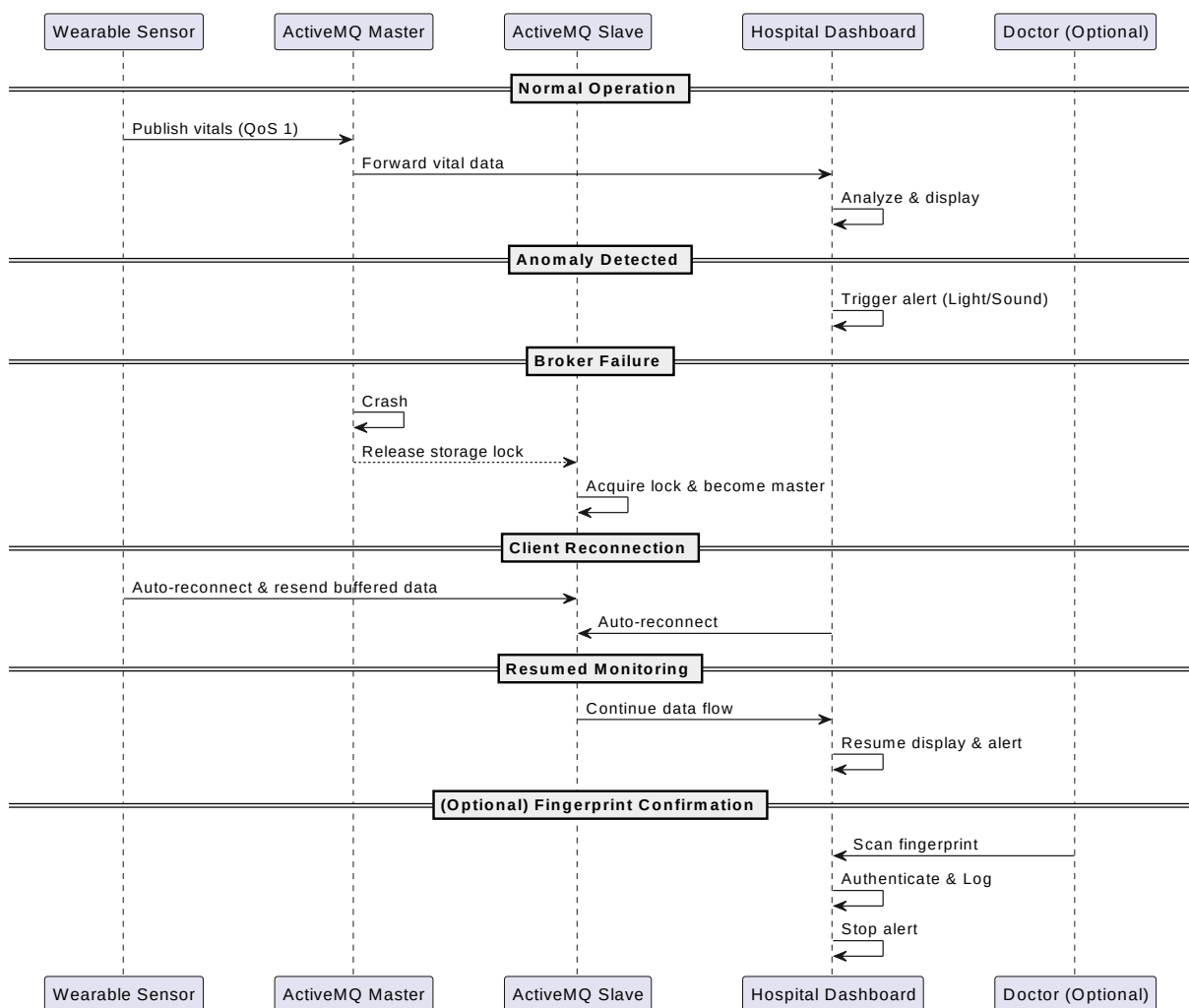


Figure 3.4: Execution timeline showing broker failure, failover, and client recovery

The execution timeline diagram visualizes the real-time sequence of events during the broker failover process. It confirms that message delivery continues within an acceptable delay window and highlights the system's resilience under stress.

❏ Execution Steps:

1. Master broker starts and locks KahaDB storage.
2. Slave broker starts in standby mode.
3. Publisher and subscriber connect to the broker and begin normal operation.

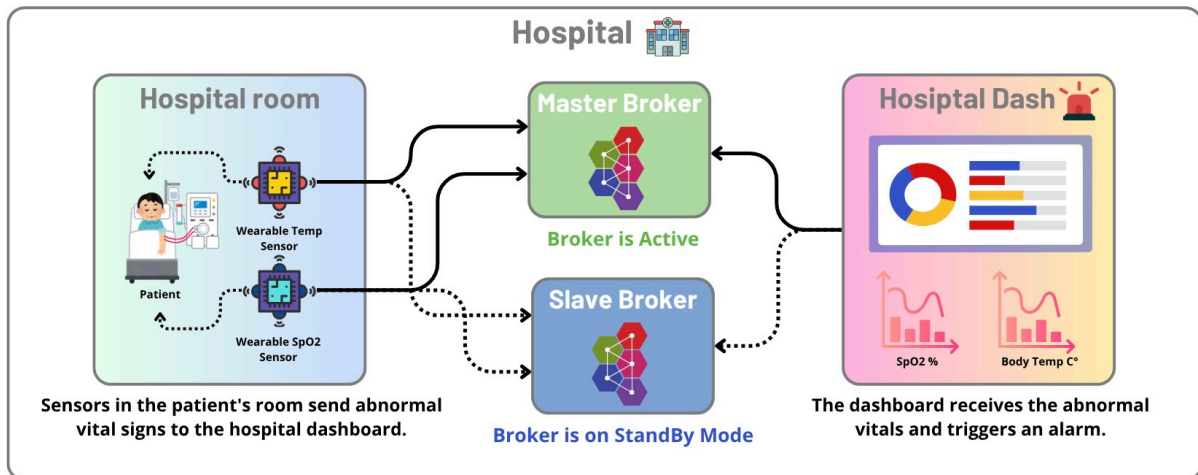


Figure 3.5: Normal operation – Master broker is active and processes abnormal vitals sent by devices. The slave remains on standby.

4. The master broker is terminated to simulate failure.
5. Slave broker detects the release of the lock, takes over.
6. Clients automatically reconnect and resume message flow.

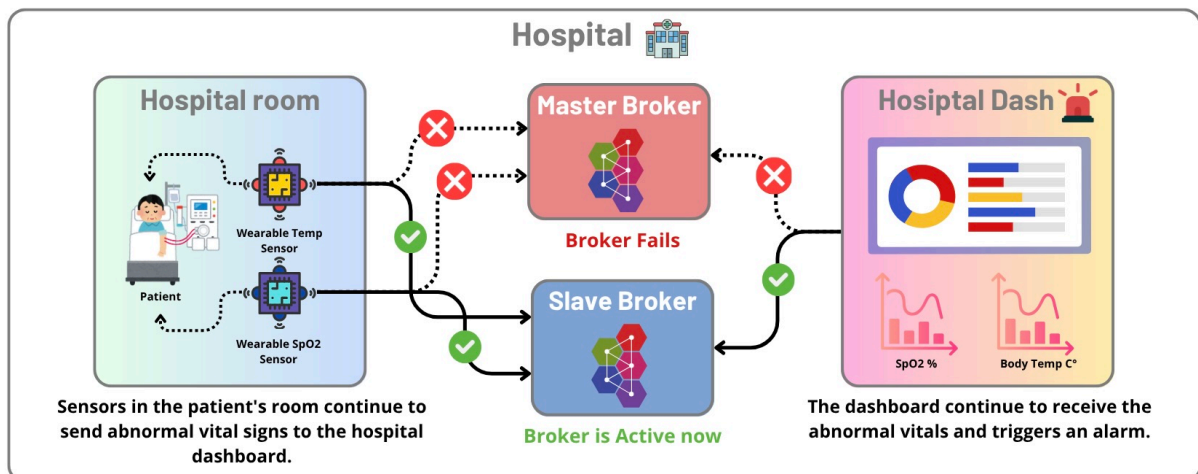


Figure 3.6: Failover – The slave broker takes over after master failure, ensuring continuous message delivery to the dashboard.

7. A red-light alert is triggered by abnormal vitals.

⚙️ Locking mechanism

The following figure shows the locking mechanism over high availability architecture :

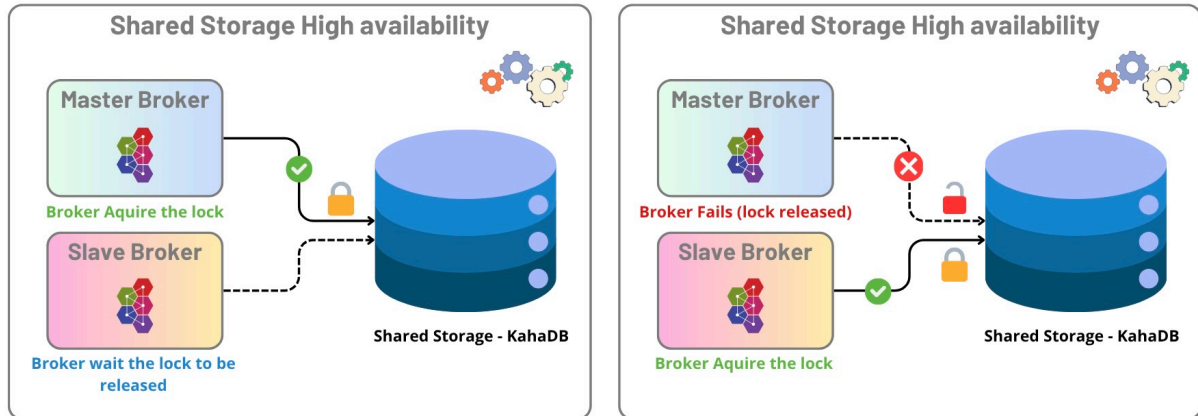


Figure 3.7: Locking mechanism – The master broker holds the lock during normal operation. Upon failure, the lock is released and acquired by the slave.

🔑 Conceptual Step – Fingerprint Verification

When the alert is raised, a doctor arrives at the room and scans their fingerprint. This biometric input is captured through a fingerprint device integrated into the web dashboard using Flask-SocketIO. The scan:

1. Authenticates the medical personnel
2. Logs the intervention time
3. Clears the visual/audio alert

This step enforces in-person verification, preventing remote or unauthorized alert silencing.

3.2.5 Observations and Analysis

1. Client Behavior During Failover

Clients experienced a 4–6 second delay during failover but reconnected automatically. The subscriber resumed displaying data and alerts continued to function.

2. Broker Logs and Role Transition

Broker logs confirmed successful lock transfer and slave promotion to master. No manual intervention was needed. The system handled the transition with minimal interruption.

3. Message Continuity and Duplication

No messages were lost. As expected with QoS 1, one duplicate was observed after failover. This is considered acceptable and is easily handled at the application layer.

4. Delay and Recovery Time

The full failover and recovery process took under 6 seconds. This is reasonable for cold-standby architectures and suitable for most hospital monitoring scenarios.

5. Accountability via Fingerprint Verification

The fingerprint interaction ensured that only authorized personnel could clear an alert. This process:

- Recorded identity and timestamp
- Enforced physical presence at the patient room
- Prevented accidental or remote alert clearance

It complements the HA system by reinforcing trust and operational transparency.

Advantages and Limitations of shared storage HA

Advantages:

1. **Transparent failover** without manual steps.
2. **Message continuity and persistence** maintained.
3. **Simple configuration** and architecture.
4. **Extensible design** allows biometric confirmation.
5. **Low resource usage:** requires fewer nodes compared to clustering solutions.

Limitations:

1. **Shared storage risk:** it is a single point of failure.
2. **Cold standby delay:** introduces a brief switchover pause.
3. **Limited scalability:** not horizontally scalable for large deployments.
4. **No load balancing:** standby broker remains idle until failure.

3.2.6 Discussion

This chapter presented the design and implementation of a high-availability MQTT system using a shared storage model with Apache ActiveMQ. It demonstrated that automatic broker failover can be achieved with minimal downtime and no data loss, even under fault conditions. The inclusion of biometric alert confirmation added a layer of accountability, reinforcing the reliability and operational trustworthiness of the architecture in healthcare environments.

3.3 Inter-Broker Communication with Mosquitto

3.3.1 Problem Modeling

In real-world deployments—particularly in healthcare—IoT systems rarely operate in isolation. Instead, they span across departments, buildings, or even entirely different institutions. In our previous scenario, we focused on ensuring high availability within a hospital’s MQTT messaging system. That setup guaranteed reliability for critical data flowing from in-room sensors to the hospital’s internal monitoring systems. However, healthcare doesn’t stop at the hospital door.

Many patients discharged from inpatient care remain in vulnerable states of recovery. To ensure continuity of care, hospitals may provide these patients with wearable sensors that monitor key health metrics such as body temperature, oxygen saturation (SpO₂), and heart rate. These devices generate real-time data that must still be monitored—especially in the early stages of post-treatment recovery. But since these patients reside outside the hospital infrastructure—often in retirement homes or assisted living centers—the MQTT messages generated by these wearables are published into a different broker domain. This introduces a new architectural challenge: how can the hospital’s internal MQTT system continue to receive and respond to alerts originating from remote, external environments ?

This scenario explores that challenge by modeling a realistic bridge between two MQTT brokers—one representing the retirement home, and the other the hospital. The goal is to ensure that emergency alerts, such as fall detection or abnormal heart rate events, published to the retirement home’s broker are seamlessly forwarded to the hospital’s system. This allows hospital staff to receive alerts in real-time, as if the patient were still inside the facility.

To support this, the retirement home deploys an autonomous network of embedded devices and wearable sensors that continuously monitor resident well-being. These include:

- **Fall Detection Sensors**, installed in resident rooms, detect sudden impacts and prolonged inactivity—critical indicators for elderly care.
- **Wearable Heart Rate Monitors**, which detect abnormal cardiac patterns like bradycardia or arrhythmia and publish emergency data.
- **Inactivity Detection Subsystems**, which use a combination of motion, bed presence, and door status sensors to detect possible unconsciousness or isolation events.

These sensor alerts are published to a local MQTT broker (Broker A) at the retirement home. A **static bridge**, configured on Broker A, forwards a selected set of topics in real time to the hospital’s MQTT broker (Broker B). This bridge serves as a lightweight inter-domain communication layer, enabling the hospital to receive and act upon alerts from patients even when they are physically located offsite.

Contextual Continuity: Beyond Hospital Walls

This scenario represents the logical continuation of the first one. While the hospital system ensures high availability internally, the bridging architecture extends its reach beyond institutional walls—turning distributed recovery spaces like retirement homes into fully monitored extensions of the hospital network. The broker bridge thus becomes a critical component in enabling seamless emergency coordination, enhancing the responsiveness of healthcare systems that span multiple physical and administrative boundaries.

Beyond the immediate use case of elderly care, this architectural pattern is widely applicable across sectors. Disaster response teams, rural clinics, or mobile healthcare units can similarly publish to local brokers while remaining connected to centralized monitoring facilities through static bridging. This design not only improves system interoperability but also opens the door to scalable, federated MQTT deployments where multiple domains contribute to a unified situational awareness. In this way, the model contributes to the evolution of decentralized, resilient, and patient-centric healthcare infrastructures.

🔑 Use Case Diagram

The following use case diagram presents a simplified view of the key interactions in this bridging scenario. It shows how sensor-generated alerts from the retirement home are transmitted to the hospital's monitoring system through a unidirectional static bridge. While the diagram abstracts away technical details, it provides a clear and accessible overview of the system's functional roles, making it easier to understand the core flow of emergency messages across the two broker domains.

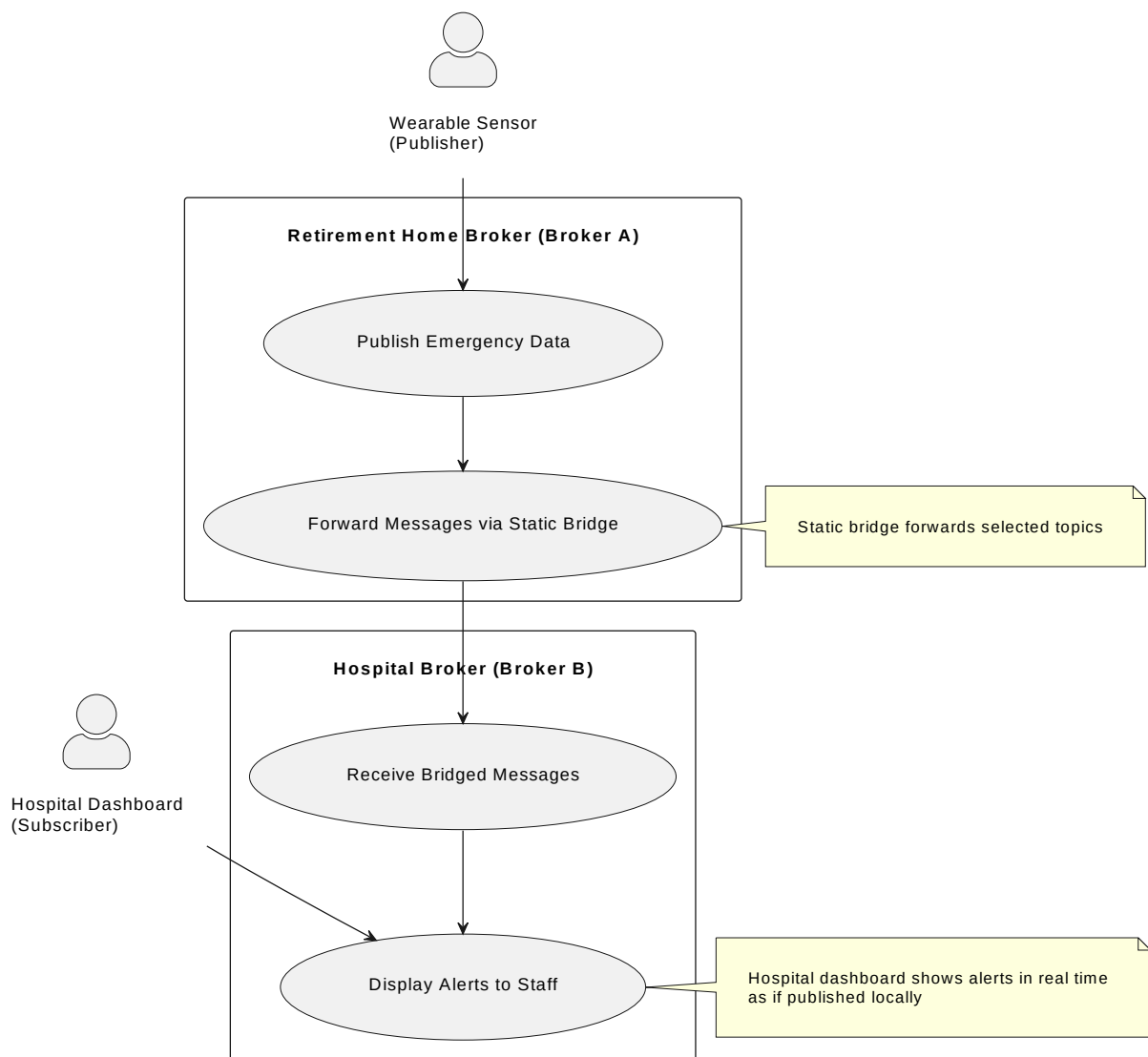


Figure 3.8: Use case diagram for MQTT message bridging from retirement home to hospital

3.3.2 Architecture and Design

The architecture of this scenario consists of two Eclipse Mosquitto brokers: **Broker A** (representing the retirement home) and **Broker B** (representing the hospital system). Each broker operates independently, with local MQTT clients connected to their respective domains. A static bridge is configured on Broker A to forward selected topics to Broker B, enabling unidirectional message propagation from the retirement home to the hospital.

1. Publishing Clients

The publishing clients connected to Broker A simulate the embedded devices deployed within the retirement home. These include fall detection sensors, wearable ECG monitors, and the Inactivity Detection Subsystem. Each device acts as an MQTT publisher and transmits emergency alerts to Broker A using predefined topic filters (e.g., `retirement/emergency/fall` or `retirement/emergency/heart`).

2. Inactivity Subsystem Logic

To better illustrate how the Inactivity Detection Subsystem processes sensor data before publishing an emergency alert, Figure 4.2 presents its internal decision-making sequence. The diagram captures the interaction among three distinct sensor inputs that must all indicate inactivity or isolation before an alert is triggered.

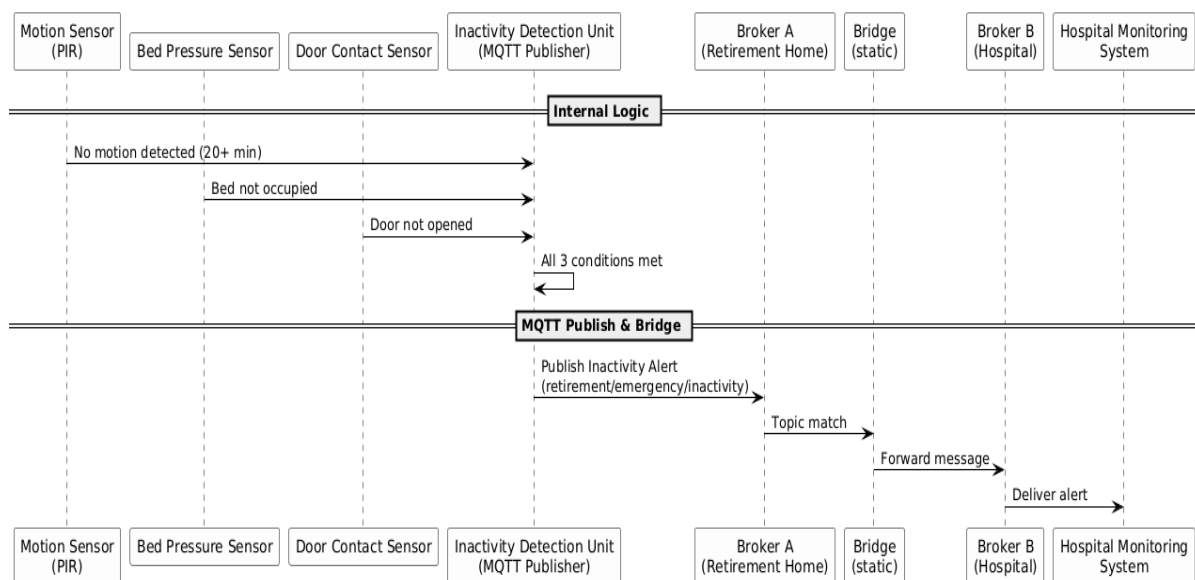


Figure 3.9: Internal Sequence – Inactivity Detection Subsystem. The subsystem evaluates three sensor inputs before publishing an emergency alert.

3. Hospital Subscriber

The subscribing client connected to Broker B simulates the hospital’s monitoring or dispatch infrastructure. It listens to the bridged emergency topics and immediately relays alerts to medical personnel through a real-time dashboard. This dashboard, built with Flask and Flask-SocketIO, uses WebSocket communication to push updates instantly to the browser. By avoiding polling and maintaining a persistent connection, this design ensures that emergency alerts from remote sites appear on-screen with minimal delay—providing the hospital staff with timely, actionable insights as if the data were published locally.

i WebSocket Role: WebSocket is a versatile, full-duplex protocol that enables low-latency communication between web clients and servers. It is particularly useful in real-time dashboards, as it allows the server to push updates without polling. MQTT brokers like Mosquitto and HiveMQ support MQTT over WebSocket, allowing direct browser integration—making it easier to embed real-time health alerts into web-based hospital interfaces.

4. Bridge Setup

The bridge configuration is statically defined in the Mosquitto configuration file on Broker A. It includes the destination address of Broker B, a bridge-specific client identifier, and a list of topic filters that determine which messages should be forwarded. The bridge runs continuously as part of the broker’s runtime, reconnecting automatically if the connection is interrupted.

5. Topic Transparency

Messages published to topics such as `retirement/emergency/fall` are forwarded by the bridge to Broker B under the same topic path. No topic remapping is used in this scenario, allowing the hospital-side subscriber to consume messages transparently, without needing to be aware of the originating broker.

6. Operational Separation

This architecture supports logical separation of domains. Each institution maintains independent control over its broker configuration, authentication mechanisms, and client sessions. The bridge serves strictly as a message transport layer between trusted endpoints. As long as the bridge remains connected and correctly configured, the hospital-side subscriber receives messages as if they were published locally.

Sequence Diagram

The following sequence diagram illustrates the end-to-end flow of messages in the bridging scenario. It captures how emergency alerts are generated by devices in the retirement home, published to Broker A, and forwarded to the hospital's system via the static bridge. In addition to normal message flow, the diagram also shows how the system behaves during a broker failure and its subsequent recovery, emphasizing the bridge's role in maintaining continuity.

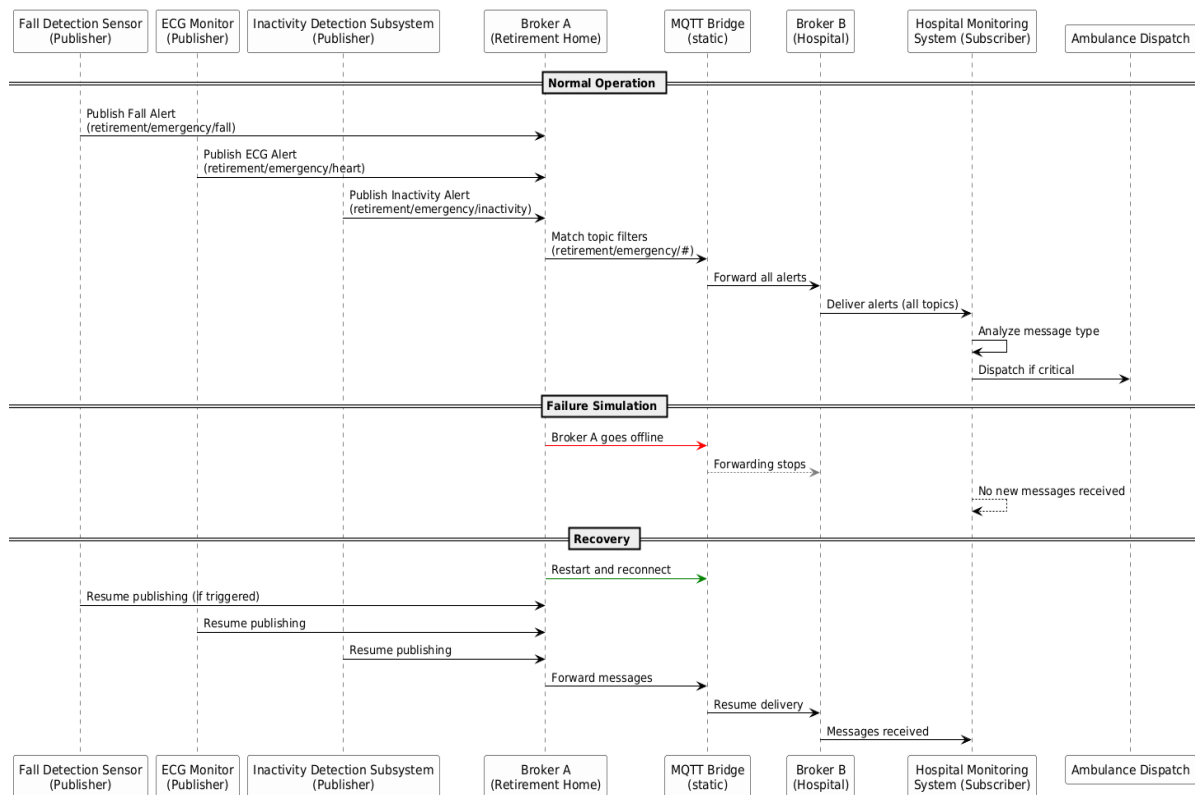


Figure 3.10: Sequence Diagram – Message flow from retirement home sensors to hospital system, including broker failure and recovery.

This diagram illustrates the modularity of the system: each component—from sensors to brokers—operates independently. This decoupling improves fault isolation and allows updates (e.g., replacing sensors) without disrupting hospital systems. MQTT's loose coupling, combined with Mosquitto's bridge, supports scalable inter-site communication. The architecture is easy to replicate across multiple facilities—homes, clinics, or mobile units. This transforms the hospital into a central node for distributed healthcare monitoring. The design is maintainable, extensible, and resilient under varying conditions. It enables real-time emergency coordination across independent yet connected environments.

3.3.3 Algorithmic Overview

1. Broker Role Logic

The logic behind Mosquitto's static bridging mechanism is straightforward and deterministic. Once the bridge is configured on Broker A, it becomes an integrated part of the broker's runtime behavior. The bridge process is responsible for establishing a connection to the remote broker (Broker B), subscribing or publishing to specified topics according to the bridge direction, and ensuring continued message delivery, even in the event of temporary disconnection. The following pseudocode summarizes the core behavior of the bridge process:

```
1 On Broker A startup:
2   - Initialize local MQTT services
3   - Attempt connection to Broker B using bridge configuration
4   - Authenticate and establish session (remote_clientid)
5   - Register topic filters for forwarding (retirement/emergency/#)
6
7 During runtime:
8   For each incoming local message:
9     If topic matches bridge filter:
10      Forward message to Broker B
11
12 If Broker B becomes unreachable:
13   - Retry connection after 'restart_timeout' interval
14   - Resume forwarding once connection is restored
```

Listing 3.2: Describing the startup and runtime behavior of Broker A (Retirement Home) configured to bridge emergency messages to Broker B (Hospital). It initializes MQTT services, establishes a session, and forwards matched topics; reconnection logic handles remote unavailability.

🔒 Unidirectional Bridge Behavior: This mechanism ensures that only messages explicitly permitted by the topic filters are forwarded across the bridge. The bridge does not replicate MQTT session state, retained messages, or broker-level metadata. It strictly forwards MQTT PUBLISH packets that match the configured topic rules.

📌 Design Advantages: The unidirectional nature of the bridge in this scenario guarantees that no message loops or cyclic traffic can occur, even in more complex deployments. This makes the design predictable, easy to debug, and resilient to common misconfiguration issues.

The flowchart below captures the runtime behavior of Mosquitto’s static bridge component. It includes the key decision points for connection handling, message forwarding and retry mechanisms in the event of failure.

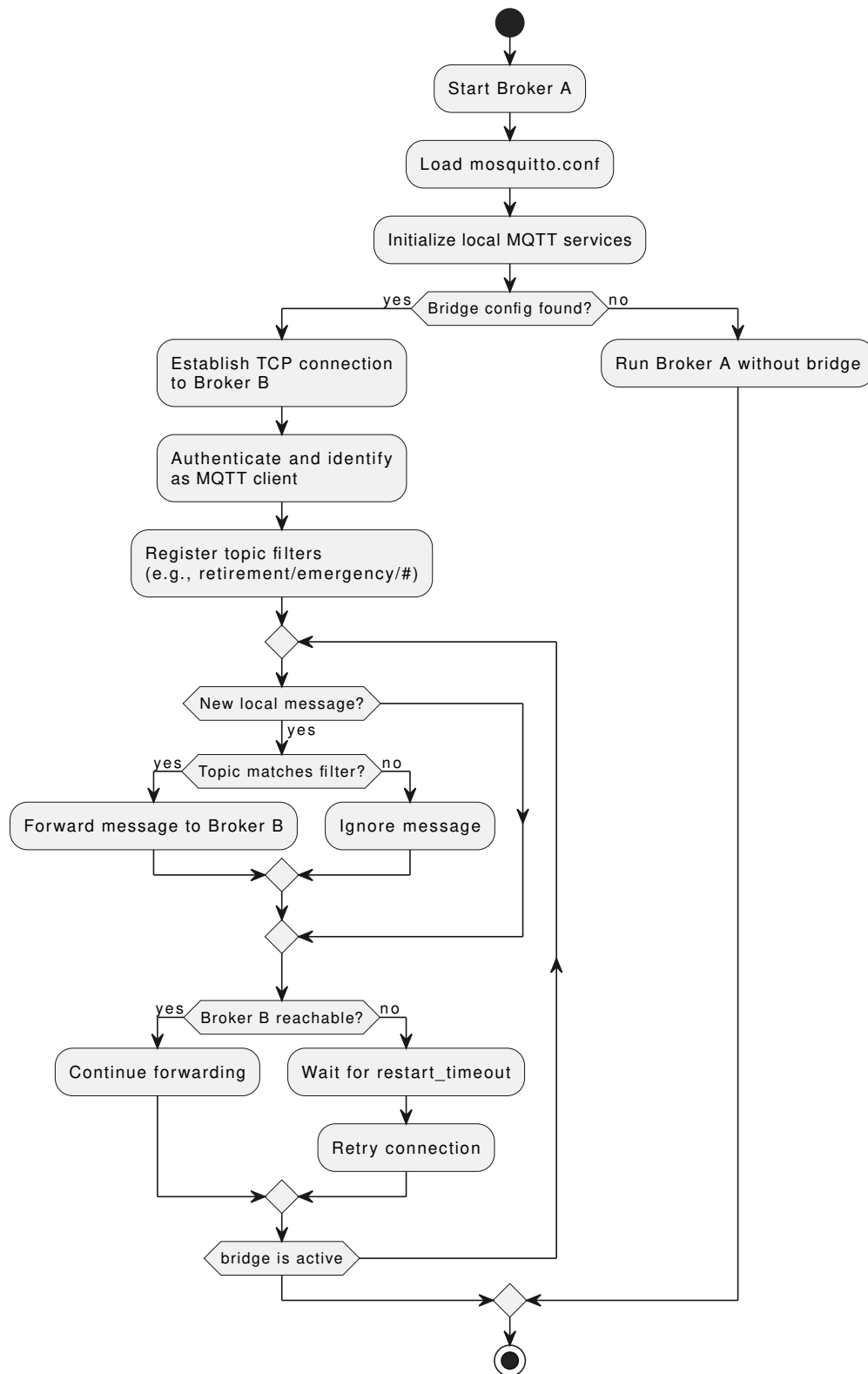


Figure 3.11: Flowchart – Static Bridge Logic in Mosquitto. This figure outlines the control flow from configuration loading through connection, topic filtering, message forwarding, and recovery attempts.

2. Bridge Configuration and Logic

The bridge between Broker A and Broker B is configured statically using Mosquitto's built-in bridging functionality. This setup is defined in the broker's configuration file (typically `mosquitto.conf`) and instructs Broker A (retirement home) to connect to Broker B (hospital) and forward selected MQTT messages.

A standard static bridge configuration includes the following key parameters:

<code>connection <name></code>	Unique label identifying the bridge session.
<code>address <host>:<port></code>	IP address or hostname and port of Broker B.
<code>remote_clientid <id></code>	MQTT client ID used by Broker A as bridge.
<code>topic <filter> <direction></code>	Topics to forward; direction can be <code>in</code> , <code>out</code> , or <code>both</code> .
<code>cleansession <true false></code>	Whether to maintain a persistent session across reconnects.
<code>restart_timeout</code>	Time (in seconds) between reconnection attempts.

Table 3.1: Key parameters in a static MQTT bridge configuration

An example bridge configuration for this scenario is shown below:

```

1 connection hospital-bridge
2 address broker_b_hostname:1883
3 remote_clientid retirementBridge
4 topic retirement/emergency/# out
5 cleansession true
6 restart_timeout 10

```

Listing 3.3: Example static bridge configuration from Broker A to Broker B

1. Forwarding Runtime Behavior

This setup ensures that any message published on Broker A to topics matching `retirement/emergency/#` will be forwarded to Broker B exactly as-is. This includes alerts from fall detection sensors, heart rate monitors, and inactivity detectors. The bridge runs as a background process inside Broker A and remains active for the duration of the broker runtime. If Broker B becomes temporarily unavailable, the bridge will automatically retry after the defined timeout period.

2. Loop Prevention and Directionality

In multi-broker topologies, careful configuration of directionality and topic filters is critical to prevent message loops. In this scenario, the bridge is configured as a one-way channel (out) from Broker A to Broker B. This directionality ensures that messages cannot circulate between brokers, eliminating the risk of recursive forwarding.

3. Topic Transparency

While Mosquitto supports prefixing or topic remapping to distinguish bridged messages, this scenario preserves the original topic format for clarity and traceability. Subscribers on Broker B receive emergency alerts exactly as published, without any topic transformation.

4. Conceptual Overview

The configuration mapping diagram links the key Mosquitto bridge parameters—such as connection direction and topic filters—to their runtime behavior. It provides a conceptual overview of how the configuration translates into operational logic.

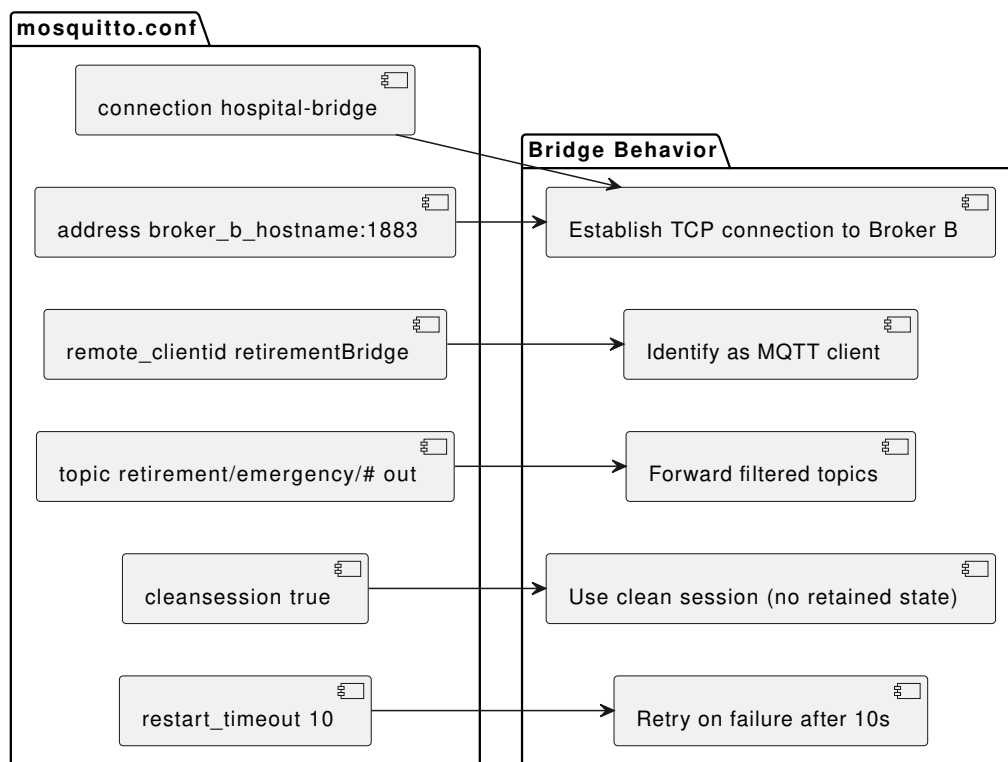


Figure 3.12: Configuration Mapping Diagram showing how Mosquitto bridge parameters such as connection direction and topic filters influence runtime behavior. Note that this illustration simplifies the configuration and does not cover all optional or advanced syntax available in Mosquitto.

3.3.4 Scenario Execution

To validate the bridging setup and observe message forwarding behavior, the system was deployed using two Eclipse Mosquitto brokers running on separate machines or ports. Broker A was configured with a static bridge targeting Broker B, and both brokers were launched sequentially. Two Python-based MQTT clients were used: a publisher connected to Broker A and a subscriber connected to Broker B.

📋 The steps executed in this scenario are as follows:

- 1. Broker Setup:** Broker B (hospital) was started first to ensure it was ready to accept incoming bridge connections. Then, Broker A (retirement home) was launched with the configured bridge. Logs confirmed that the bridge successfully established a session using the specified `remote_clientid` and subscribed to the relevant topic filters.
- 2. Client Connection:** A simulated publishing device was launched and connected to Broker A. It began publishing simulated heart rate data every 5 seconds to the topic `retirement/emergency/heart`. Meanwhile, the subscriber client—connected to Broker B—subscribed to the same topic and awaited incoming messages. The messages emulated abnormal ECG readings to simulate real-world cardiac monitoring use cases. In extended tests, additional messages were also published to topics such as `retirement/emergency/fall` and `retirement/emergency/inactivity` to verify forwarding of multiple alert types.
- 3. Normal Operation:** Messages published by the client on Broker A were forwarded to Broker B via the static bridge. The subscriber received the messages without delay, as if they had originated from a local source. No message loss or duplication was observed under stable network conditions.
- 4. + Failure Simulation:** To test resilience, Broker A was intentionally stopped. As expected, the bridge connection was dropped, and the subscriber on Broker B ceased receiving messages. Since the publisher was also disconnected, message production halted during this interval.
- 5. + Recovery:** Broker A was restarted, and the bridge reconnected to Broker B automatically within the configured `restart_timeout` window. Upon re-establishing the session, the publisher resumed transmitting, and the subscriber continued receiving alerts in real time.

⇔ Message Flow and System Behavior

The communication bridge between the retirement home (Broker A) and the hospital (Broker B) was designed to forward only high-priority alerts. While the bridge remains persistently active and technically capable of forwarding all messages, non-critical data—such as stable or normal vitals—are retained locally at Broker A. This design aligns with the hospital’s operational focus, which is to receive only data requiring medical attention.

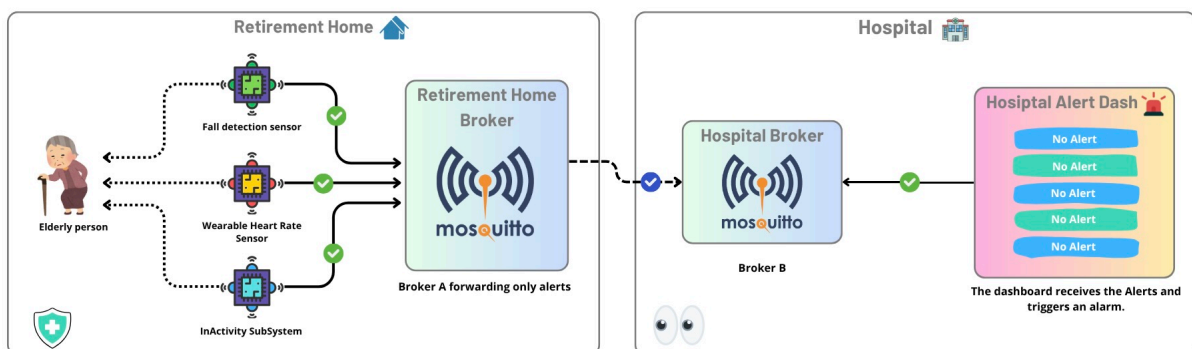


Figure 3.13: Normal condition – Stable vital signs are published locally at Broker A but not forwarded to the hospital.

This selective forwarding strategy reduces unnecessary data traffic, preserves hospital-side processing resources, and strengthens data relevance in emergency dashboards. Only in the event of abnormal readings—such as sudden falls or irregular heart rates—are alerts published on specific topics (e.g., `retirement/emergency/fall`, `retirement/emergency/hearttrate`), which are explicitly allowed to pass through the bridge.

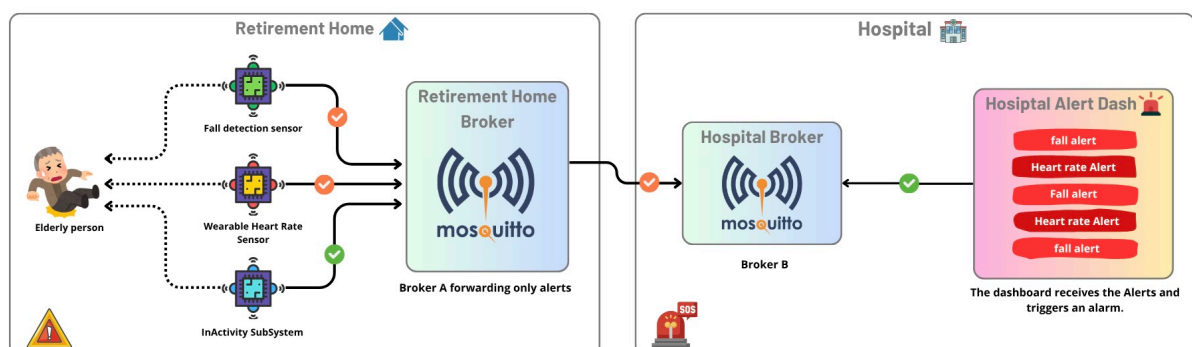


Figure 3.14: Emergency condition – Anomalous vitals and a fall event are detected and forwarded to the hospital. The dashboard receives the alert and triggers an alarm.

This execution confirmed that static bridging works reliably for unidirectional message propagation across MQTT brokers. It also demonstrated the bridge’s ability to recover from disconnection events without manual intervention. The subscriber remains unaware of the bridging mechanism, preserving the illusion of direct connectivity and satisfying the architectural goal of transparent inter-domain communication.

To better illustrate the system’s runtime response to broker failure and recovery, the following timeline diagram depicts the sequence of events—from initial message publication to automatic bridge reconnection and message resumption.

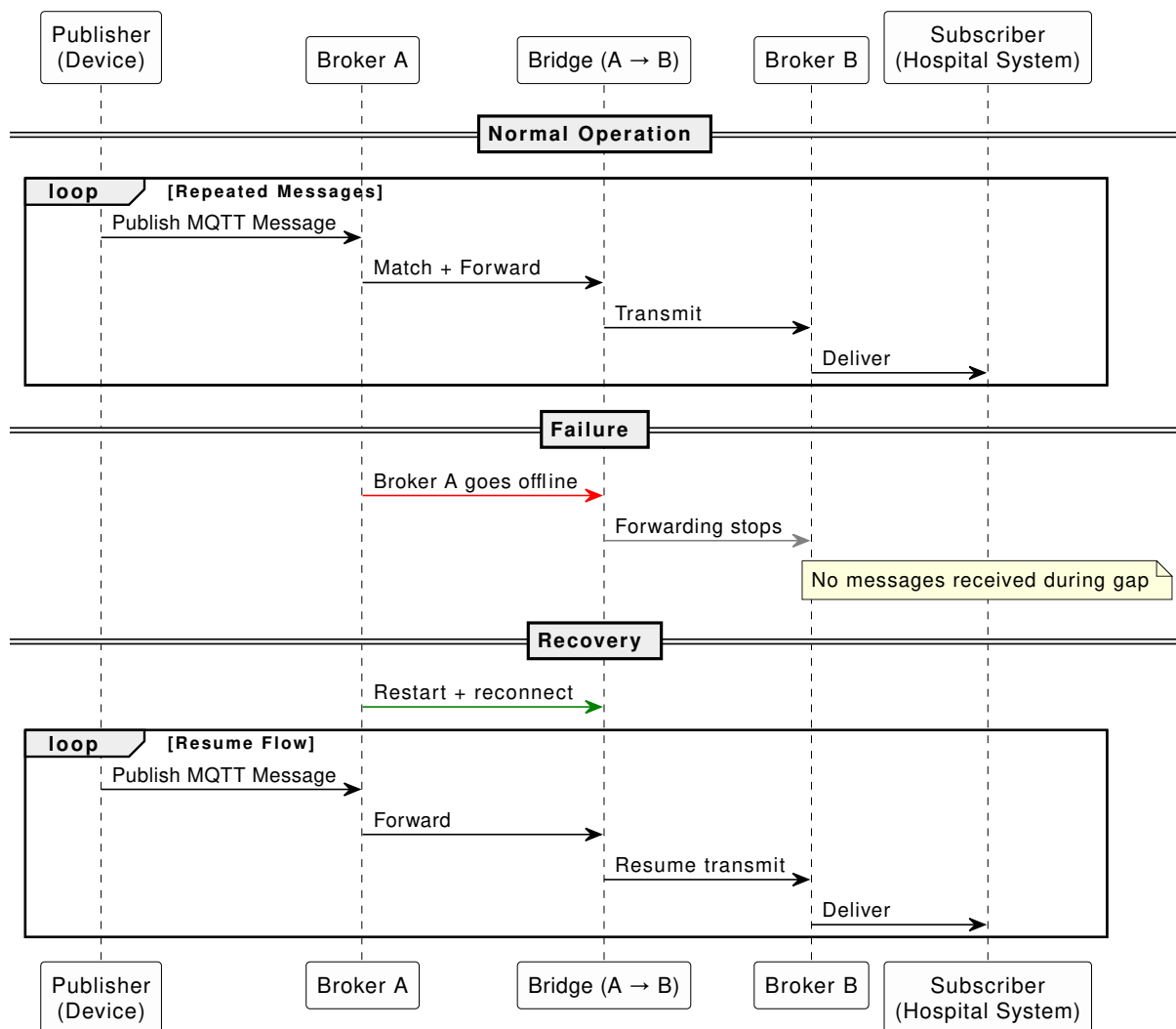


Figure 3.15: Execution Timeline – Message delivery flow under normal operation, bridge failure, and automatic recovery.

3.3.5 Observations and Analysis

The bridging scenario produced clear and consistent results across normal operation, failure, and recovery phases. This section summarizes the observed behavior from both the client and broker perspectives.

1. Message Forwarding Behavior

During normal operation, every message published to `retirement/emergency/heart` on Broker A was received by the subscriber on Broker B without noticeable delay. The subscriber did not require any knowledge of the message's origin; it behaved as if the data were published locally. Broker logs on both sides confirmed successful message delivery and proper session establishment through the static bridge.

2. Bridge Stability and Reconnection

When Broker A (retirement home) was stopped to simulate a failure, the bridge connection was dropped as expected. The subscriber remained connected to Broker B (hospital), but no new messages arrived during the downtime. Upon restarting Broker A, Mosquitto's bridge logic automatically reconnected within the configured `restart_timeout` period. Forwarding resumed immediately and without manual intervention. This confirmed the bridge's built-in resilience to temporary remote broker failures.

3. Subscriber Consistency and Duplicate Handling

No duplicate messages were observed under QoS 0 or QoS 1 in this scenario. Since the bridge acts as a standard MQTT client from Broker B's perspective, message delivery followed MQTT protocol rules. The subscriber maintained consistent behavior across multiple reconnect cycles. In practical deployments, using QoS 1 or QoS 2 with appropriate duplicate handling remains advisable for more critical systems.

4. Failure Visibility and Impact

Although the bridge recovered automatically, it is important to note that no messages were forwarded during Broker A's downtime. This created a temporary communication gap, which may be significant in safety-critical environments. The bridge does not queue or store messages while disconnected, unless additional mechanisms such as persistent sessions or retained messages are used—both of which were beyond the scope of this scenario.

Advantages and Limitations of Broker Bridging

The static bridging configuration implemented using Mosquitto provided a lightweight and effective mechanism for enabling inter-broker communication. This section summarizes the key benefits and limitations observed during the retirement home and hospital scenario.

Advantages

1. **Simplicity of Setup:** Bridging requires minimal configuration and no external dependencies. Once defined in the Mosquitto configuration file, the bridge operates seamlessly as part of the broker's runtime.
2. **Decentralized Operation:** Each broker remains autonomous, managing its own clients, topics, and authentication policies. This makes bridging ideal for systems operated by separate institutions—such as a healthcare provider and a residential facility.
3. **Transparent Message Delivery:** Subscribers on the hospital side receive messages exactly as if they were published locally. No special adjustments or awareness of bridging are required by clients.
4. **Automatic Recovery:** Mosquitto handles bridge reconnections internally. If the destination broker (Broker B) becomes temporarily unavailable, the bridge resumes forwarding automatically once connectivity is restored.

Limitations

1. **No State Replication or HA:** Unlike clustered or high-availability MQTT deployments, bridging does not replicate session state, retained messages, or subscription metadata. It only forwards matching messages at the time they are published.
2. **Message Loss During Outage:** If Broker A is offline or disconnected, no messages can be buffered or queued for later delivery unless additional features such as persistent sessions or message persistence are configured.
3. **Unidirectional and Static Configuration:** Bridges are typically configured in one direction. Bidirectional setups require careful topic design to prevent feedback loops or message duplication—especially in multi-broker networks.
4. **Limited Scalability:** While effective for point-to-point or segmented environments, static bridging becomes harder to manage in systems requiring dynamic discovery, routing or full-mesh connectivity.

3.3.6 Discussion

By implementing a unidirectional static bridge between Mosquitto brokers, this chapter demonstrated how distributed MQTT environments can be connected across institutional boundaries. The architecture allowed emergency alerts to be forwarded transparently and in real time from a retirement home to a hospital dashboard. Although the approach lacks HA features, it provides a lightweight and effective model for decentralized IoT deployments. In summary, Mosquitto's bridging functionality offers a practical solution for linking MQTT brokers across institutional or geographic boundaries. It is particularly well-suited for systems with clear separation of concerns, such as those seen in healthcare deployments. However, it is not a substitute for high-availability or clustering when long-term reliability, state synchronization, and message continuity are mission-critical.

3.4 Conclusion of the chapter

This chapter examined two complementary architectural solutions to address key resilience challenges in MQTT-based healthcare systems. The first scenario focused on broker-level high availability using Apache ActiveMQ, demonstrating how fault tolerance can be achieved through shared persistence and automatic failover—ensuring continuous message flow even under broker crashes. The second scenario extended this resilience across physical and institutional boundaries, using Mosquitto's static bridging to forward emergency alerts from a retirement home to a hospital dashboard in real time.

Together, these approaches illustrate that MQTT's lightweight protocol can be adapted for both intra-domain reliability and inter-domain coordination. While ActiveMQ HA ensures local availability and accountability, Mosquitto bridging enables decentralized expansion without introducing excessive complexity. Though neither solution fully replicates the capabilities of clustered messaging systems, both offer practical trade-offs for healthcare environments where timely and trustworthy communication is critical.

4 Implementation

4.1 Implementation Overview

This chapter details the setup of the two MQTT-based systems and their evaluation. It covers broker configuration, client behavior, and how each architecture responds to failure scenarios in real-world conditions.

4.2 Environment Setup

4.2.1 General Development Environment

All development and testing were performed on a Windows 10 environment using Python 3.13.2. The system was implemented in a cross-platform manner to ensure compatibility with other operating systems with minimal changes.

PyCharm Community Edition was used as the main integrated development environment (IDE). It provided support for virtual environments, real-time debugging, and integrated terminal access—features that streamlined the development of MQTT clients and interface components.

The figure below shows the development environment in use:

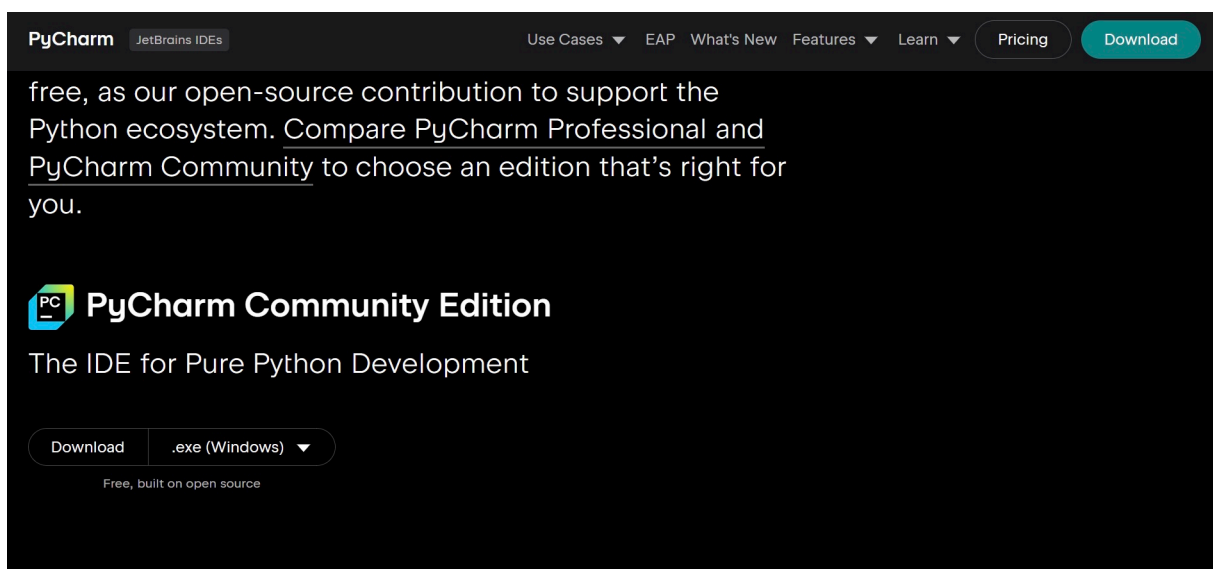


Figure 4.1: PyCharm Community Edition – IDE used for MQTT client development

4.2.2 Dependencies and Toolchain

The following tools and libraries were used throughout the implementation:

Core Dependencies:

1. Apache ActiveMQ Classic 6.1.6

MQTT broker for Scenario 1 with shared storage high availability.

2. Eclipse Mosquitto 2.0.3

Lightweight MQTT broker used in Scenario 2 with static bridging.

3. Paho MQTT library 1.6.1

MQTT client library for all publishers and subscribers.

4. Flask 2.0.3 and Flask-SocketIO 5.1.1

Backend engine and WebSocket handler for the real-time dashboard.

5. Werkzeug 2.0.3

Internal Flask dependency for HTTP handling.

6. Eventlet 0.33.0

Asynchronous server used to enable real-time communication with Flask-SocketIO.

Interface and Visualization Tools:

1. Tkinter 8.6

A standard Python GUI library used to create simple, native desktop interfaces.

2. PyQt6 6.9.0

A feature-rich Python binding for the Qt application framework, used to build complex and modern graphical interfaces.

3. Matplotlib 3.10.3

A comprehensive data visualization library for creating static, animated, and interactive plots in Python.

All required Python packages were listed in a `requirements.txt` file and installed using the following command:

```
pip install -r requirements.txt
```

⚙️ Tool Selection Rationale

The choice of tools and libraries was guided by factors such as maturity, active community support, and suitability for MQTT-based applications. Apache ActiveMQ was selected for its stable high availability support through shared storage. Eclipse Mosquitto, being lightweight and widely adopted, offered an efficient solution for bridging-based communication. Flask and Flask-SocketIO enabled real-time data visualization with minimal overhead, while PyQt6 and Tkinter provided flexible options for crafting desktop GUIs, catering to both simple and advanced interface requirements.

💎 Version Compatibility

To ensure consistency and reproducibility, all dependencies were explicitly version-pinned in the `requirements.txt` file. Compatibility was verified on multiple systems to guarantee uniform behavior during MQTT communication, GUI interaction, and message processing. This measure helped prevent runtime conflicts and eased deployment across development machines.

🔧 Deployment Considerations

While this project was developed in a controlled environment, the overall design remains lightweight and suitable for deployment on modest hardware such as Raspberry Pi or low-cost virtual machines. For future scaling, lightweight containerization with Docker and basic monitoring scripts (e.g., using `systemd`) could enhance reliability without introducing significant overhead.

🔒 Security Scope

No encryption or authentication (e.g., TLS, client credentials) was implemented in this project, as it was developed in a controlled test environment. For production use, MQTT over TLS, certificate-based authentication, and topic-level access control should be integrated as security enhancements.

4.3 Scenario 1 – Shared Storage High Availability

This implementation builds upon the architecture introduced in Chapter 3, where a master–slave broker pair ensures uninterrupted MQTT messaging via shared KahaDB storage. The following steps demonstrate the configuration and validation of this HA system in practice.

4.3.1 Installing and Running ActiveMQ

This subsection documents the installation process for Apache ActiveMQ Classic 6.1.6. The setup was carried out on a Windows environment to prepare two broker instances for high availability testing.

Step 1 – Download and Extract ActiveMQ

The broker was downloaded from the official Apache ActiveMQ website. Version 6.1.6 was selected to ensure compatibility with MQTT and shared storage functionality.

The figure below shows the download page:

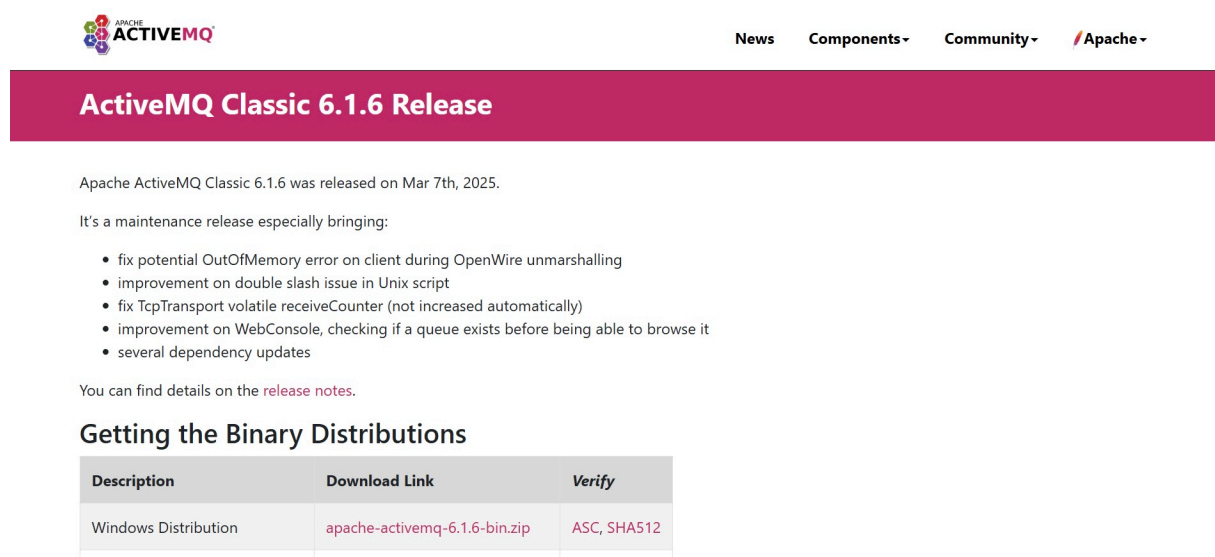


Figure 4.2: Downloading Apache ActiveMQ Classic 6.1.6 from the official website

After downloading, the archive was extracted to a chosen directory to keep the setup straightforward and demonstrate the basic functionality of ActiveMQ.

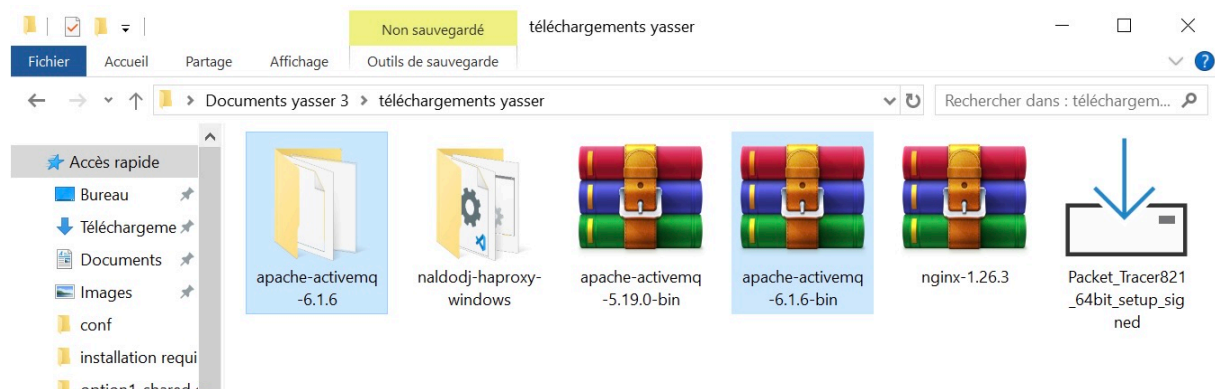


Figure 4.3: Extracting Apache ActiveMQ into a working directory

Step 2 – Exploring the Broker Directory Structure

The extracted folder contains essential subdirectories for configuration and operation. Two key directories for HA setup are:

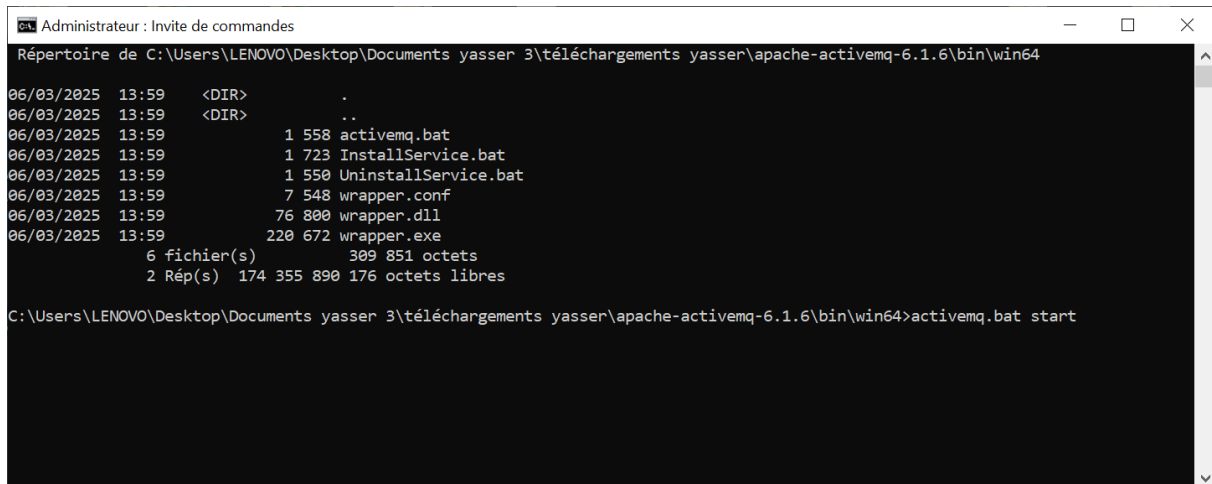
- **/conf:** Contains broker configuration files such as `activemq.xml`, `jetty.xml`, and logging settings.
- **/data:** Hosts persistent message files (KahaDB) used in shared storage HA.

Nom	Modifié le	Type	Taille
bin	06/03/2025 12:59	Dossier de fichiers	
conf	06/03/2025 12:59	Dossier de fichiers	
data	13/04/2025 15:12	Dossier de fichiers	
docs	06/03/2025 12:59	Dossier de fichiers	
examples	06/03/2025 12:59	Dossier de fichiers	
lib	06/03/2025 12:59	Dossier de fichiers	
webapps	06/03/2025 12:59	Dossier de fichiers	
webapps-demo	06/03/2025 12:59	Dossier de fichiers	
activemq-all-6.1.6	06/03/2025 12:59	Executable Jar File	11 114 Ko
LICENSE	06/03/2025 12:59	Fichier	41 Ko
NOTICE	06/03/2025 12:59	Fichier	4 Ko
README	06/03/2025 12:59	Document texte	3 Ko

Figure 4.4: Apache ActiveMQ folder structure – highlighting key directories

Step 3 – Launching the Broker

To start the broker, the `activemq.bat` script is executed in Command Prompt (Windows). This initializes MQTT connectors, the web admin console, and persistence services.



```

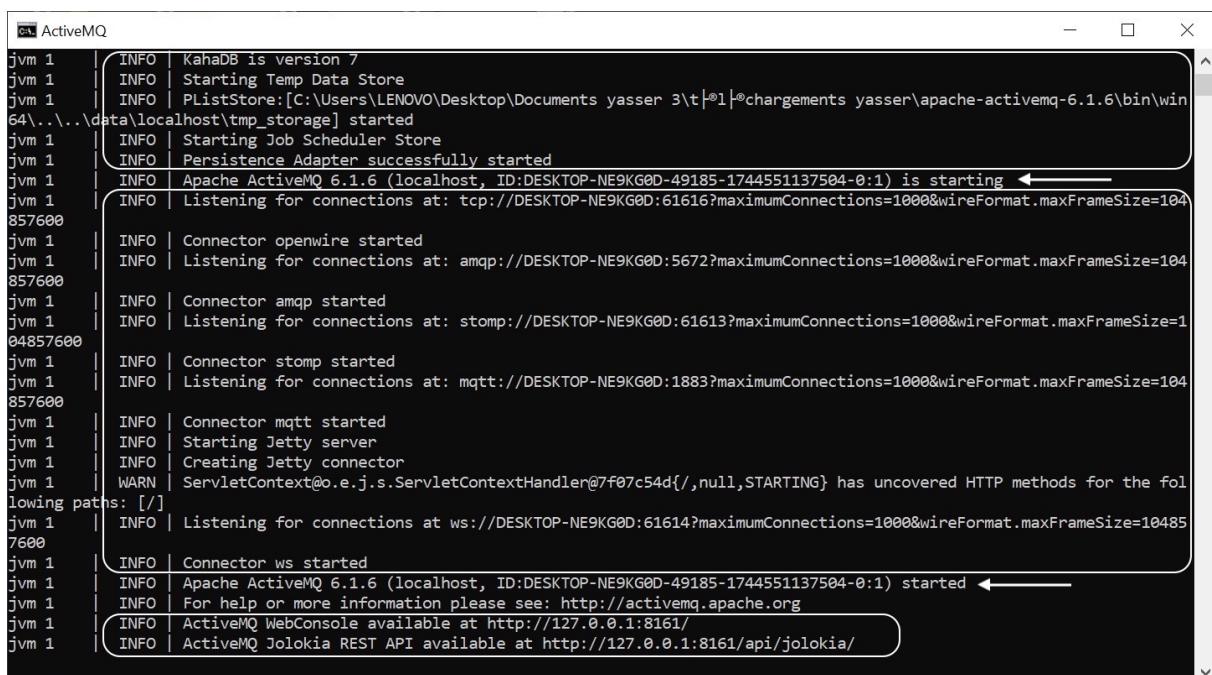
Administrateur : Invite de commandes
Répertoire de C:\Users\LENOVO\Desktop\Documents yasser 3\téléchargements yasser\apache-activemq-6.1.6\bin\win64
06/03/2025 13:59 <DIR> .
06/03/2025 13:59 <DIR> ..
06/03/2025 13:59          1 558 activemq.bat
06/03/2025 13:59          1 723 InstallService.bat
06/03/2025 13:59          1 550 UninstallService.bat
06/03/2025 13:59          7 548 wrapper.conf
06/03/2025 13:59         76 800 wrapper.dll
06/03/2025 13:59        220 672 wrapper.exe
          6 fichier(s)          309 851 octets
          2 Rép(s) 174 355 890 176 octets libres

C:\Users\LENOVO\Desktop\Documents yasser 3\téléchargements yasser\apache-activemq-6.1.6\bin\win64>activemq.bat start
  
```

Figure 4.5: Starting the broker via `activemq.bat start`

Broker startup logs confirm successful initialization:

- MQTT transport enabled (port 1883 by default)
- KahaDB persistence adapter initialized
- Jetty web interface enabled at `http://localhost:8161`



```

ActiveMQ
jvm 1 | INFO | KahaDB is version 7
jvm 1 | INFO | Starting Temp Data Store
jvm 1 | INFO | PlistStore:[C:\Users\LENOVO\Desktop\Documents yasser 3\t*\chargements yasser\apache-activemq-6.1.6\bin\win
64\...\data\localhost\tmp_storage] started
jvm 1 | INFO | Starting Job Scheduler Store
jvm 1 | INFO | Persistence Adapter successfully started
jvm 1 | INFO | Apache ActiveMQ 6.1.6 (localhost, ID:DESKTOP-NE9KG0D-49185-1744551137504-0:1) is starting
jvm 1 | INFO | Listening for connections at: tcp://DESKTOP-NE9KG0D:6161?maximumConnections=1000&wireFormat.maxFrameSize=104
857600
jvm 1 | INFO | Connector openwire started
jvm 1 | INFO | Listening for connections at: amqp://DESKTOP-NE9KG0D:5672?maximumConnections=1000&wireFormat.maxFrameSize=104
857600
jvm 1 | INFO | Connector amqp started
jvm 1 | INFO | Listening for connections at: stomp://DESKTOP-NE9KG0D:61613?maximumConnections=1000&wireFormat.maxFrameSize=1
04857600
jvm 1 | INFO | Connector stomp started
jvm 1 | INFO | Listening for connections at: mqtt://DESKTOP-NE9KG0D:1883?maximumConnections=1000&wireFormat.maxFrameSize=104
857600
jvm 1 | INFO | Connector mqtt started
jvm 1 | INFO | Starting Jetty server
jvm 1 | INFO | Creating Jetty connector
jvm 1 | WARN | ServletContext@o.e.j.s.ServletContextHandler@7f07c54d[/,null,STARTING] has uncovered HTTP methods for the fol
lowing paths: [/]
jvm 1 | INFO | Listening for connections at ws://DESKTOP-NE9KG0D:61614?maximumConnections=1000&wireFormat.maxFrameSize=10485
7600
jvm 1 | INFO | Connector ws started
jvm 1 | INFO | Apache ActiveMQ 6.1.6 (localhost, ID:DESKTOP-NE9KG0D-49185-1744551137504-0:1) started
jvm 1 | INFO | For help or more information please see: http://activemq.apache.org
jvm 1 | INFO | ActiveMQ WebConsole available at http://127.0.0.1:8161/
jvm 1 | INFO | ActiveMQ Jolokia REST API available at http://127.0.0.1:8161/api/jolokia/
  
```

Figure 4.6: Broker startup logs confirming services and persistence configuration

Step 4 – Accessing the Web Console

Once running, the ActiveMQ broker provides a web interface for monitoring and management.

- **URL:** `http://localhost:8161/admin`
- **Default login:** `admin / admin`

Functional Overview: The web console offers an intuitive interface that allows developers to inspect broker status, manage destinations (queues and topics), and monitor connected clients in real time. It is especially helpful in early development stages, providing immediate feedback on message routing and system behavior.

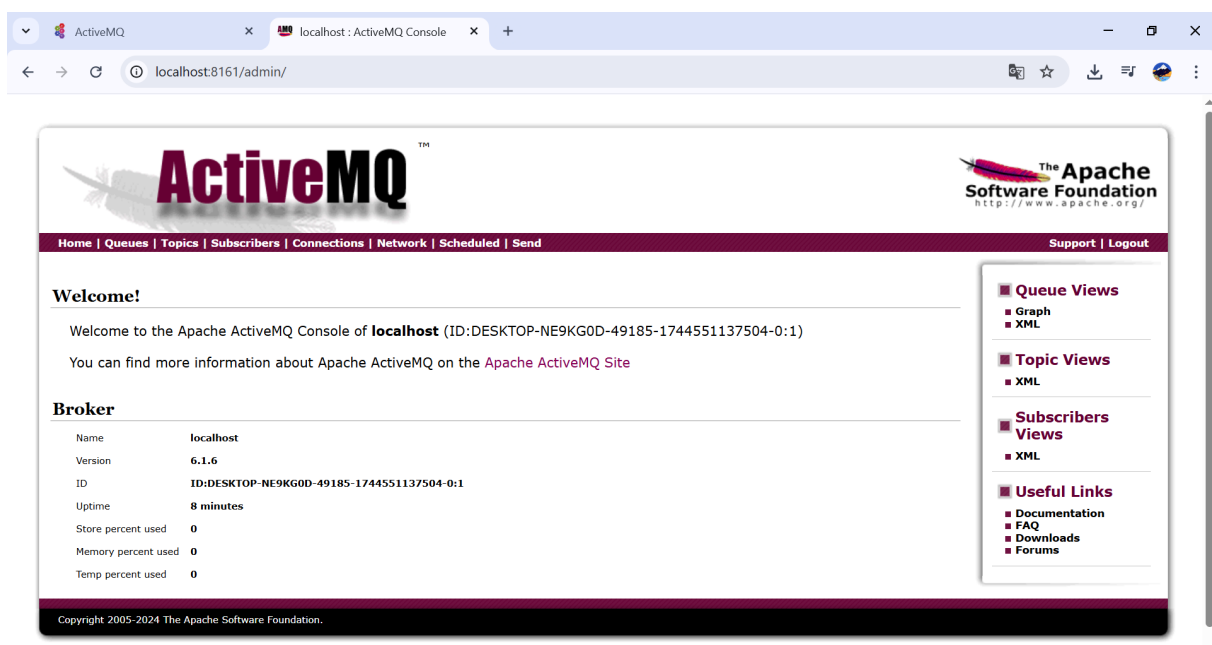


Figure 4.7: ActiveMQ Web Console – topic monitoring and broker status

Security Reminder: The default credentials (`admin / admin`) are acceptable for local development but must be changed in any real-world deployment. Failing to secure the console can expose the broker to unauthorized access and potential message loss or interception.

✂ Troubleshooting Notes

Issue	Solution
Java not recognized	Ensure Java 8+ is installed and <code>JAVA_HOME</code> is set correctly.
Permission denied on ports or folders	Run CMD as Administrator and verify write access to <code>/data</code> .
Broker startup fails with lock error	Ensure no other instance is using the KahaDB lock file.
Web console not loading	Verify Jetty is configured on port 8161 and not blocked by a firewall.

Table 4.1: Common issues during installation and startup

🔑 **Additional Notes:** While the table above outlines the most common issues, users may also encounter problems specific to their environment—such as antivirus interference, missing environment variables, or conflicting services using required ports.

🕒 **Best Practices:** To minimize setup issues, it is recommended to:

- Perform the installation on a clean, minimal environment where possible.
- Always start the broker with administrative privileges on Windows.
- Test broker functionality with simple MQTT clients before integrating into more complex systems.

👍 **Tip for Beginners:** If you're new to ActiveMQ or messaging systems in general, the web console can serve as a live sandbox. You can manually publish and subscribe to topics via the browser to validate that the system is functioning correctly before deploying client applications.

4.3.2 Broker Configuration for Shared Storage HA

This subsection details how the ActiveMQ brokers were configured for high availability using a shared KahaDB message store. The objective is to ensure that only one broker is active at any time, with automatic failover in case of failure.

Step 1 – Preparing the Shared KahaDB Directory

In a shared storage master–slave architecture, both brokers must point to the same physical KahaDB directory. Only one broker can acquire the lock and become active.

In this setup, a local directory was used:

- `C:/activemq-data/kahadb`

This folder was created by:

1. Starting one broker to initialize the folder structure.
2. Copying the initialized `kahadb` folder to a shared location.
3. Configuring both brokers to reference this exact path.

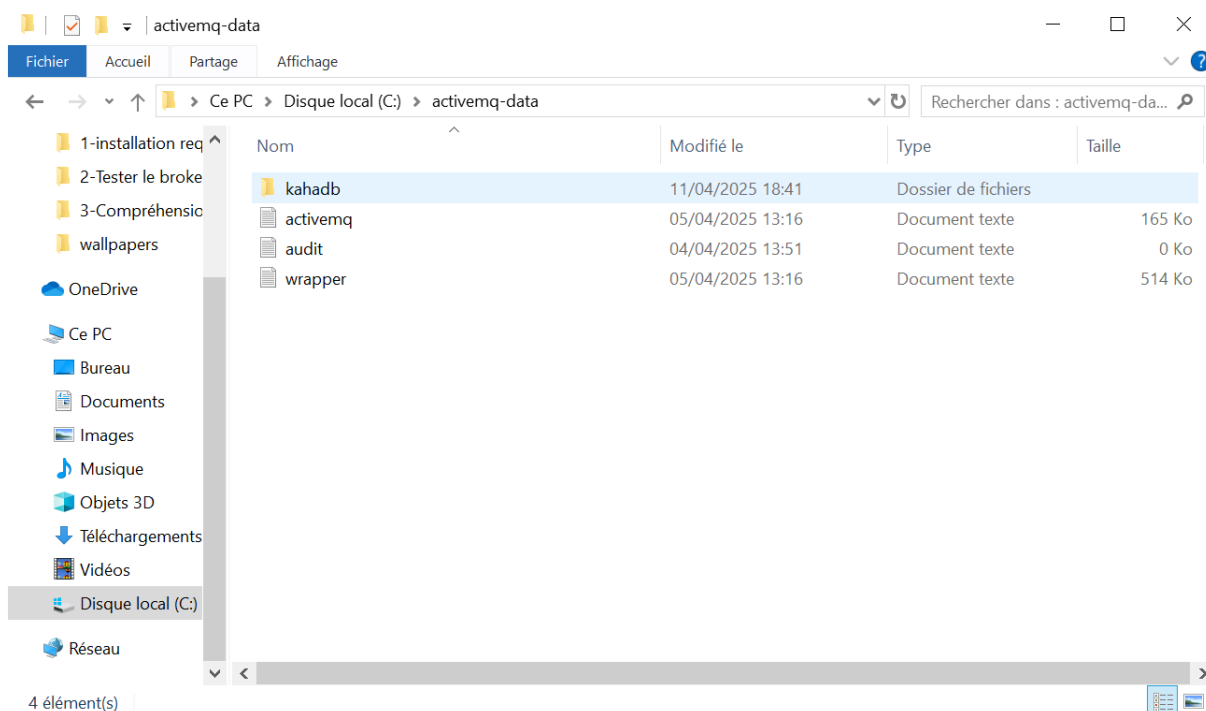


Figure 4.8: Creating a fresh shared `kahadb` directory for local HA testing

Step 2 – Dual Broker Folder Structure

Two broker instances were prepared from the same ActiveMQ distribution:

- `apache-activemq-6.1.6` – master broker
- `apache-activemq-6.1.6-SLAVE` – slave broker

Both were placed under `C:/Programmes` with separate `conf` folders and Jetty servers.

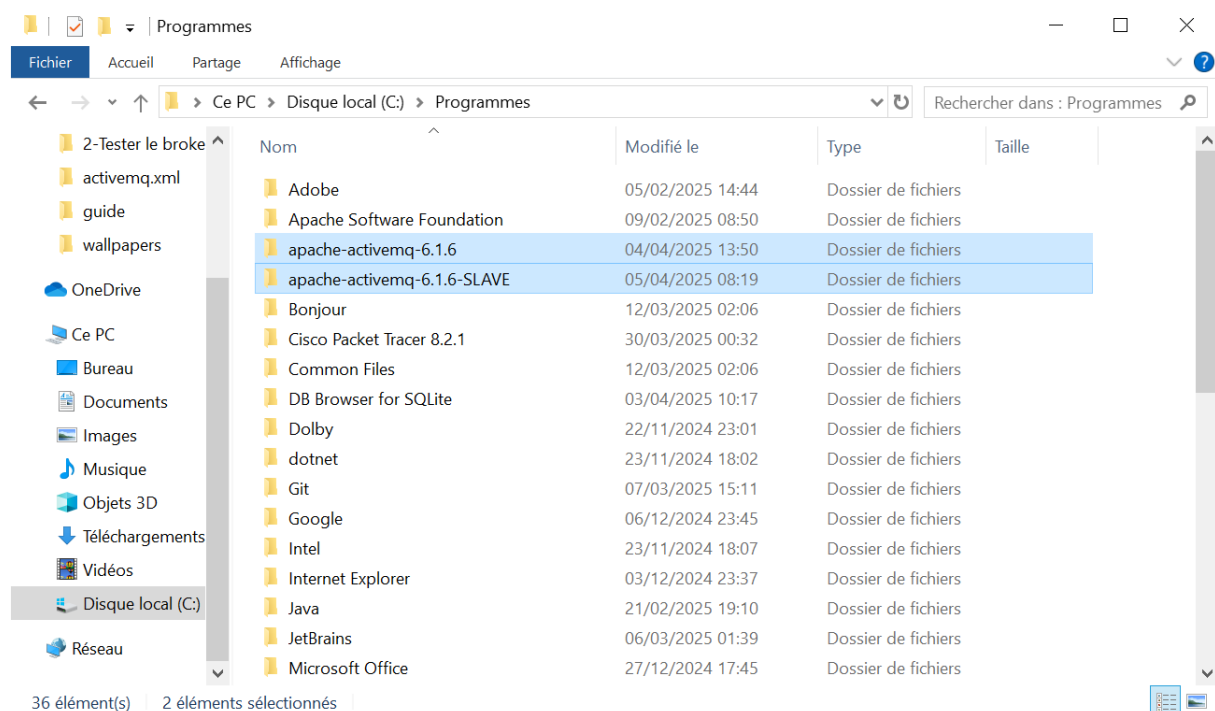


Figure 4.9: Two ActiveMQ broker folders: master and slave instances

Note on Alternative Setup: Instead of using two separate folders for the master and slave brokers, it is also possible to launch both instances from a single ActiveMQ installation. This involves duplicating the `activemq.xml` configuration file—assigning one to each broker—and carefully adjusting settings to avoid conflicts. For example, each broker must be assigned unique MQTT and web console ports to prevent binding errors. One option is to disable the Jetty web interface for the slave broker entirely, which allows it to function for failover purposes without a web console. Alternatively, a separate `jetty-slave.xml` file can be used, with modified ports, and explicitly imported in the slave’s configuration. In this project, the dual-folder approach was used to prevent such overlaps and simulate a more realistic, isolated broker deployment.

Step 3 – High Availability Configuration

To enable shared-storage high availability between two ActiveMQ brokers, several configuration adjustments are required. These changes ensure that both instances can coordinate over a common persistence layer while maintaining unique runtime characteristics to avoid conflicts.

Specifically, both brokers were configured to:

- Use unique broker names
- Share the same KahaDB directory for persistent storage
- Listen on different MQTT and web console ports

Critical Design Insights and Deployment Challenges

1. Broker Configuration and Lock Management

These configuration choices are critical for maintaining broker coordination and safe message persistence. Assigning distinct broker names avoids ambiguity, and using different ports ensures clients connect reliably. Most importantly, only one broker must write to the shared KahaDB at any time. Concurrent writes can corrupt the state or cause lock contention. ActiveMQ prevents this via a file-based lock mechanism (SharedFileLocker), allowing only one broker to access the store at once—making failover safe and deterministic.

2. Deployment Considerations and Testing

During testing, ensuring both brokers could access the shared KahaDB—especially across machines—posed challenges. Network-mounted storage (like NFS) added latency and lock inconsistencies. We used local disks for early tests and ensured synchronized system clocks between hosts to avoid timestamp issues during failover. These considerations were essential to achieving low-downtime, stable HA behavior.

The following steps outline each configuration in detail, with separate examples for the master and slave brokers. These distinctions are essential for enabling shared persistence, avoiding port collisions, and ensuring clean broker identification. Proper configuration also prevents runtime conflicts and lock contention on the shared KahaDB directory. Each setting contributes to a stable and predictable failover process. This structured breakdown facilitates troubleshooting and simplifies deployment in real-world HA scenarios.

Step 1 – Assigning a Unique Broker Name

Each broker instance in a high availability setup must have a unique name to distinguish it during coordination and failover. This is defined using the `brokerName` attribute inside the `<broker>` tag in the `activemq.xml` configuration file. Below are the respective configurations for the master and slave brokers:

1. Master Broker Configuration

The master broker is assigned the name `MasterBroker`, as shown in the following configuration snippet:

```
1 <broker xmlns="http://activemq.apache.org/schema/core"
2     brokerName="MasterBroker"
3     dataDirectory="${activemq.data}">
4     <!-- Additional configuration -->
5 </broker>
```

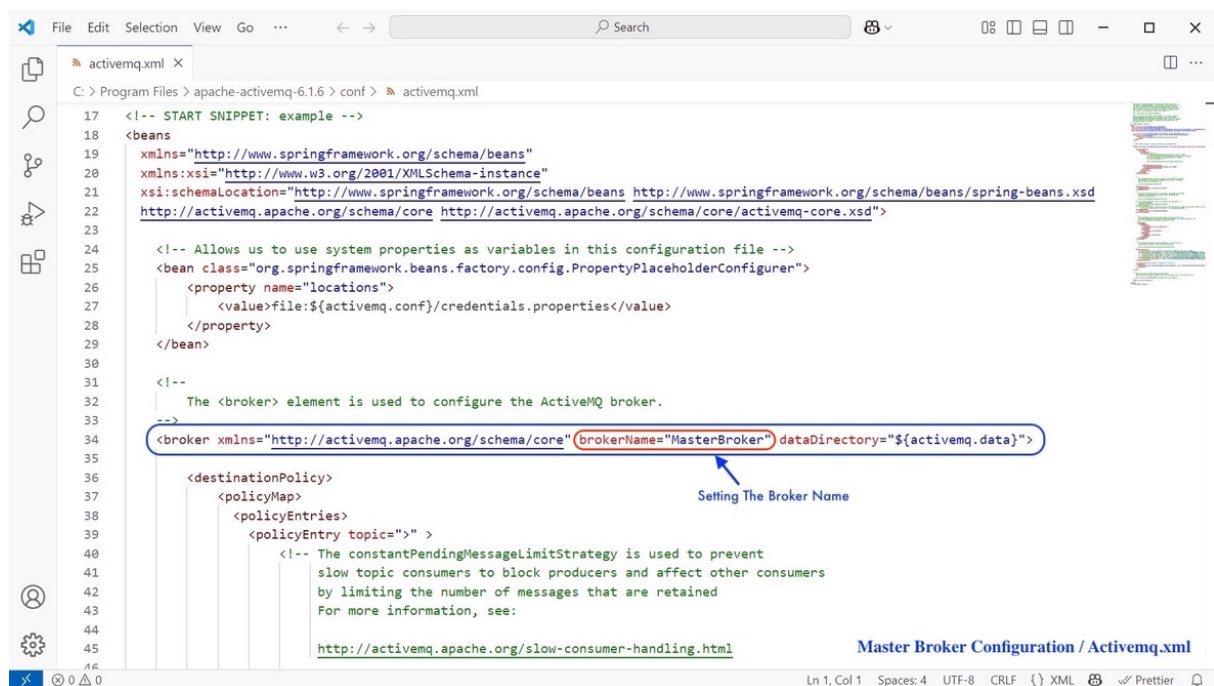


Figure 4.10: Master broker configured with unique name: `MasterBroker`

2. Slave Broker Configuration

Likewise, the slave broker is configured with a distinct name, `SlaveBroker`, to avoid conflicts and enable proper failover behavior.

```
1 <broker xmlns="http://activemq.apache.org/schema/core"
2     brokerName="SlaveBroker"
3     dataDirectory="${activemq.data}">
4     <!-- Additional configuration -->
5 </broker>
```

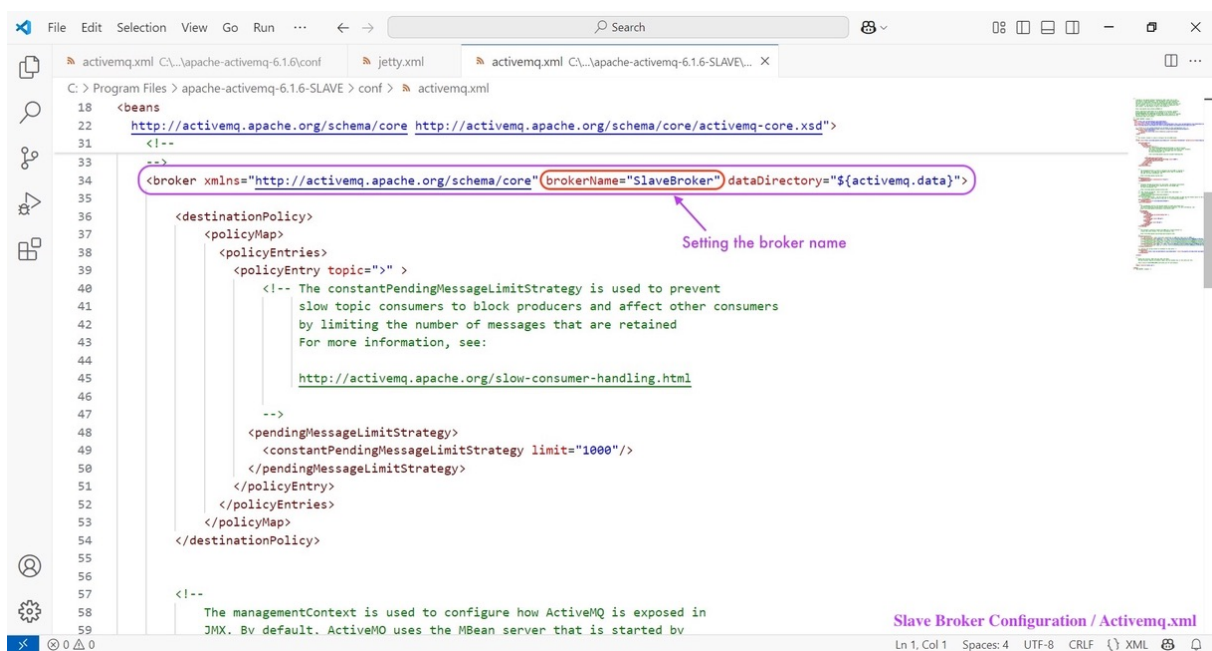


Figure 4.11: Slave broker assigned unique name: `SlaveBroker`

Step 2 – Configuring Shared KahaDB Directory

To enable persistence and failover coordination, both brokers must point to the same shared KahaDB directory. This allows the slave broker to take over message processing if the master becomes unavailable, while maintaining access to consistent message data and transaction state.

✚ The KahaDB storage path is defined inside the `<persistenceAdapter>` section of the `activemq.xml` configuration file. Both brokers should reference the **same absolute path**, whether on a shared disk or a network-mounted volume.

1. Master Broker Configuration

Below is the configuration snippet from the master broker's `activemq.xml` file:

```
1 <persistenceAdapter>
2   <kahaDB directory="C:/activemq-data/kahadb"/>
3 </persistenceAdapter>
```

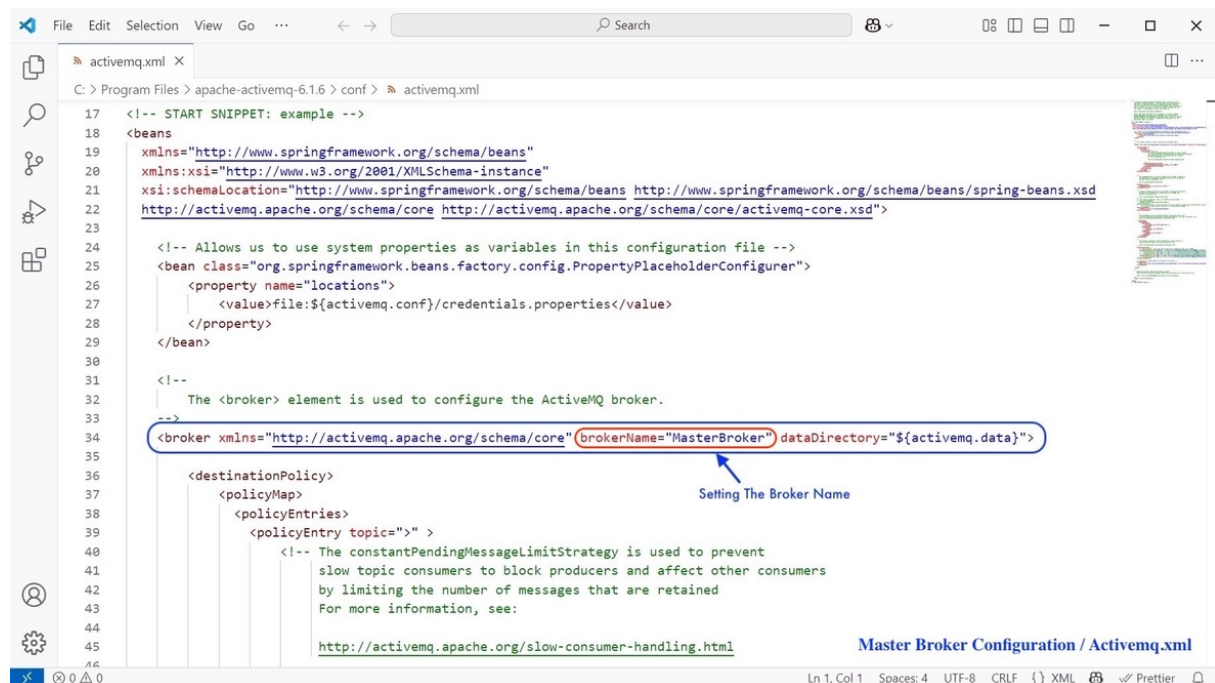


Figure 4.12: Master broker referencing shared KahaDB directory

2. Slave Broker Configuration

The slave broker uses the exact same KahaDB path, as shown below:

```

1 <persistenceAdapter>
2   <kahaDB directory="C:/activemq-data/kahadb"/>
3 </persistenceAdapter>

```

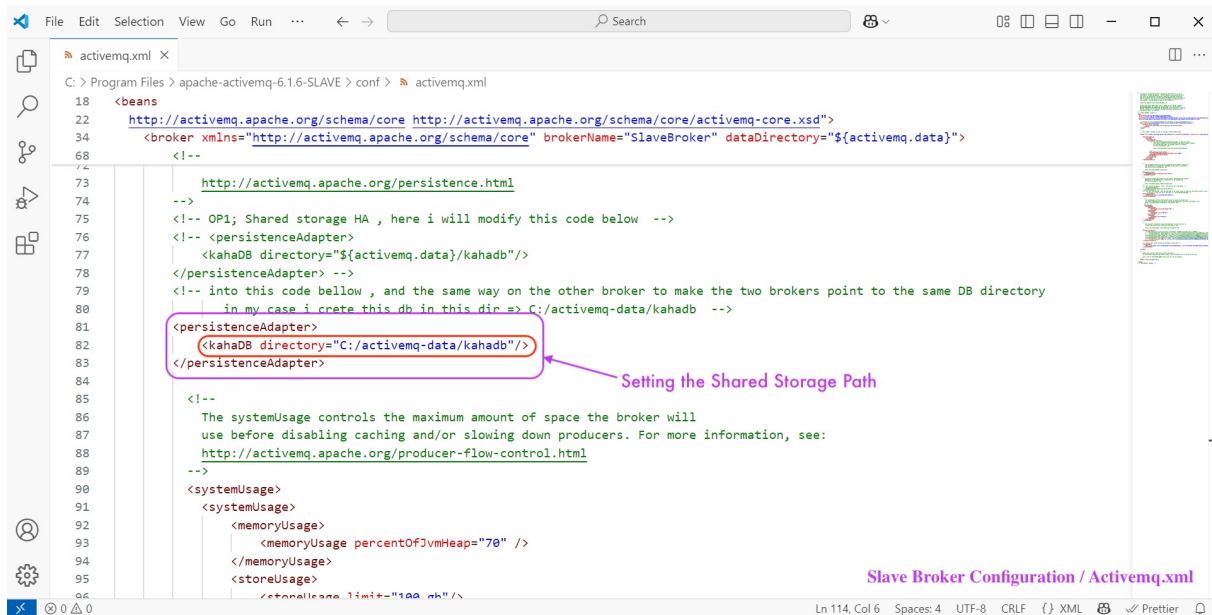


Figure 4.13: Slave broker using the same shared KahaDB path

✚ Important Note: Only one broker (typically the master) can actively write to the shared KahaDB at any time. If both brokers attempt to write simultaneously, a lock conflict will occur. This lock mechanism is what enforces failover behavior—ensuring only one broker is in control.

Step 3 – MQTT Transport Configuration

To prevent port conflicts and allow both brokers to run concurrently on the same machine, each broker must be assigned a unique port for the MQTT transport connector. This is done by modifying the `<transportConnector>` tag within the `activemq.xml` file.

By default, ActiveMQ uses port 1883 for MQTT. In this configuration:

- The master broker retains the default port 1883.
- The slave broker is configured to use port 1884.

1. Master Broker Configuration

```
1 <transportConnector name="mqtt" uri="mqtt://0.0.0.0:1883"/>
```

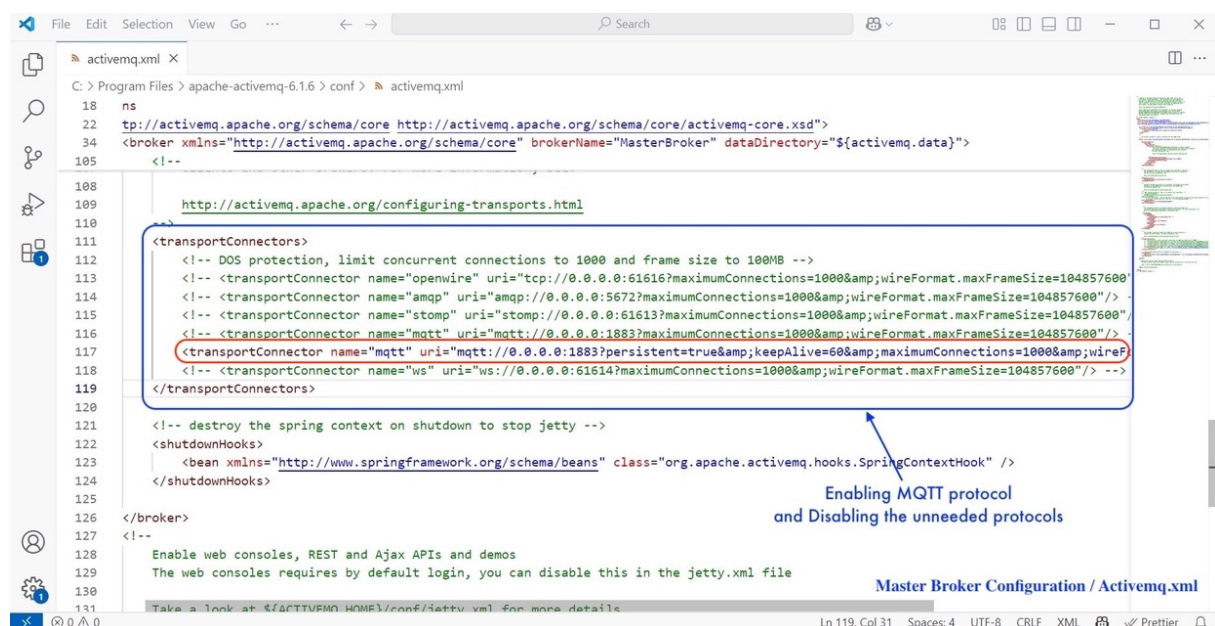


Figure 4.14: Master broker – MQTT configuration (port 1883)

2. Slave Broker Configuration

```
1 <transportConnector name="mqtt" uri="mqtt://0.0.0.0:1884"/>
```

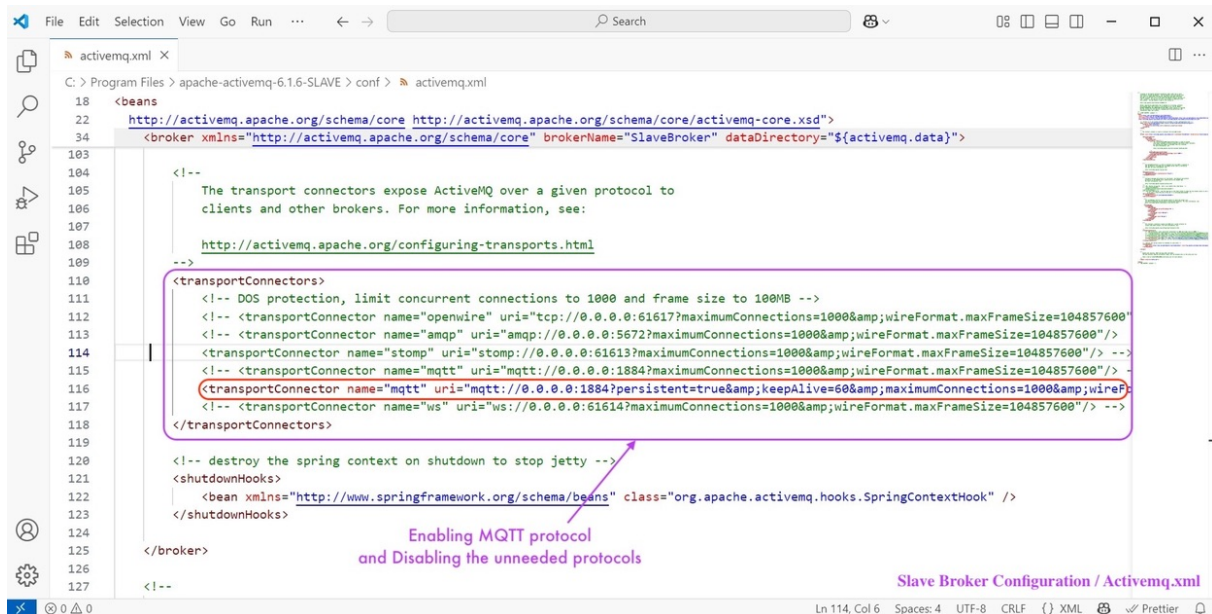


Figure 4.15: Slave broker – MQTT configuration (port 1884)

Note: Additional transport connectors (such as AMQP, STOMP, or OpenWire) were disabled in both brokers to reduce surface complexity and focus exclusively on MQTT traffic for the purposes of this deployment.

Step 4 – Web Console Port Configuration

ActiveMQ includes a built-in web interface (powered by Jetty) for broker monitoring and management. To avoid port conflicts between the two brokers, each instance must expose the web console on a different HTTP port.

1. Master Broker Configuration

The master broker retains the default Jetty port configuration provided by ActiveMQ. Since no changes are required, the web console remains accessible at the default address without modification.

- **Master Web Console URL:** `http://localhost:8161/admin`

2. Slave Broker Configuration

To avoid conflict, the slave broker's `jetty.xml` file was modified to use port 8162 instead of the default 8161. This ensures that both web interfaces can run simultaneously on the same host.

- **Slave Web Console URL:** `http://localhost:8162/admin`

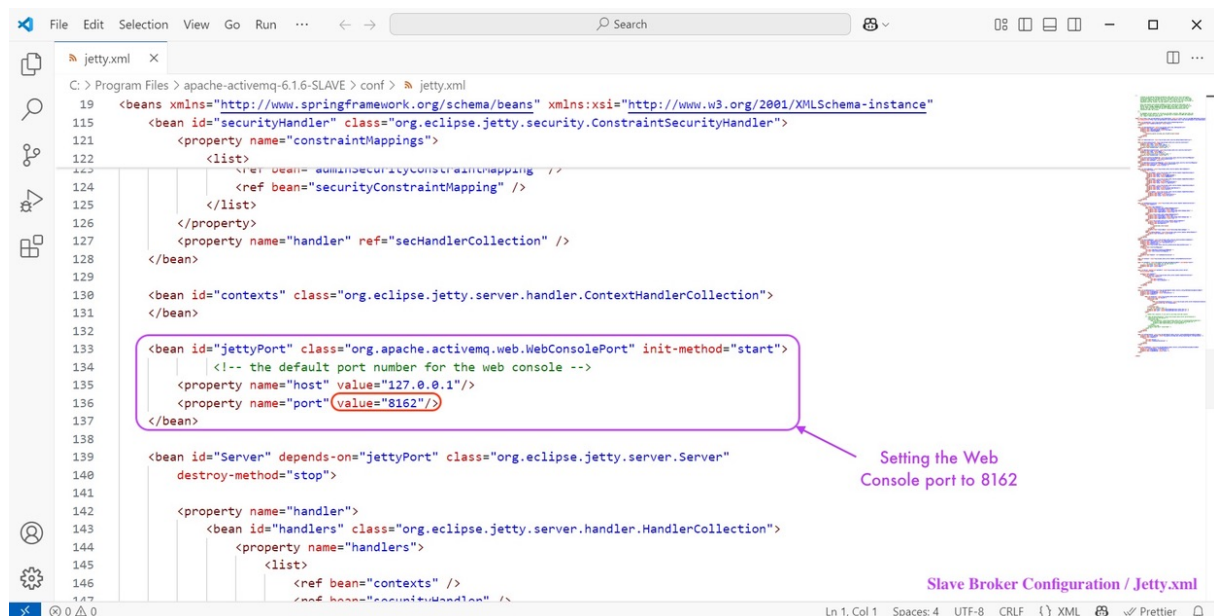


Figure 4.16: Slave broker Jetty port changed to 8162

Tip: In a dual-folder setup like this one, each broker has its own `conf` directory. You can directly modify the `jetty.xml` file for the slave broker to assign a new port, without needing to rename or duplicate the file.

Step 6 – Lock Coordination Validation

To verify proper high-availability behavior, the brokers were launched sequentially to confirm lock acquisition on the shared KahaDB storage.

1. The master broker successfully acquired the lock and became active.
2. The slave broker detected the existing lock and entered standby mode.

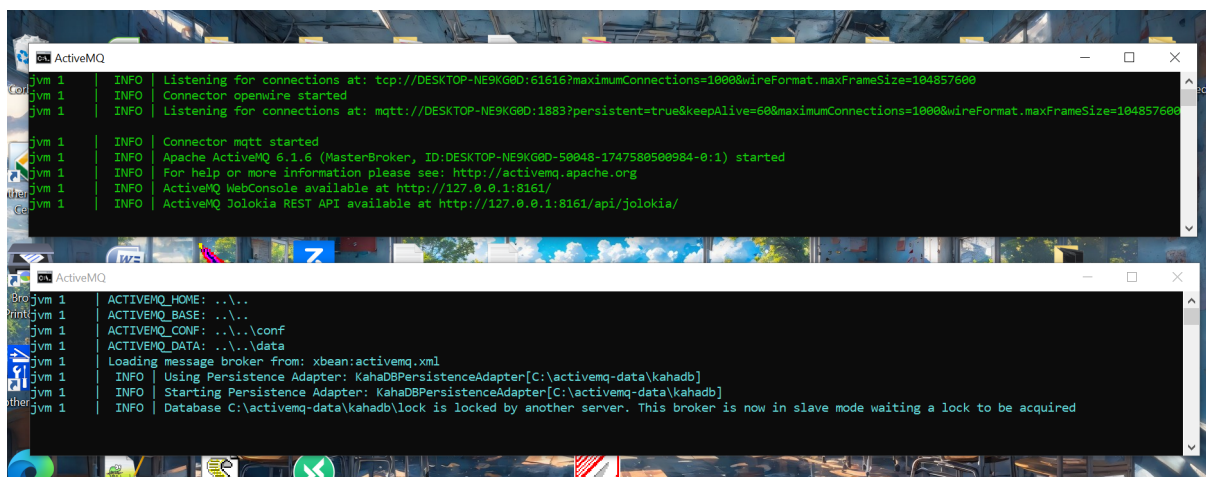


Figure 4.17: Master broker running (active); slave broker waiting (standby)

Note: No clients were connected at this stage. This step serves only to validate that the shared storage and lock coordination mechanism operate correctly before introducing MQTT traffic.

4.3.3 Publisher and Subscriber Client Interfaces

This subsection presents the MQTT clients developed to simulate communication between wearable health sensors and a hospital monitoring dashboard.

Both clients were designed to function reliably with the high-availability broker architecture. They support automatic failover handling—detecting disconnections and reconnecting to the standby broker without user input—ensuring continuity of data transmission and visibility.

This behavior enables realistic simulation of clinical workflows, where real-time monitoring must persist despite backend failures. It also demonstrates the system’s ability to maintain QoS guarantees under failure conditions.

1. Subscriber Client – Hospital Monitoring Dashboard

The subscriber represents the hospital dashboard. It is a Flask web application using Flask-SocketIO to display incoming MQTT messages in real time. The interface supports automatic reconnection and live topic monitoring.

1. Launching the subscriber from the PyCharm IDE:

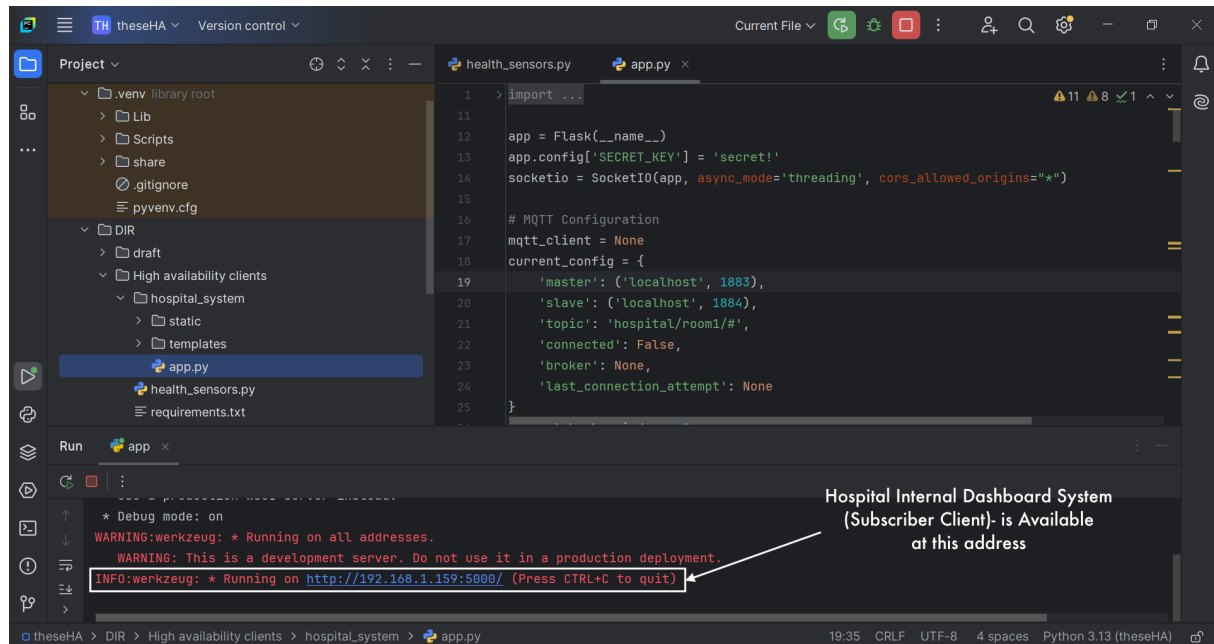


Figure 4.18: Subscriber client launched from the development environment

2. Initial layout of the dashboard GUI before connection:

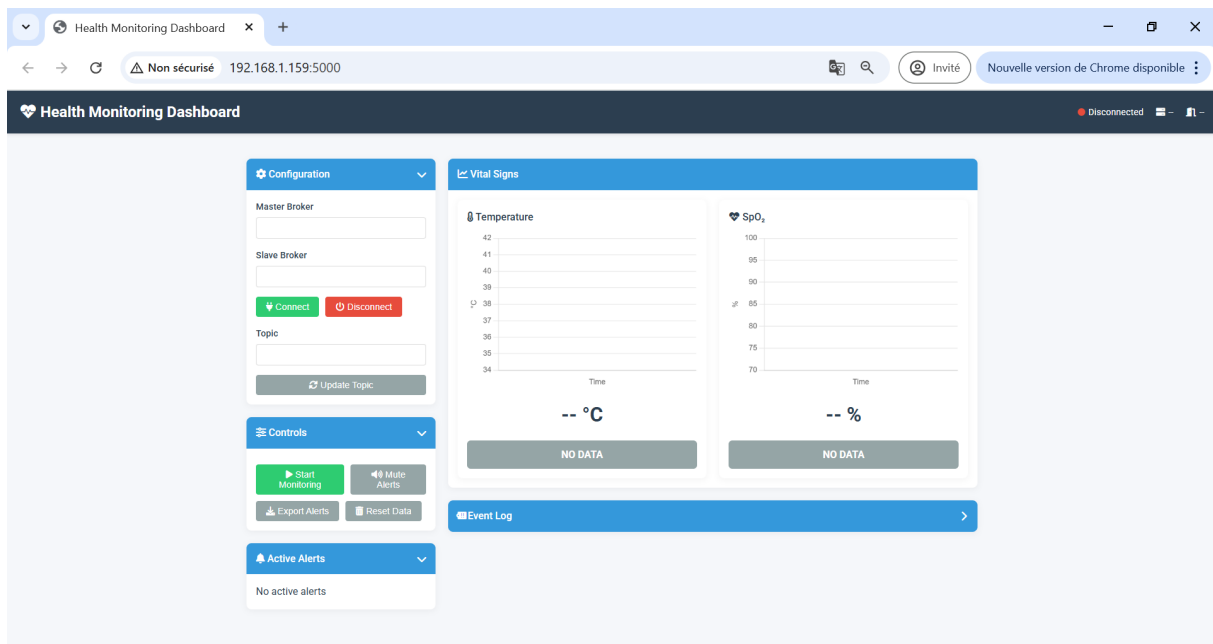


Figure 4.19: Subscriber GUI layout with URI fields and controls before connecting

3. Client configured with URIs of master and slave brokers:

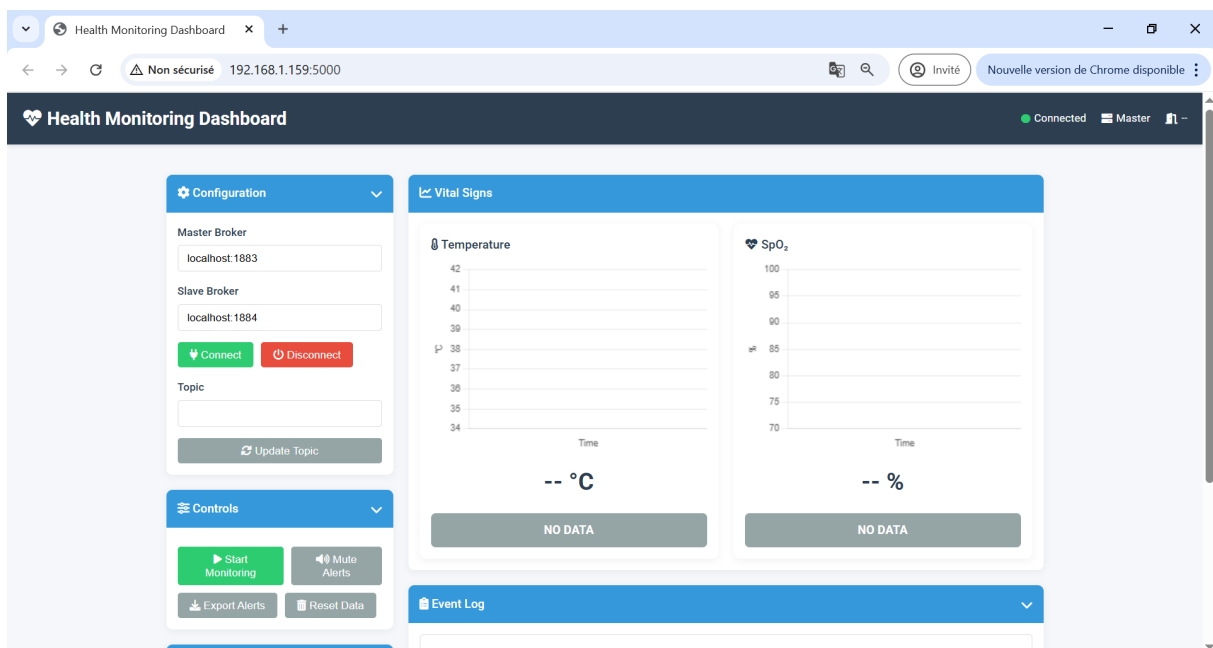


Figure 4.20: Entering the broker addresses for high availability support

4. Successful connection established with the master broker:

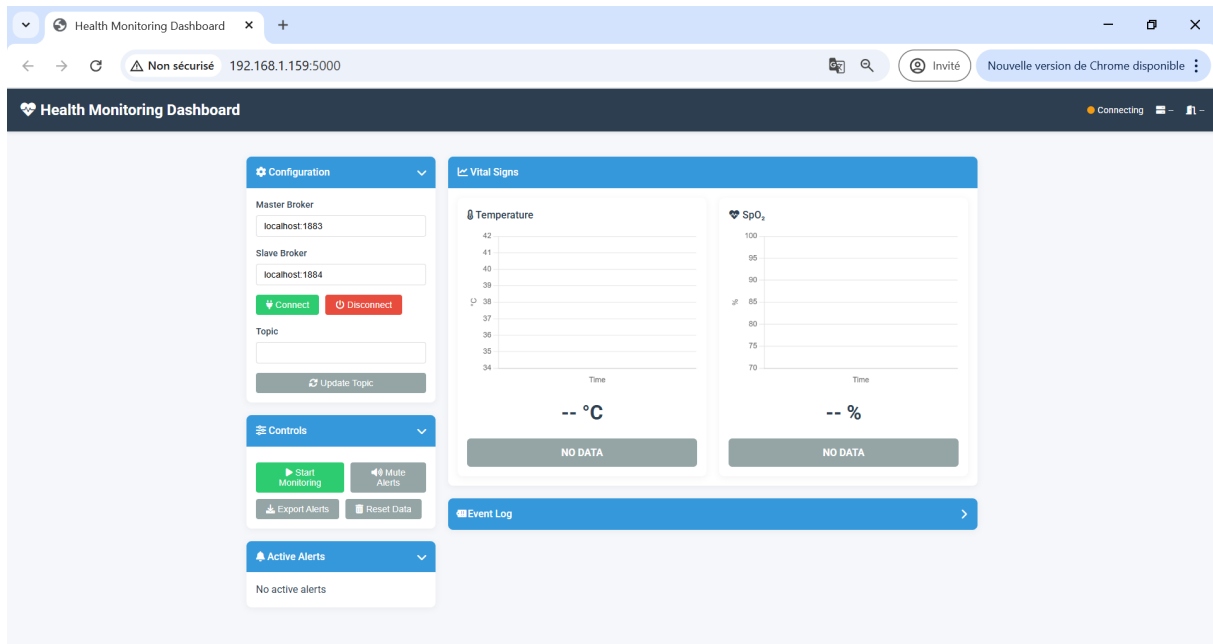


Figure 4.21: Subscriber connected successfully to the master broker (port 1883)

5. Topic entered and live monitoring is ready to start:

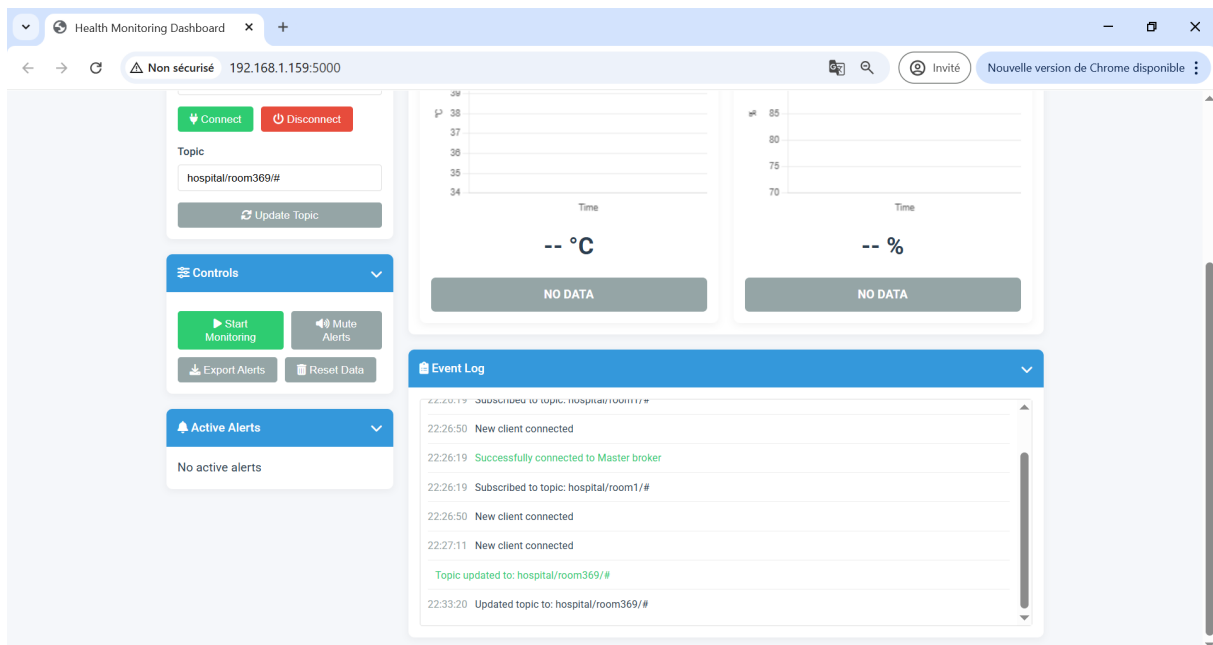


Figure 4.22: User enters topic name and is ready to monitor sensors data

2. Publisher Client – Wearable Sensor Simulator

The publisher, built with Python (Tkinter + Paho MQTT), simulates a wearable device that sends periodic sensor data and supports manual alert triggering.

1. Launching the publisher client from PyCharm:

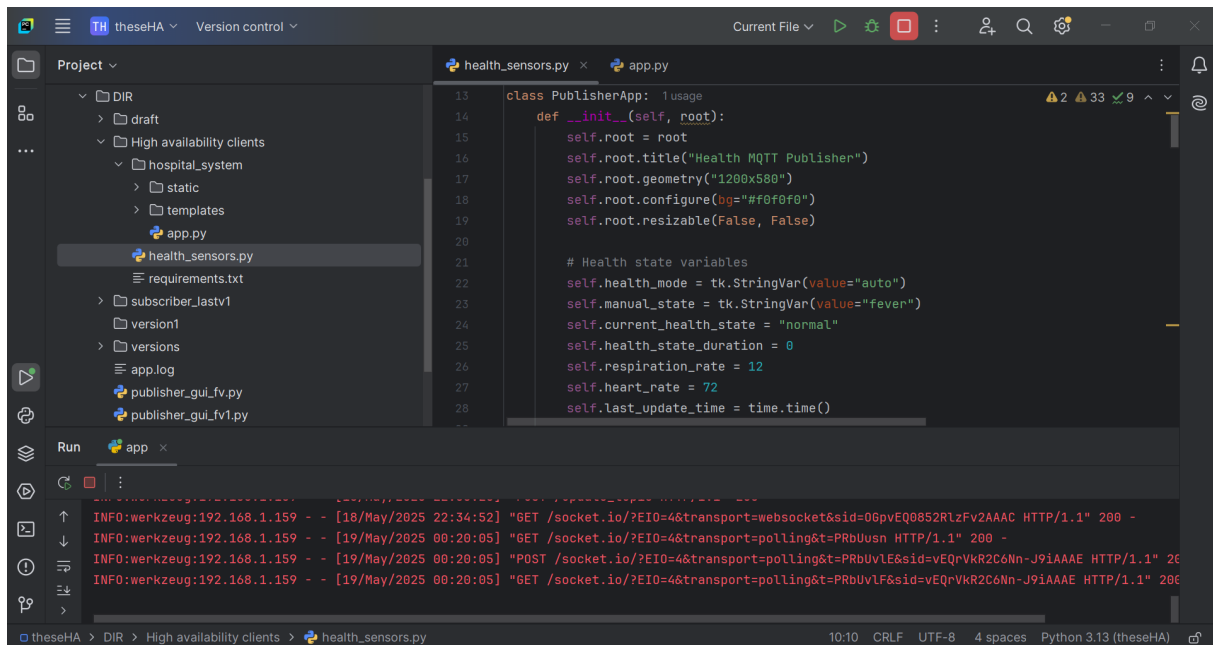


Figure 4.23: Publisher client launched for sending health data to the broker

2. Entering the broker URI and connecting successfully:

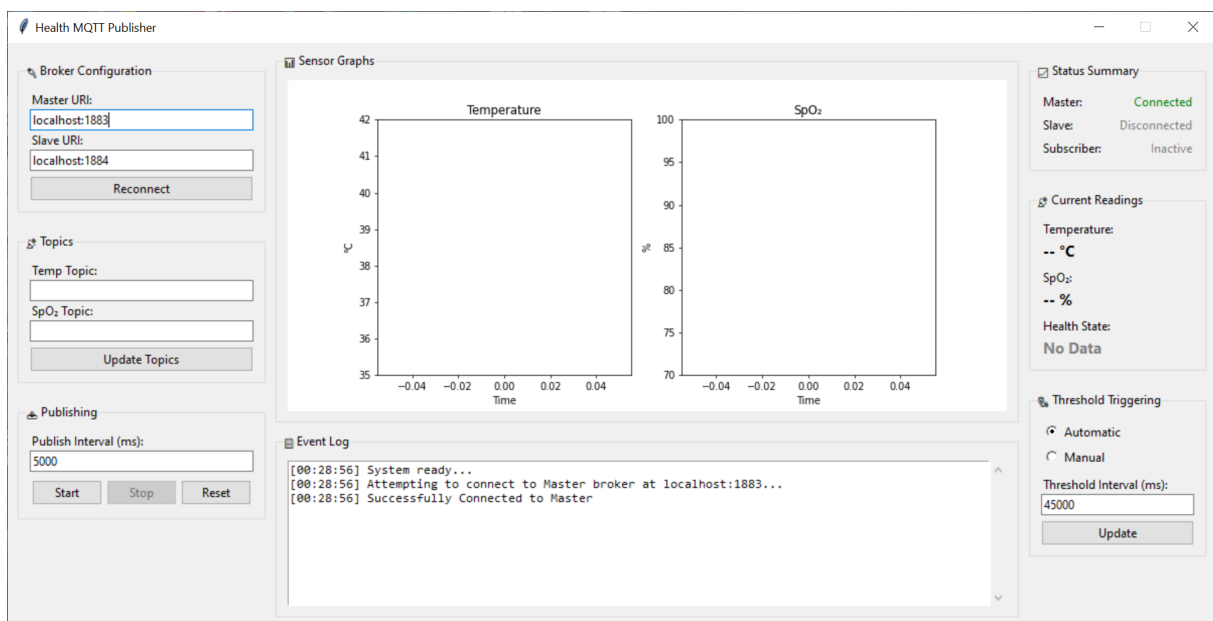


Figure 4.24: Publisher connects to the master broker (MQTT port 1883)

3. Adjusting publishing interval for dynamic testing:

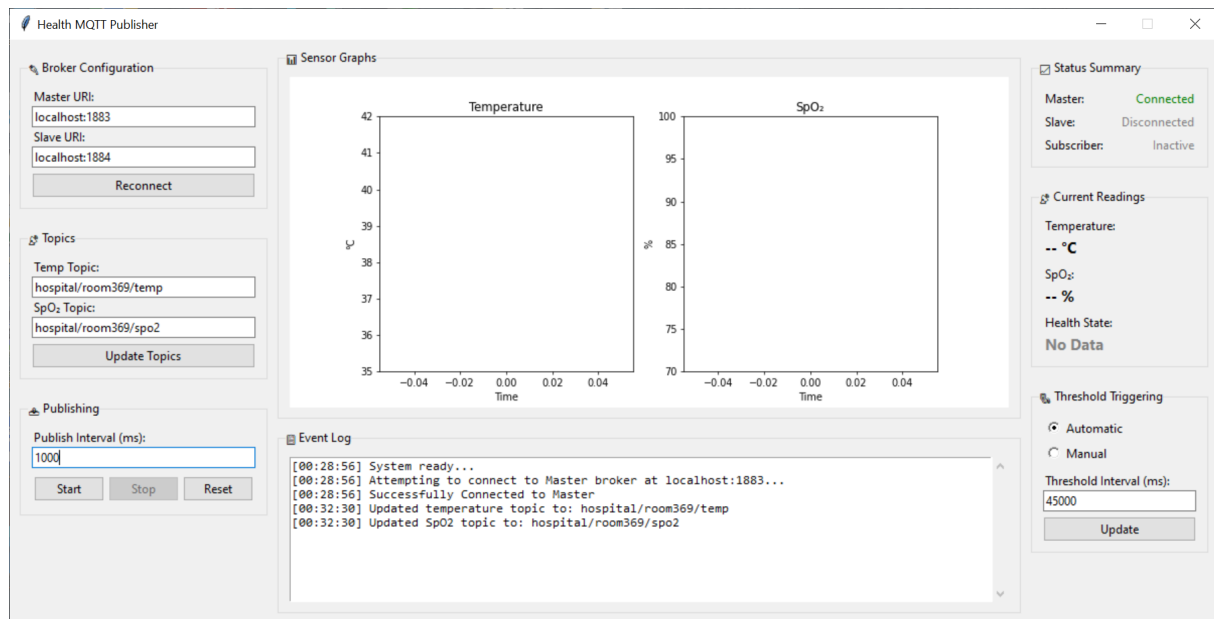


Figure 4.25: User modifies the publishing rate to simulate real sensor behavior

4. Manual Emergency Alert Trigger Configuration:

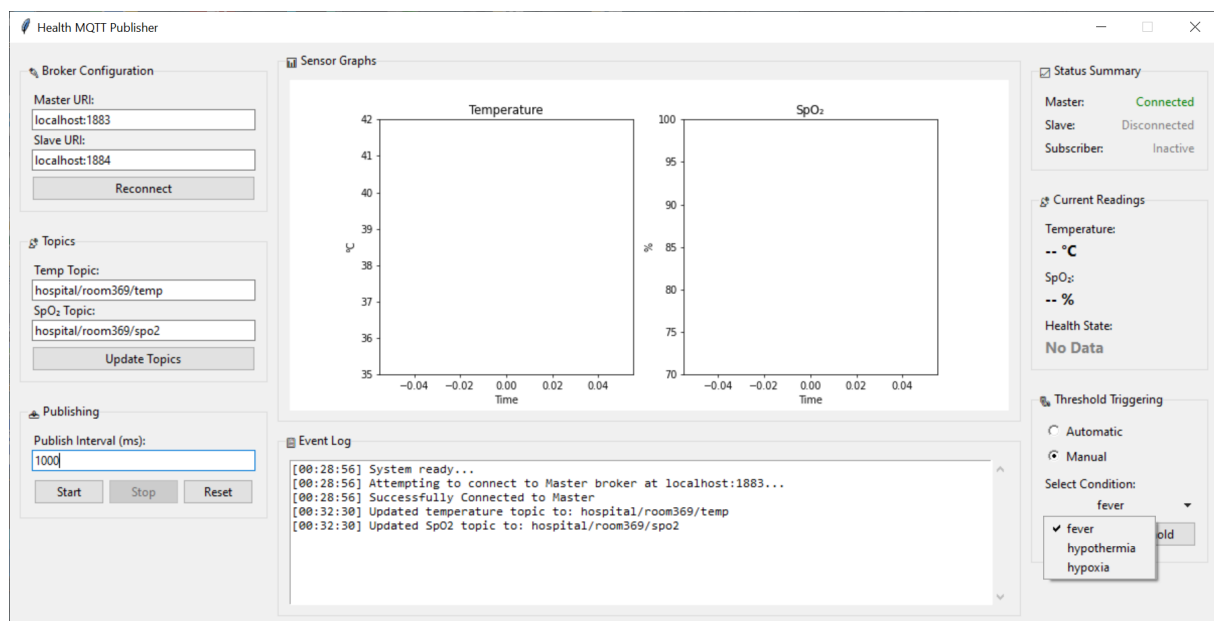


Figure 4.26: Showing manual configuration options for triggering an emergency alert (

5. Publisher starts sending sensor data continuously:

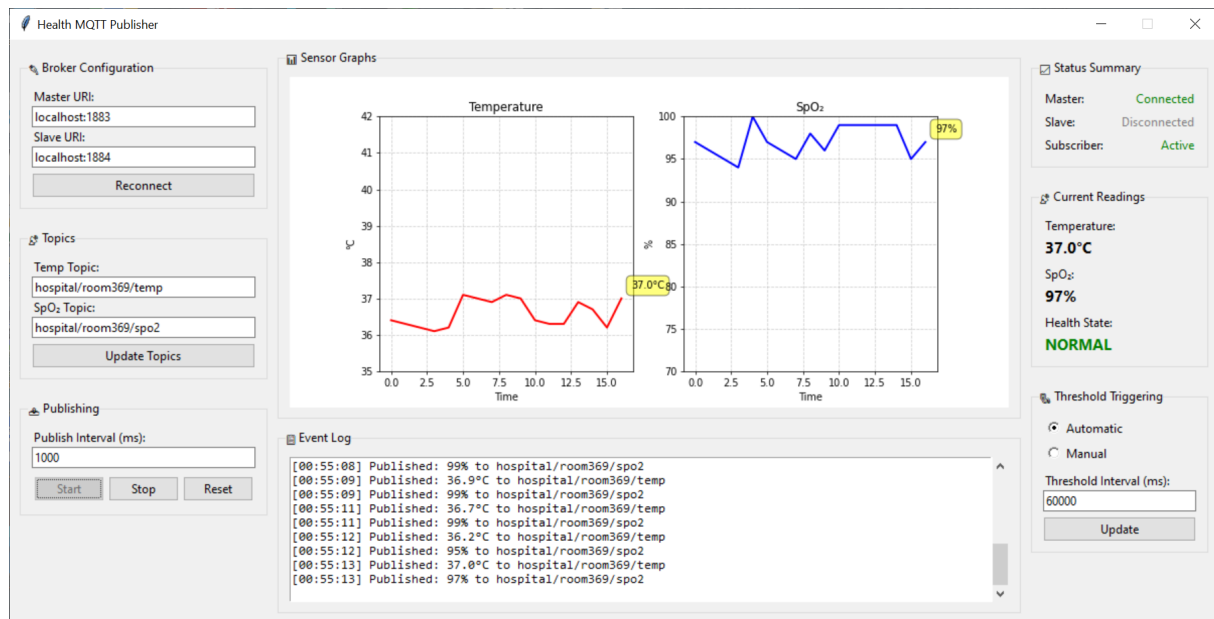


Figure 4.27: Live publishing of sensor data confirmed by client interface

These interfaces are now connected to the HA broker system and actively simulating the full MQTT data flow. The next section demonstrates their behavior during failover.

4.3.4 Client Interaction Mechanism

To ensure seamless MQTT communication and resilience during broker failover, both the publisher and subscriber clients were designed with autonomous switching logic. This subsection presents the internal mechanisms of each client using structured algorithm blocks.

1. Publisher Client Algorithm – Wearable Health Simulator

The publisher simulates a wearable health sensors, generating data and sending it periodically to the MQTT broker. It supports automatic reconnection, offline message queuing, and configurable health simulation.

Algorithm 1 Publisher Client Behavior

```

1: Initialize GUI, sensor simulation, and MQTT client
2: Set broker URIs: Master and Slave
3: Attempt connection to current broker
4: if connection fails then
5:     Switch to alternate broker
6: end if
7: while publishing is enabled do
8:     Generate synthetic temperature and SpO2 readings
9:     Display values on GUI
10:    if connected to broker then
11:        Publish messages to respective topics with QoS 1
12:        if queued messages exist then
13:            Resend queued messages
14:        end if
15:    else
16:        Queue current messages
17:        Attempt failover to alternate broker
18:    end if
19:    if auto health mode enabled then
20:        Randomly trigger a health threshold event
21:    else if manual mode is enabled then
22:        Apply selected health condition from GUI
23:    end if
24:    Wait for defined publishing interval
25: end while

```

2. Subscriber Client Algorithm – Hospital Dashboard

The subscriber component simulates a hospital dashboard and is implemented as a Flask web application enhanced with Flask-SocketIO for real-time communication. It consists of two primary layers: a backend MQTT listener and a frontend WebSocket-connected dashboard.

Received MQTT messages are parsed and forwarded to the frontend via SocketIO WebSockets. This ensures continuous, low-latency updates on the dashboard without page reloads or polling. The frontend dynamically updates visual charts, logs, and metric indicators as new data arrives.

Algorithm 2 Subscriber Client Behavior

```

1: Launch Flask + SocketIO web server
2: Initialize MQTT client with broker URIs
3: Attempt connection to current broker
4: if connection fails then
5:     Switch to alternate broker
6: end if
7: Subscribe to specified topic (e.g., hospital/room1/#)
8: while subscriber is running do
9:     if new MQTT message received then
10:         Parse topic and payload
11:         Update dashboard (graphs, metrics, logs)
12:         Check thresholds for temperature and SpO2
13:         if alert condition met then
14:             Trigger visual/audio alert and log event
15:         end if
16:     end if
17:     if broker disconnects then
18:         Switch to alternate broker and reconnect
19:     end if
20: end while

```

4.3.5 Failover Execution and Results

This subsection presents the end-to-end test of the ActiveMQ high availability setup under simulated failure conditions. The goal is to demonstrate:

1. Normal system behavior under active broker
2. Master broker failure and slave promotion
3. MQTT clients auto-reconnecting and resuming communication
4. No loss of alerts or sensor messages

1. Normal Operation

During normal operation, the publisher continuously sends simulated temperature data to the master broker. The subscriber receives the messages in real time and updates the dashboard accordingly.

1. Figure below shows the publisher actively sending MQTT messages to the master broker:

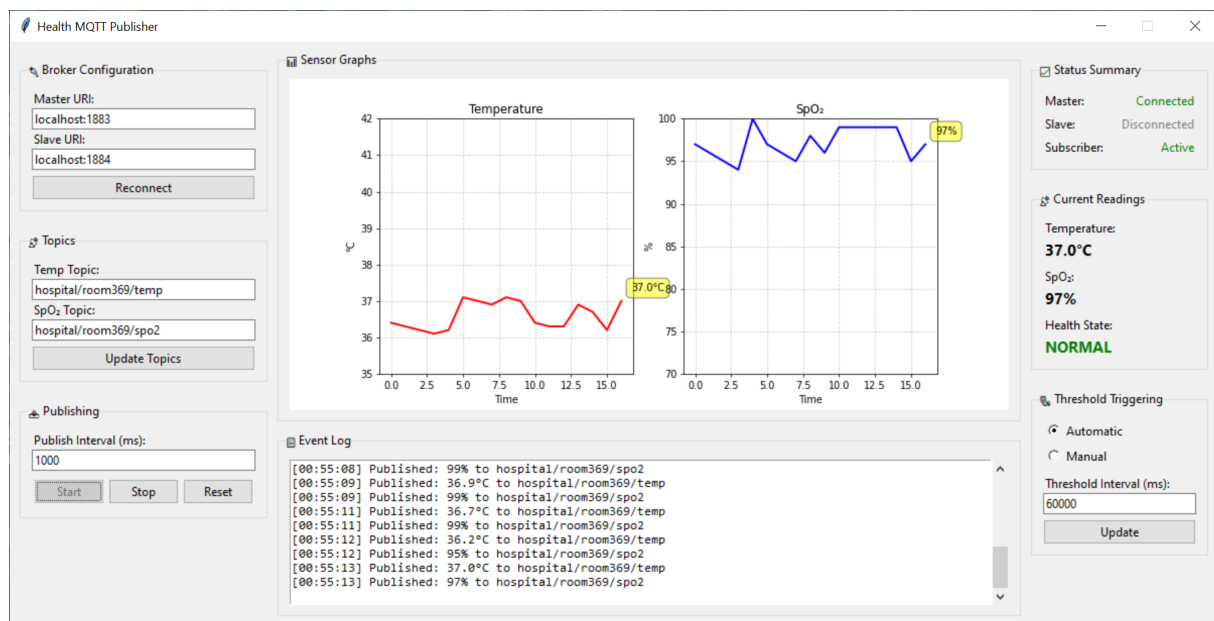


Figure 4.28: Publisher connected to the master broker and sending periodic data

2. The subscriber dashboard below is successfully receiving and displaying the data:

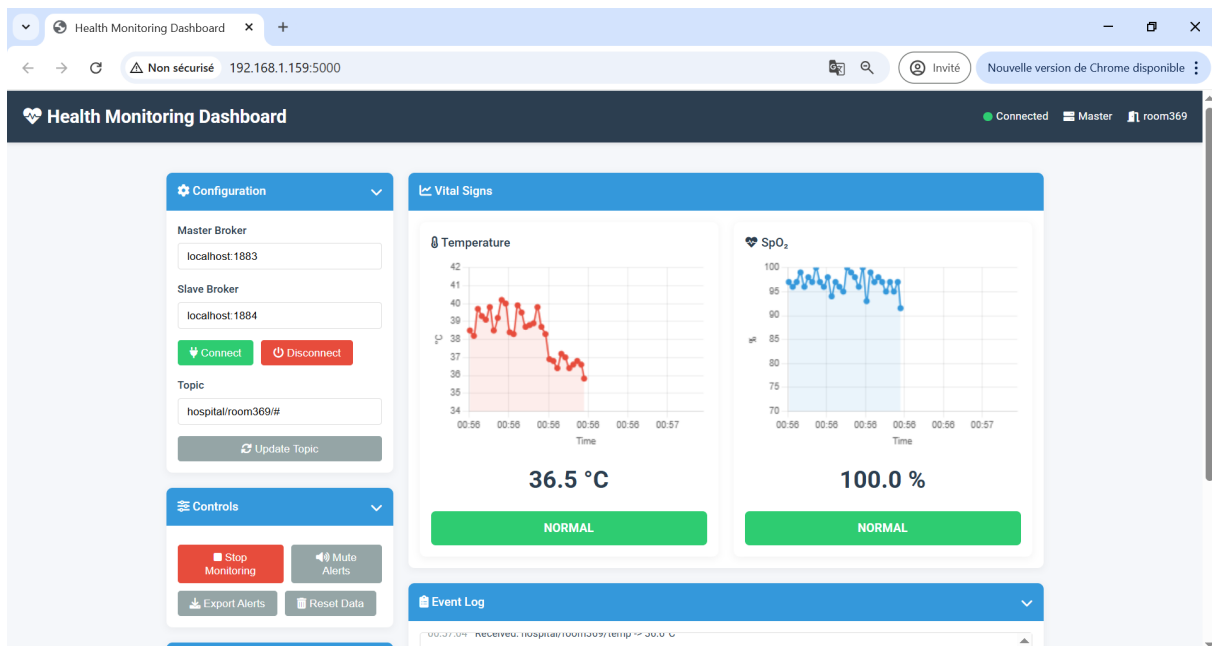


Figure 4.29: Subscriber connected to master broker, showing incoming messages

3. The subscriber log also confirms continuous reception of MQTT messages:

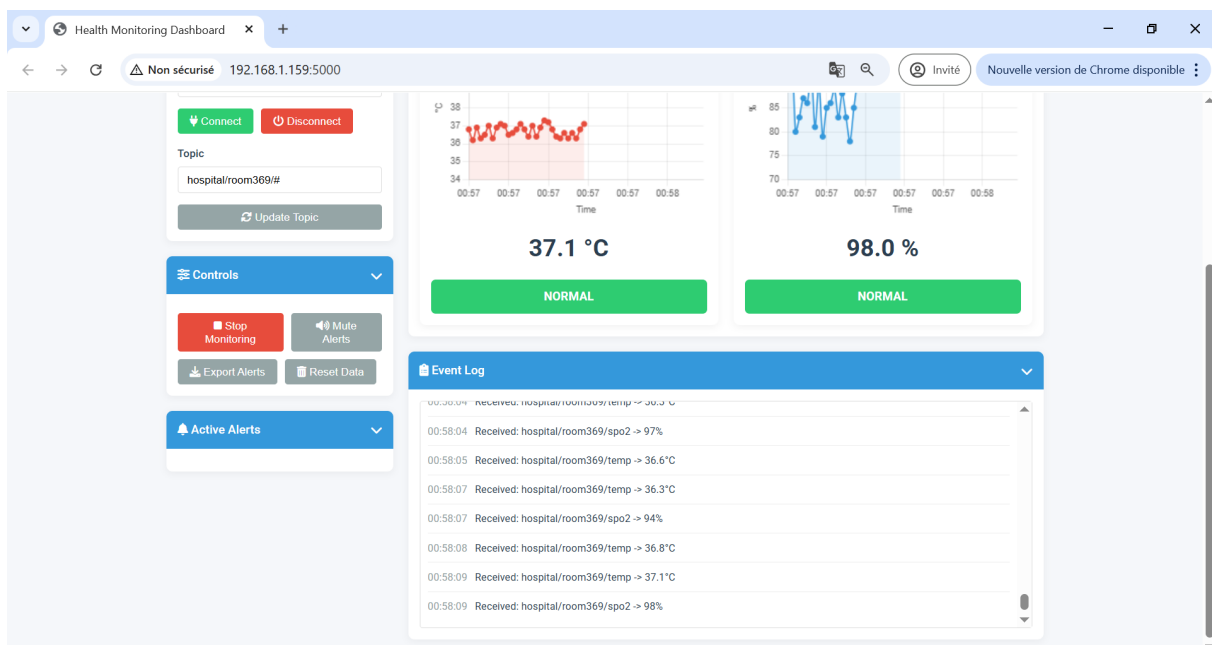
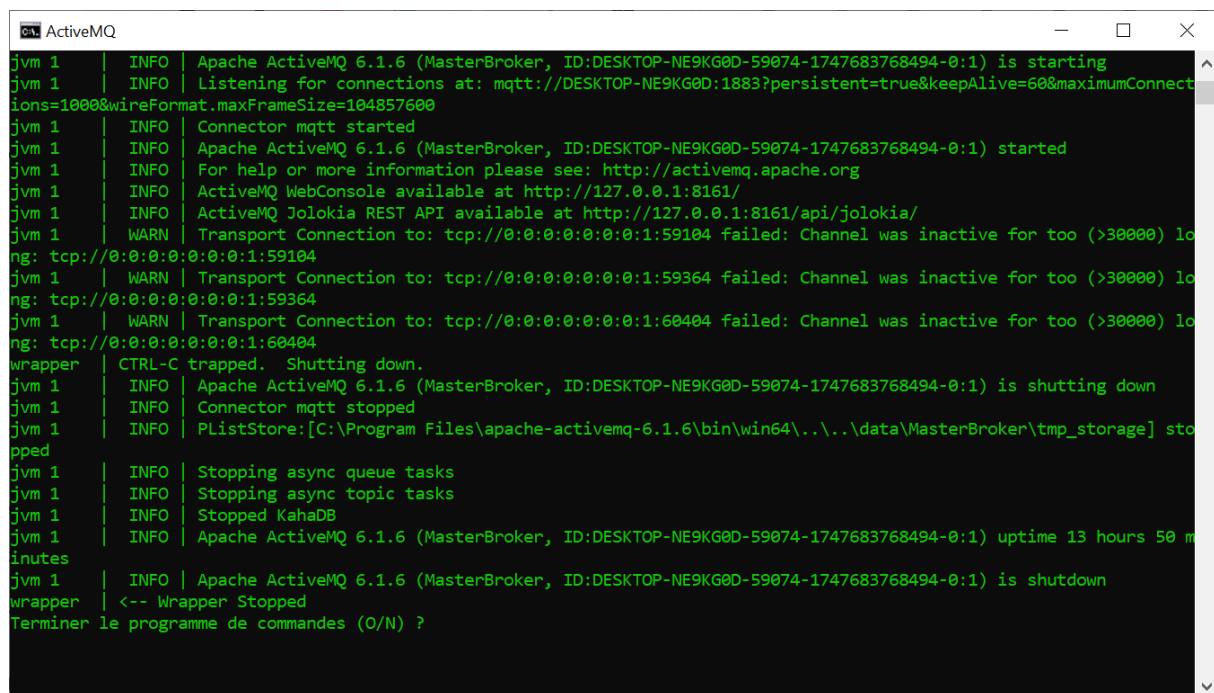


Figure 4.30: Subscriber log showing consistent message flow from the broker

2. Simulated Broker Failure

To validate failover behavior, the master broker is manually stopped while clients are still connected and actively transmitting data.

1. The figure below shows the master broker being shut down:



```

ActiveMQ
jvm 1 | INFO | Apache ActiveMQ 6.1.6 (MasterBroker, ID:DESKTOP-NE9KG0D-59074-1747683768494-0:1) is starting
jvm 1 | INFO | Listening for connections at: mqtt://DESKTOP-NE9KG0D:1883?persistent=true&keepAlive=60&maximumConnections=1000&wireFormat.maxFrameSize=104857600
jvm 1 | INFO | Connector mqtt started
jvm 1 | INFO | Apache ActiveMQ 6.1.6 (MasterBroker, ID:DESKTOP-NE9KG0D-59074-1747683768494-0:1) started
jvm 1 | INFO | For help or more information please see: http://activemq.apache.org
jvm 1 | INFO | ActiveMQ WebConsole available at http://127.0.0.1:8161/
jvm 1 | INFO | ActiveMQ Jolokia REST API available at http://127.0.0.1:8161/api/jolokia/
jvm 1 | WARN | Transport Connection to: tcp://0:0:0:0:0:0:1:59104 failed: Channel was inactive for too (>30000) lo
ng: tcp://0:0:0:0:0:0:1:59104
jvm 1 | WARN | Transport Connection to: tcp://0:0:0:0:0:0:1:59364 failed: Channel was inactive for too (>30000) lo
ng: tcp://0:0:0:0:0:0:1:59364
jvm 1 | WARN | Transport Connection to: tcp://0:0:0:0:0:0:1:60404 failed: Channel was inactive for too (>30000) lo
ng: tcp://0:0:0:0:0:0:1:60404
wrapper | CTRL-C trapped. Shutting down.
jvm 1 | INFO | Apache ActiveMQ 6.1.6 (MasterBroker, ID:DESKTOP-NE9KG0D-59074-1747683768494-0:1) is shutting down
jvm 1 | INFO | Connector mqtt stopped
jvm 1 | INFO | PListStore[C:\Program Files\apache-activemq-6.1.6\bin\win64\...\data\MasterBroker\tmp_storage] sto
pped
jvm 1 | INFO | Stopping async queue tasks
jvm 1 | INFO | Stopping async topic tasks
jvm 1 | INFO | Stopped KahaDB
jvm 1 | INFO | Apache ActiveMQ 6.1.6 (MasterBroker, ID:DESKTOP-NE9KG0D-59074-1747683768494-0:1) uptime 13 hours 50 m
inutes
jvm 1 | INFO | Apache ActiveMQ 6.1.6 (MasterBroker, ID:DESKTOP-NE9KG0D-59074-1747683768494-0:1) is shutdown
wrapper | <-- Wrapper Stopped
Terminer le programme de commandes (O/N) ?
  
```

Figure 4.31: Simulated failure by stopping the master broker manually

3. Slave Activation and Client Reconnection

After the master goes offline, the slave broker detects the released lock and transitions into active mode. The MQTT clients automatically detect disconnection and reconnect to the slave. This seamless transition is made possible by the shared KahaDB mechanism, which ensures that only one broker can hold the persistence lock at a time. Once the master broker shuts down or fails, the lock is released, allowing the slave to take control almost immediately.

From the client's perspective, the reconnection process is transparent. Assuming the clients were configured with a proper failover URI, they automatically reconnect to the newly active broker without requiring any manual intervention. This behavior is essential in real-time systems, where message flow must resume with minimal disruption after failover events.

1. The following figure shows the slave broker successfully acquiring the shared lock and becoming active:

```

ActiveMQ
jvm 1 | [...]lib,...\lib\camel,...\lib\optional,...\lib\web,...\lib\extra]
jvm 1 | ACTIVEMQ_HOME: ..\..
jvm 1 | ACTIVEMQ_BASE: ..\..
jvm 1 | ACTIVEMQ_CONF: ..\..\conf
jvm 1 | ACTIVEMQ_DATA: ..\..\data
jvm 1 | Loading message broker from: xbean:activemq.xml
jvm 1 | INFO | Using Persistence Adapter: KahaDBPersistenceAdapter[C:\activemq-data\kahadb]
jvm 1 | INFO | Starting Persistence Adapter: KahaDBPersistenceAdapter[C:\activemq-data\kahadb]
jvm 1 | INFO | Database C:\activemq-data\kahadb\lock is locked by another server. This broker is now in slave mode
jvm 1 | waiting a lock to be acquired
jvm 1 | INFO | Starting KahaDBStore
jvm 1 | INFO | Opening MessageDatabase
jvm 1 | INFO | KahaDB is version 7
jvm 1 | INFO | Starting Temp Data Store
jvm 1 | INFO | PListStore[C:\Program Files\apache-activemq-6.1.6-SLAVE\bin\win64\...\data\SlaveBroker\tmp_stora
ge] started
jvm 1 | INFO | Starting Job Scheduler Store
jvm 1 | INFO | Persistence Adapter successfully started
jvm 1 | INFO | Apache ActiveMQ 6.1.6 (SlaveBroker, ID:DESKTOP-NE9KG0D-60740-1747733583179-0:1) is starting
jvm 1 | INFO | Listening for connections at: mqtt://DESKTOP-NE9KG0D:1884?persistent=true&keepAlive=60&maximumConne
ctions=1000&wireFormat.maxFrameSize=104857600
jvm 1 | INFO | Connector mqtt started
jvm 1 | INFO | Apache ActiveMQ 6.1.6 (SlaveBroker, ID:DESKTOP-NE9KG0D-60740-1747733583179-0:1) started
jvm 1 | INFO | For help or more information please see: http://activemq.apache.org
jvm 1 | INFO | ActiveMQ WebConsole available at http://127.0.0.1:8162/
jvm 1 | INFO | ActiveMQ Jolokia REST API available at http://127.0.0.1:8162/api/jolokia/

```

Figure 4.32: Slave broker now active and in control after master broker shutdown

2. The publisher client detects the failover and reconnects to the slave:

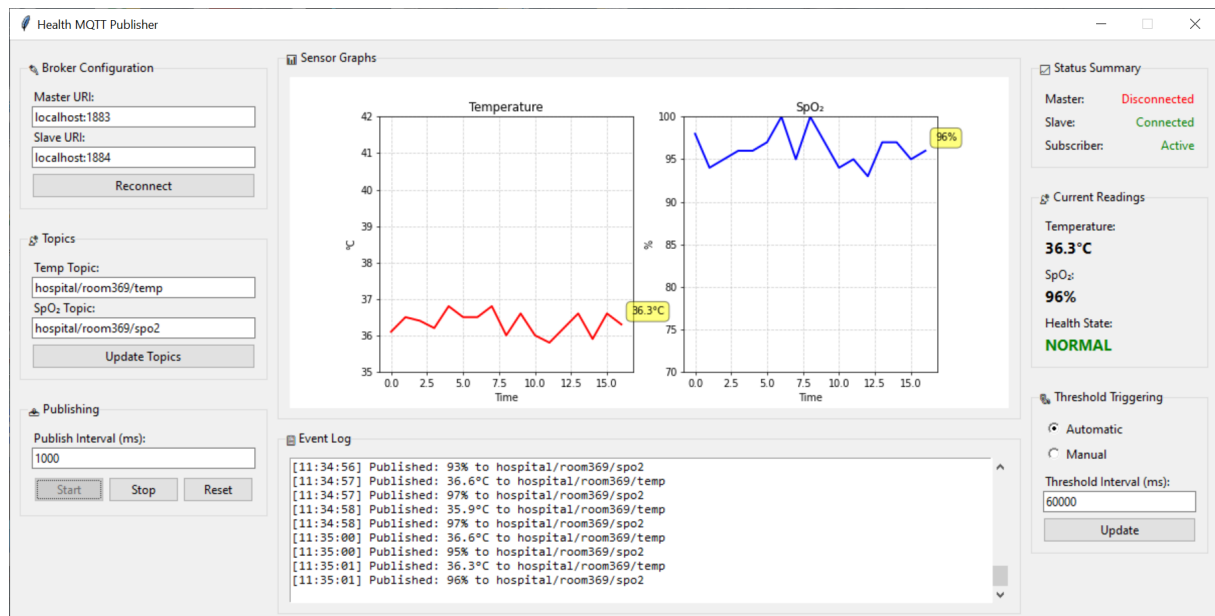


Figure 4.33: Publisher reconnected to the slave and resumes data transmission

3. The subscriber client also reconnects and resumes receiving data:

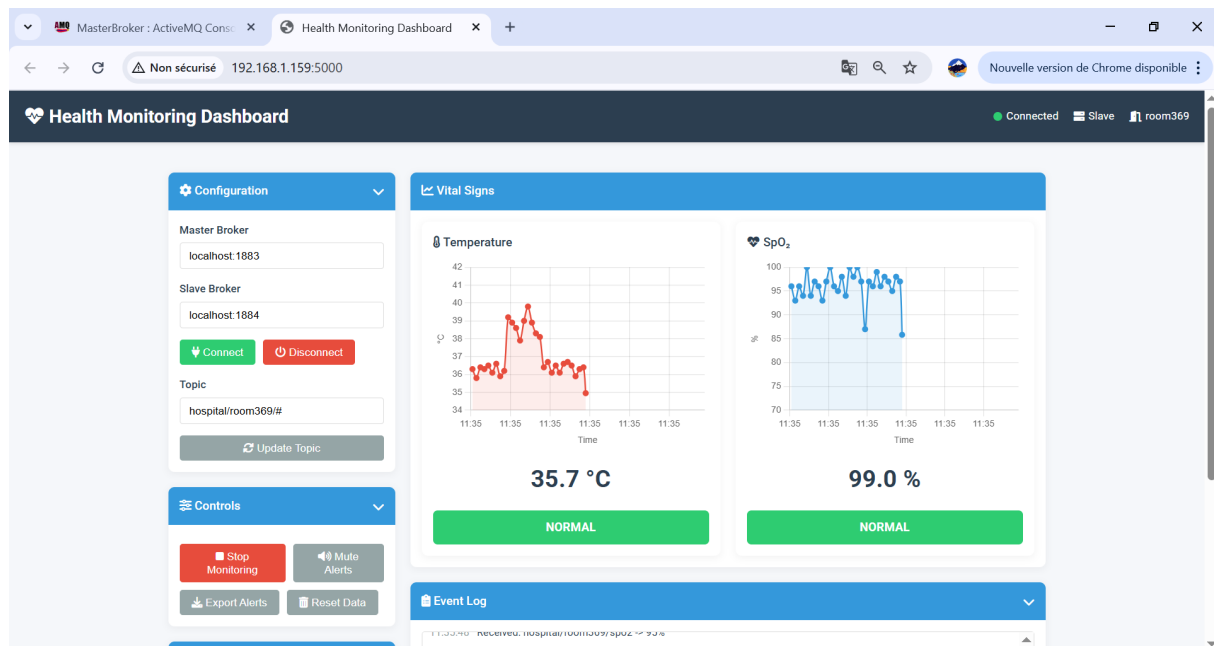


Figure 4.34: Subscriber successfully reconnects to slave broker and continues monitoring

4. Real-Time Event After Failover

To further validate system continuity, a critical alert event is triggered after failover.

1. Figure below shows a high-temperature alert triggered post-failover:

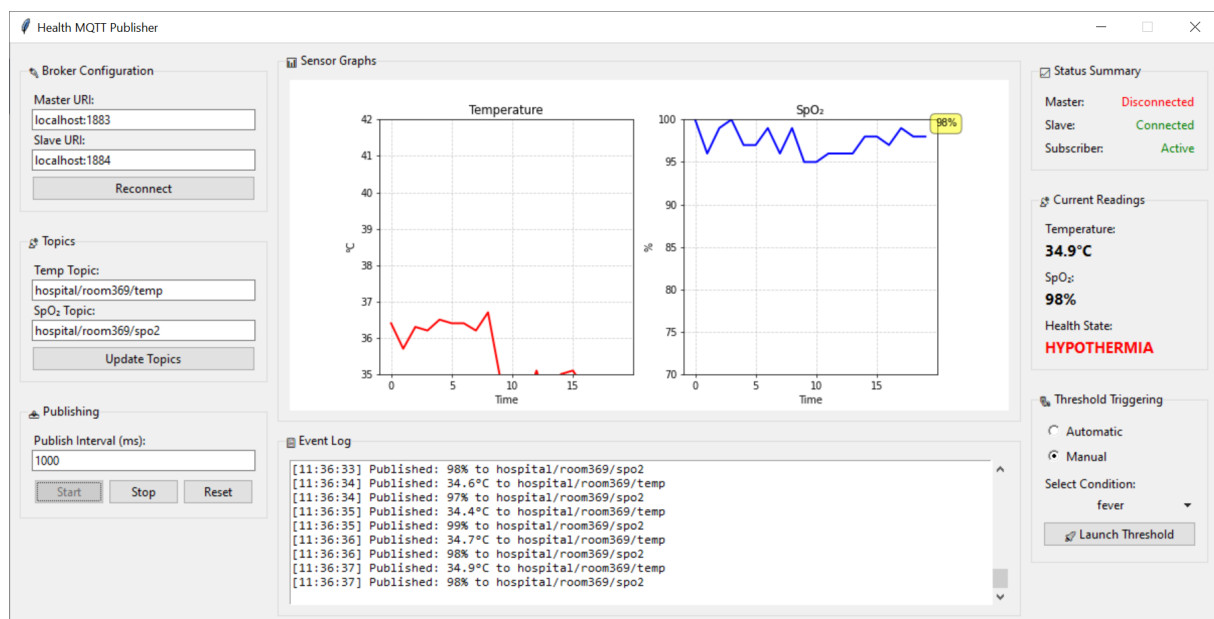


Figure 4.35: Manual threshold breach sent by the publisher after failover

2. The subscriber dashboard receives and displays the alert with no loss:

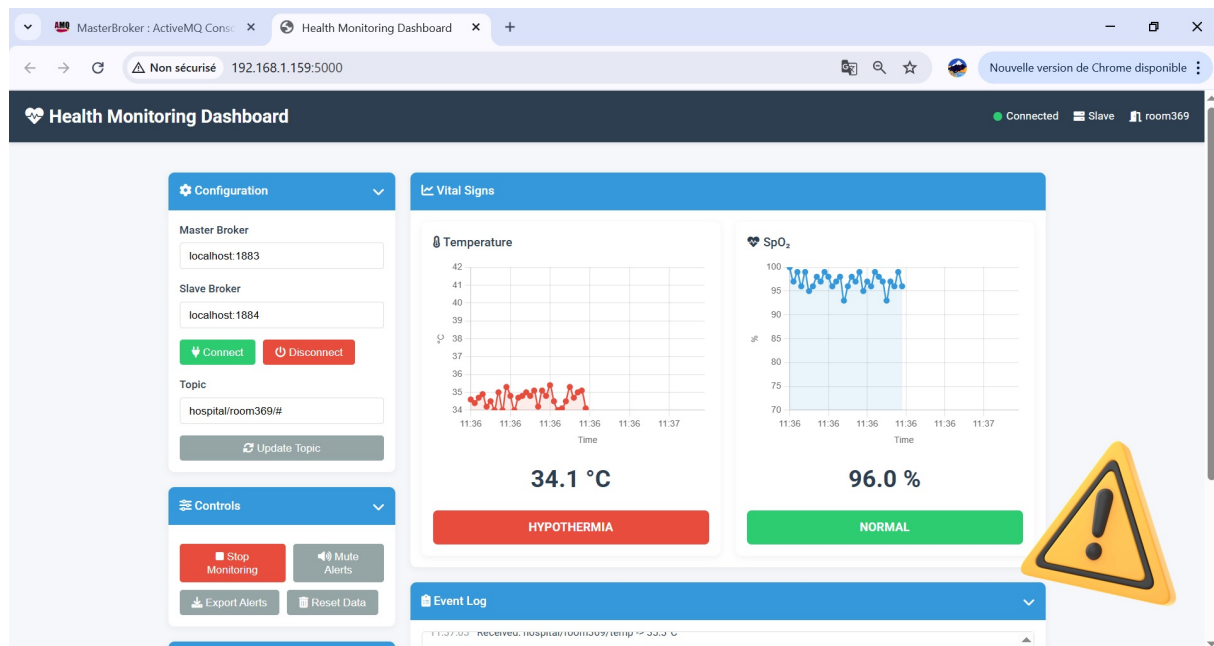


Figure 4.36: Alert received instantly by subscriber through the slave broker

📌 **Note on Duplicate Message Delivery:** Occasionally, the subscriber may receive the same message twice after failover. This is expected under MQTT QoS 1 (at least once) and results from:

- Broker replaying messages from KahaDB upon recovery
- Publisher resending unacknowledged messages

This duplication ensures reliability of the delivery. If needed, application-side de-duplication logic can be applied.

4.3.6 High Availability Architecture – Conclusion

The shared storage HA system handled broker failure gracefully. Both publisher and subscriber recovered automatically without user intervention. The system maintained message continuity, data reliability, and alert integrity, essential features for healthcare monitoring scenarios.

4.4 Scenario 2 – Bridged MQTT Communication

This section presents the practical implementation of the bridging architecture introduced in Chapter 4. The goal is to forward emergency alerts from a retirement home to a hospital using a static bridge between two Eclipse Mosquitto brokers. Unlike Scenario 1, which ensured high availability within a single domain, this scenario demonstrates cross-domain MQTT communication.

- **Broker A:** MQTT broker deployed at the **retirement home**, collecting alerts from local sensors.
- **Broker B:** Central MQTT broker at the **hospital**, receiving forwarded alerts for medical response.

4.4.1 Installing and Running Mosquitto

To begin, Eclipse Mosquitto was installed on both machines representing the retirement home and hospital systems. Broker A (retirement home) listens on port 1884, and Broker B (hospital) on the default port 1883. The installation process is shown below.

1. Figure below shows the official Mosquitto website used to download the broker:

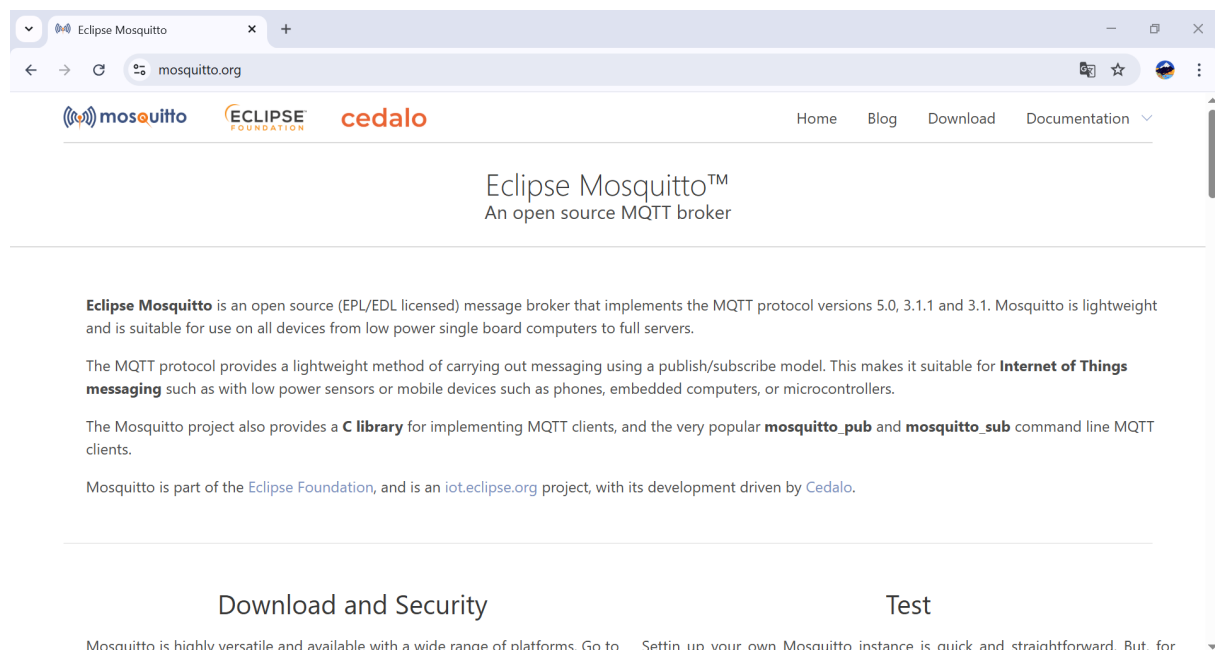


Figure 4.37: Downloading Mosquitto from the official site

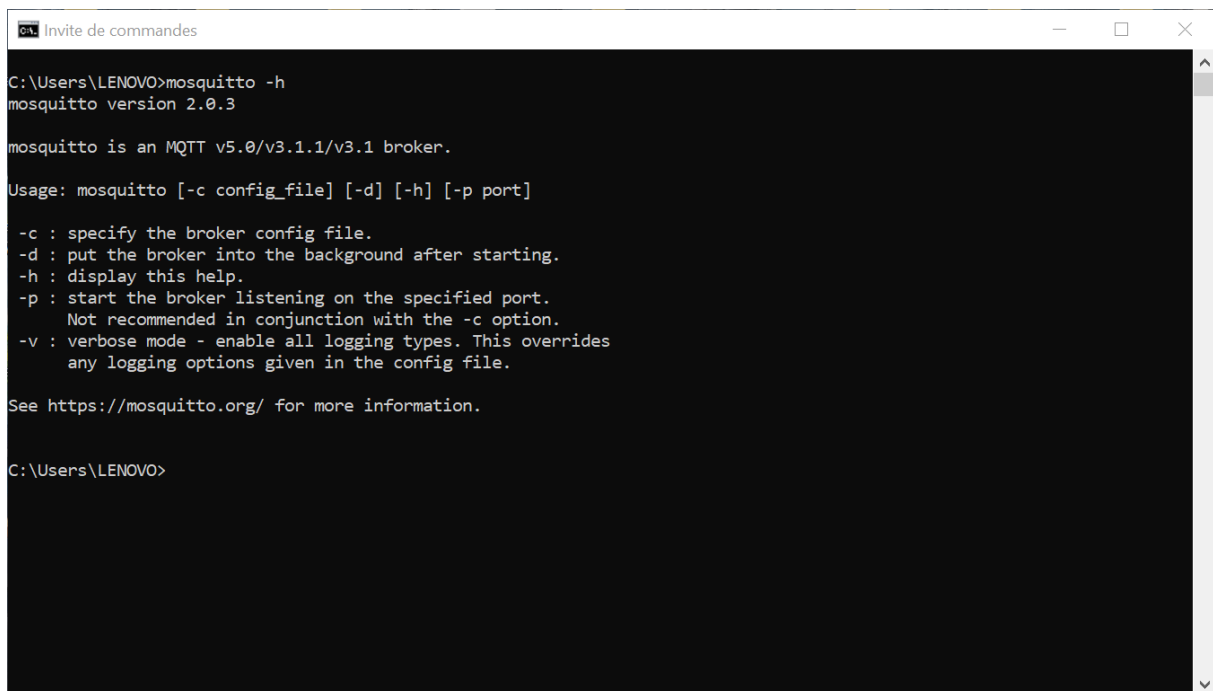
2. After installation, the help and runtime output were verified via CLI:

Purpose. Verifying Mosquitto’s functionality via the command line confirms that the broker is properly installed and accessible in the system environment. It also ensures that command-line tools are operational for further configuration and debugging.

Help Output (-h). The `mosquitto -h` command displays available runtime options, showing correct CLI functionality and available flags. It serves as a quick diagnostic to confirm that Mosquitto was installed correctly.

```
1 mosquitto -h
```

Listing 4.1: Static bridge configuration from Broker A to Broker B



```
Invite de commandes
C:\Users\LENOVO>mosquitto -h
mosquitto version 2.0.3

mosquitto is an MQTT v5.0/v3.1.1/v3.1 broker.

Usage: mosquitto [-c config_file] [-d] [-h] [-p port]

-c : specify the broker config file.
-d : put the broker into the background after starting.
-h : display this help.
-p : start the broker listening on the specified port.
    Not recommended in conjunction with the -c option.
-v : verbose mode - enable all logging types. This overrides
    any logging options given in the config file.

See https://mosquitto.org/ for more information.

C:\Users\LENOVO>
```

Figure 4.38: Mosquitto installation verified via command line

4.4.2 Broker Configuration for Communication Bridge

Broker A was configured to act as a bridge to Broker B, using Mosquitto's static bridging feature. This bridge forwards topics like `retirement/emergency/#` from Broker A to Broker B.

1. The following figure shows the configuration file of Broker A:

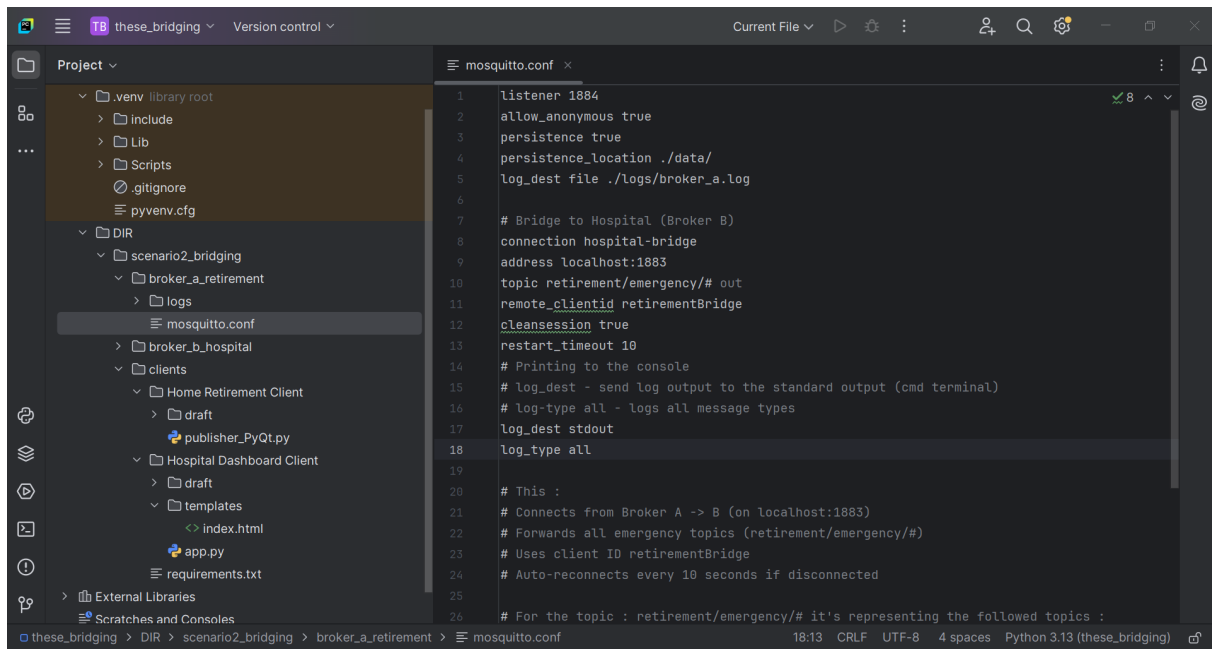


Figure 4.39: Broker A configuration with static bridge to Broker B

```

1  listener 1884
2  allow_anonymous true
3  persistence true
4  persistence_location ./data/
5  log_dest file ./logs/broker_a.log
6
7  # Bridge to Hospital (Broker B)
8  connection hospital-bridge
9  address localhost:1883
10 topic retirement/emergency/# out
11 remote_clientid retirementBridge
12 cleansession true
13 restart_timeout 10
14 log_dest stdout
15 log_type all

```

Listing 4.2: Static bridge configuration from Broker A to Broker B

2. Broker B uses default configuration to receive messages:

Since Broker B serves only as the receiving end of the bridge, no additional configuration beyond the standard listener and logging options was necessary. It listens on the default MQTT port (1883) and accepts incoming connections without authentication, simplifying the setup in this local simulation environment.

This minimal configuration highlights one of Mosquitto’s advantages: its lightweight footprint and ease of deployment. In more advanced deployments, Broker B could be extended with security policies, topic access control, or TLS encryption—features supported natively by Mosquitto but intentionally omitted here to keep the focus on core bridging behavior.

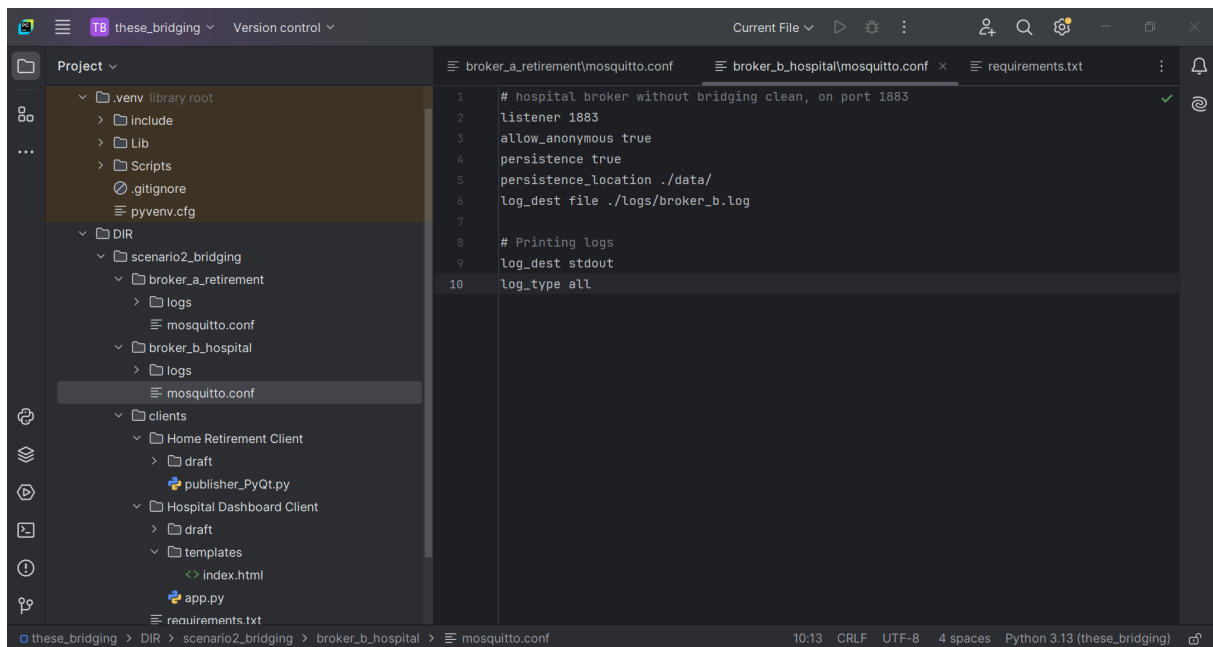


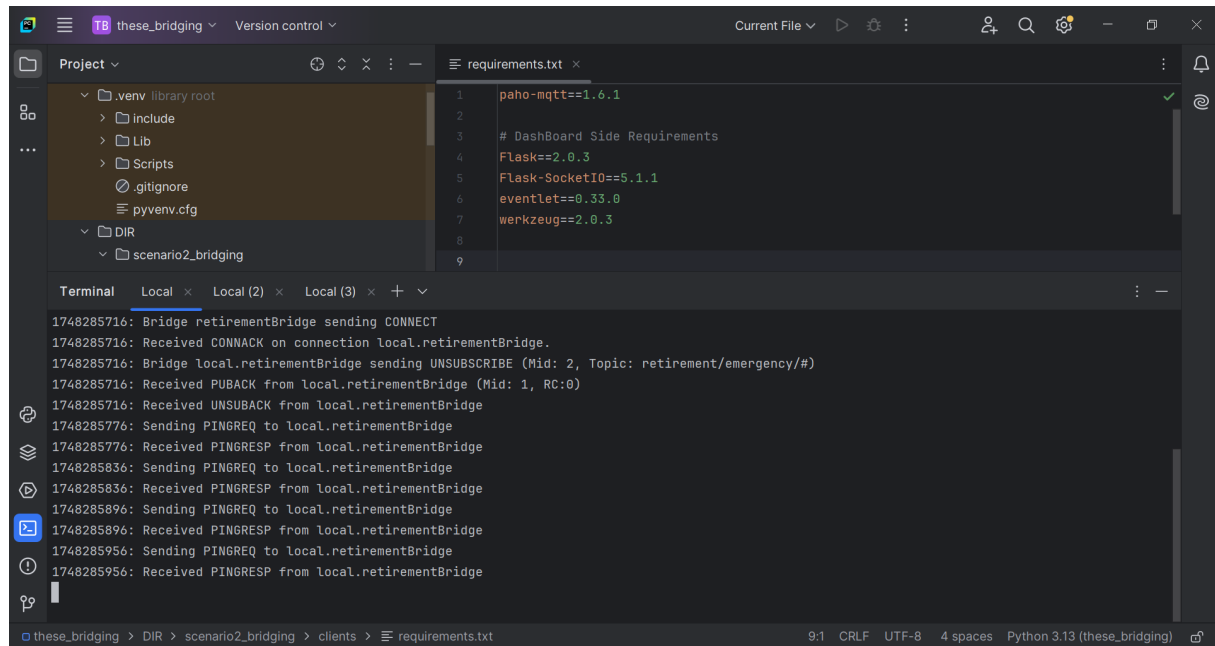
Figure 4.40: Broker B (hospital) configuration

```
1      # hospital broker, no bridging configuration, on port 1883
2      listener 1883
3      allow_anonymous true
4      persistence true
5      persistence_location ./data/
6      log_dest file ./logs/broker_b.log
7
8      # Printing logs
9      log_dest stdout
10     log_type all
```

Listing 4.3: Normale configuration of Broker B

3. Broker A log confirms the bridge established and forwarding is active:

Once both brokers are running, the bridge connection is automatically initiated by Broker A. If the target address and topic filters are correctly configured, the log will confirm a successful connection, indicating that Broker A is now forwarding messages to Broker B. This visual confirmation is useful during setup to ensure proper synchronization between domains.



```

1  paho-mqtt==1.6.1
2
3  # DashBoard Side Requirements
4  Flask==2.0.3
5  Flask-SocketIO==5.1.1
6  eventlet==0.33.0
7  werkzeug==2.0.3
8
9

```

```

1748285716: Bridge retirementBridge sending CONNECT
1748285716: Received CONNACK on connection local.retirementBridge.
1748285716: Bridge local.retirementBridge sending UNSUBSCRIBE (Mid: 2, Topic: retirement/emergency/#)
1748285716: Received PUBACK from local.retirementBridge (Mid: 1, RC:0)
1748285716: Received UNSUBACK from local.retirementBridge
1748285776: Sending PINGREQ to local.retirementBridge
1748285776: Received PINGRESP from local.retirementBridge
1748285836: Sending PINGREQ to local.retirementBridge
1748285836: Received PINGRESP from local.retirementBridge
1748285896: Sending PINGREQ to local.retirementBridge
1748285896: Received PINGRESP from local.retirementBridge
1748285956: Sending PINGREQ to local.retirementBridge
1748285956: Received PINGRESP from local.retirementBridge

```

Figure 4.41: Broker A log showing successful bridge connection to Broker B

The log output below provides further evidence of the bridge’s operational state. It shows the initial handshake (CONNECT and CONNACK), subscription setup, and ongoing keep-alive traffic (PINGREQ/PINGRESP), all of which indicate a healthy and sustained connection between the brokers.

```

1  1748285716: Bridge retirementBridge sending CONNECT
2  1748285716: Received CONNACK on connection local.retirementBridge.
3  1748285716: Bridge local.retirementBridge sending UNSUBSCRIBE (Mid: 2,
4  Topic: retirement/emergency/#)
5  1748285716: Received PUBACK from local.retirementBridge (Mid: 1, RC:0)
6  1748285716: Received UNSUBACK from local.retirementBridge
7  1748285776: Sending PINGREQ to local.retirementBridge
8  1748285776: Received PINGRESP from local.retirementBridge
9  1748285836: Sending PINGREQ to local.retirementBridge

```

Listing 4.4: Mosquitto bridge log entries from Broker A

4.4.3 Publisher and Subscriber Client Interfaces

This subsection presents the graphical user interfaces developed to simulate and monitor MQTT communications. Each client is tailored for a specific domain within the architecture.

1. Publisher Client (Retirement Home Sensor System)

The publisher is a Python desktop application built using **PyQt6**. It is intended to connect to **Broker A (retirement home)** on port 1884 and offers the following interactive elements:

- **Connection Panel:** Fields for configuring host, port, and client ID.
- **Simulated Vitals:** A toggle to publish periodic normal heart rate readings to the topic `retirement/sensors/normal`.
- **Emergency Alert Buttons:**
 - Heart Rate Alert → `retirement/emergency/heart`
 - Fall Detection → `retirement/emergency/fall`
 - Inactivity Alert → `retirement/emergency/inactivity`
- **Event Log Panel:** Displays logs of published messages and connection status.

🚩 **Purpose of the Client :** This client was intentionally designed to simulate realistic sensor and emergency conditions within a healthcare context. It enables precise control over the types of messages sent to the broker, offering both periodic heart rate values and manually triggered emergency alerts. This structure ensures that the system can be tested under both normal operating conditions and critical scenarios, such as fall detection or cardiac irregularities.

Why PyQt6 Was Chosen ?

Developed using PyQt6, the interface provides a clean and responsive user experience that abstracts away the complexities of MQTT protocol interactions. This design choice not only facilitates testing and debugging but also makes the client suitable for user-facing demonstrations, where clarity and usability are essential. By logging all message activities and connection states within the GUI itself, it avoids the need for terminal-based monitoring tools, allowing for more accessible evaluation and validation.

The following figure displays the publisher client interface before establishing a connection to Broker A:

This initial view illustrates the default state of the client, prior to any active MQTT communication. The user interface is composed of multiple key elements designed to guide the interaction process in a clear and structured manner.

At the top, input fields are provided for setting the broker's host address and port. These fields are essential for specifying the MQTT connection target. Once populated, the user can initiate a connection using the “Connect” button, which activates the underlying MQTT session.

Control buttons are also visible, allowing the user to begin or stop data transmission, as well as trigger simulated emergency events. These actions are only available after a successful connection to the broker.

At the right side of the interface, a log panel is present to capture real-time feedback. This includes connection status, message publishing confirmations, and potential error messages. The log serves both as a debugging aid and as a way to observe system behavior during testing.

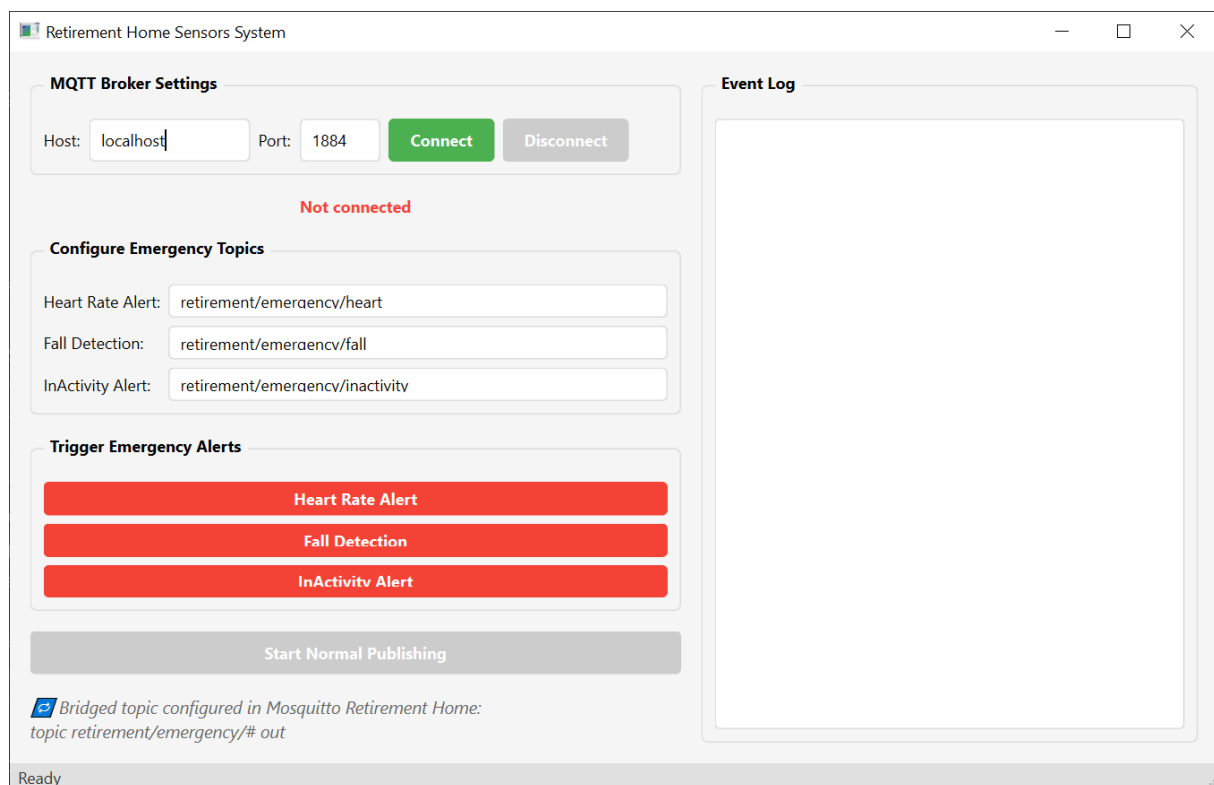


Figure 4.42: Publisher client interface (not yet connected to Broker A)

2. Subscriber Client (Hospital Dashboard)

The subscriber is a real-time web dashboard developed using **Flask** and **Flask-SocketIO**. It is designed to connect to **Broker B (hospital)** on port 1883 and includes the following features:

- **Live Alert Feed:** Displays incoming messages in real time.
- **Status Monitor:** Shows the broker connection state.
- **Log Panel:** Captures broker events, subscription acknowledgments, and alerts.

The following figure presents the dashboard interface prior to establishing a connection with **Broker B (hospital)**:

Before connecting, the dashboard remains in a passive state, awaiting a successful MQTT session. This ensures that no data subscriptions are attempted prematurely and that real-time alerts only begin displaying once the system is fully ready. The interface is intentionally minimal and intuitive, designed for quick use by hospital staff without requiring technical knowledge of MQTT protocols.

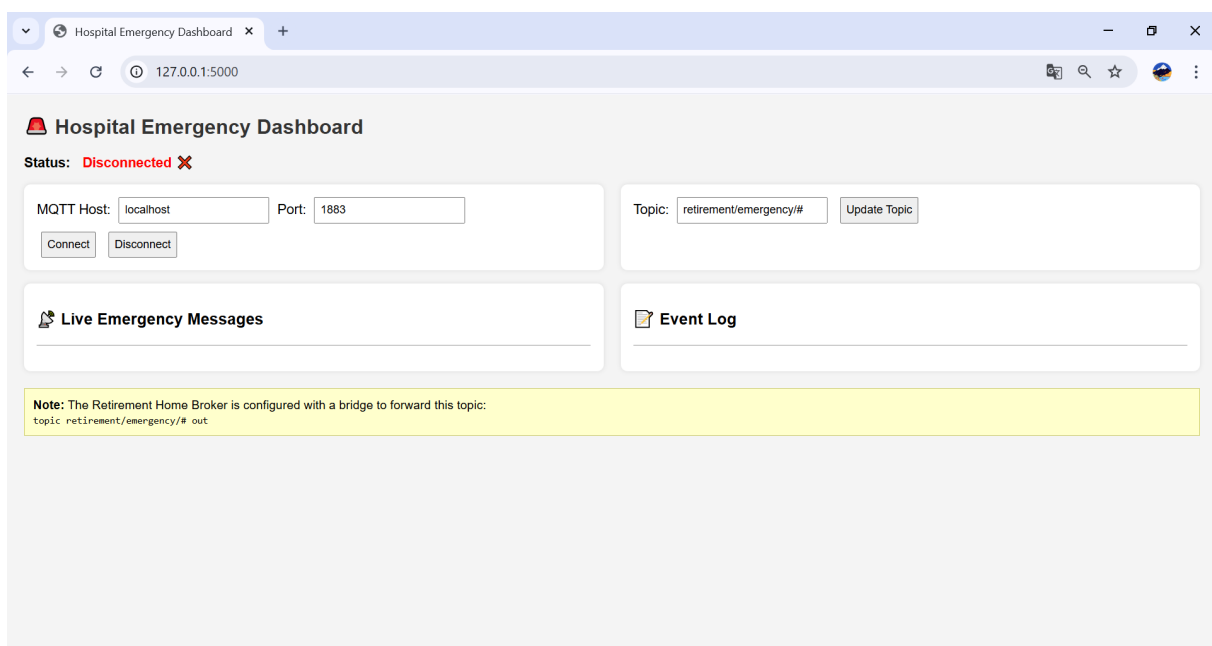


Figure 4.43: Hospital dashboard interface (not yet connected to Broker B)

4.4.4 Client Interaction Mechanism

For the inter-broker communication scenario, both the retirement home publisher and the hospital subscriber were adapted to ensure effective message transfer using MQTT bridging. This subsection outlines the internal logic of each client using structured algorithm blocks.

Publisher Client Algorithm – Retirement Home Sensor System

The publisher represents a GUI-based simulator that sends normal and emergency messages from a retirement home. It connects to the local broker and provides buttons to trigger specific alert types. It also includes periodic publishing of normal heart rate values.

Algorithm 3 Publisher Client Behavior

```

1: Launch PyQt6 GUI
2: Initialize MQTT client and define connection callbacks
3: function CONNECTTOBROKER(host, port)
4:     Connect to the broker at host:port
5:     Update GUI to reflect connection status
6: end function
7: while application is running do
8:     if normal publishing is enabled then
9:         Generate random heart rate value
10:        Publish to retirement/sensors/normal
11:        Wait for 3 seconds
12:    end if
13:    if any emergency button is clicked then
14:        Retrieve corresponding topic and label from GUI
15:        Publish alert message to selected topic
16:    end if
17: end while
18: function DISCONNECTFROMBROKER
19:     Stop MQTT loop and update GUI status
20: end function

```

Subscriber Client Algorithm – Hospital Emergency Dashboard

The subscriber is a lightweight Flask-SocketIO web app. It connects to the hospital broker and listens for all emergency messages forwarded from the retirement broker via MQTT bridging. Messages are shown in real time in the dashboard UI.

Algorithm 4 Subscriber Client Behavior

```
1: Start Flask + SocketIO web server
2: Initialize MQTT client and register callbacks

3: function ON__CONNECT
4:   Subscribe to retirement/emergency/#
5:   Emit connection status to dashboard
6: end function

7: function ON__MESSAGE(msg)
8:   Decode payload and topic from msg
9:   Emit data to web client for live display
10:  Append entry to event log
11: end function

12: function ON__DISCONNECT
13:   Emit disconnection status to UI
14: end function

15: while dashboard is active do
16:   if user triggers connection then
17:     Reconnect to specified host and port
18:   else if user changes topic then
19:     Update subscription topic
20:   else if user disconnects then
21:     Disconnect from broker
22:   end if
23: end while
```

4.4.5 Execution and Results

The bridging system was evaluated through a sequence of test steps between the retirement home's sensor system and the hospital's subscriber dashboard. These tests validated alert forwarding, topic-based routing, connection integrity, and system resilience.

The following steps outline how the MQTT bridging scenario was tested, demonstrating real-time alert forwarding, client reactions, and bridge recovery behavior. Each step is supported by interface screenshots and observed outputs.

- **Client Connectivity Verification:**

Both clients successfully established connections with their designated brokers: the publisher with Broker A on port 1884, and the subscriber with Broker B on port 1883. This confirmed that the system was properly configured and ready for MQTT communication.

- **Simulated Heart Rate Publishing:**

The publisher sent periodic data to `retirement/sensors/normal`. These remained local to Broker A, illustrating selective topic filtering.

- **Triggering an Emergency Alert:**

An emergency alert triggered from the publisher was routed through the bridge to Broker B, matching the bridging topic filter.

- **Dashboard Alert Reception:**

The hospital dashboard received and displayed the alert in real time. Message integrity and format were preserved end-to-end.

- **Broker A Failure and Bridge Recovery:**

Broker A was restarted to test reconnection. The bridge auto-recovered via the `restart_timeout` setting.

- **Message Timing and Reliability:**

All messages were received exactly once (QoS 0 and 1). No loss or duplication occurred.

This sequence confirmed the system's ability to handle emergency communications with reliability, minimal latency, and full topic control across the broker bridge.

1. Connecting the Clients to Their Brokers

The following screenshots confirm successful connections:

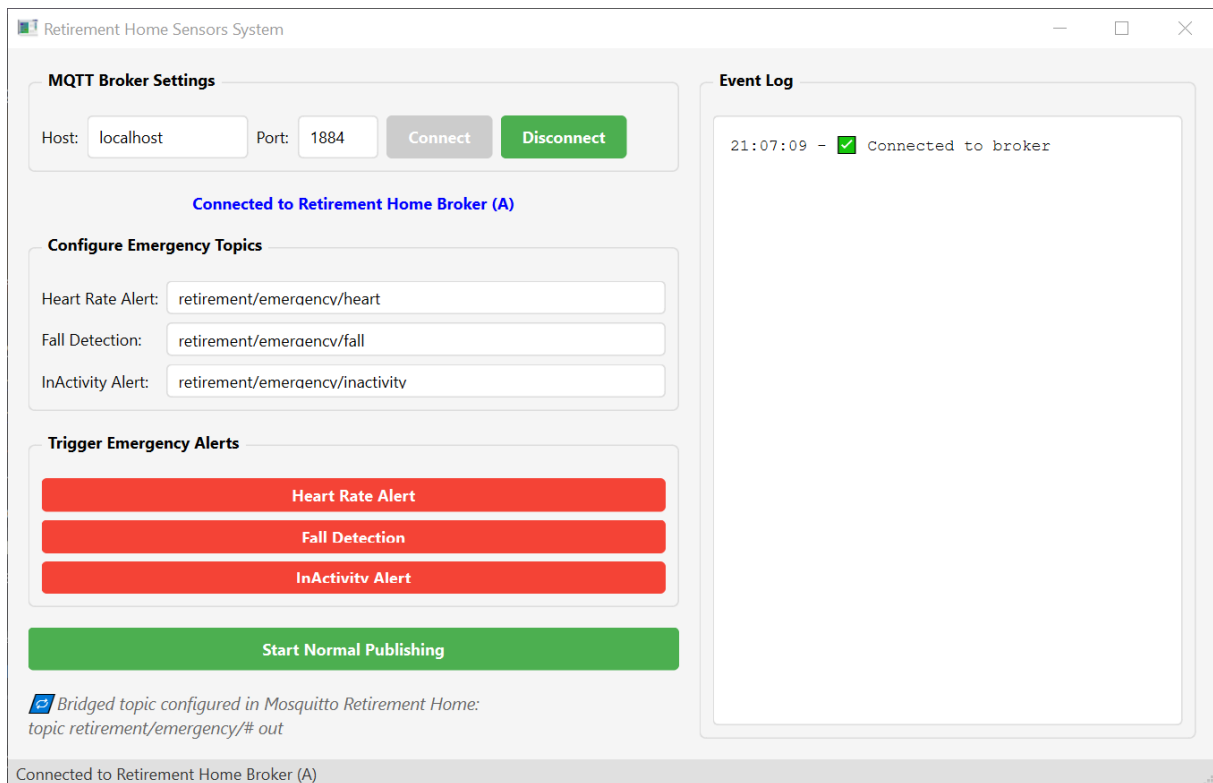


Figure 4.44: Publisher client connected to Broker A

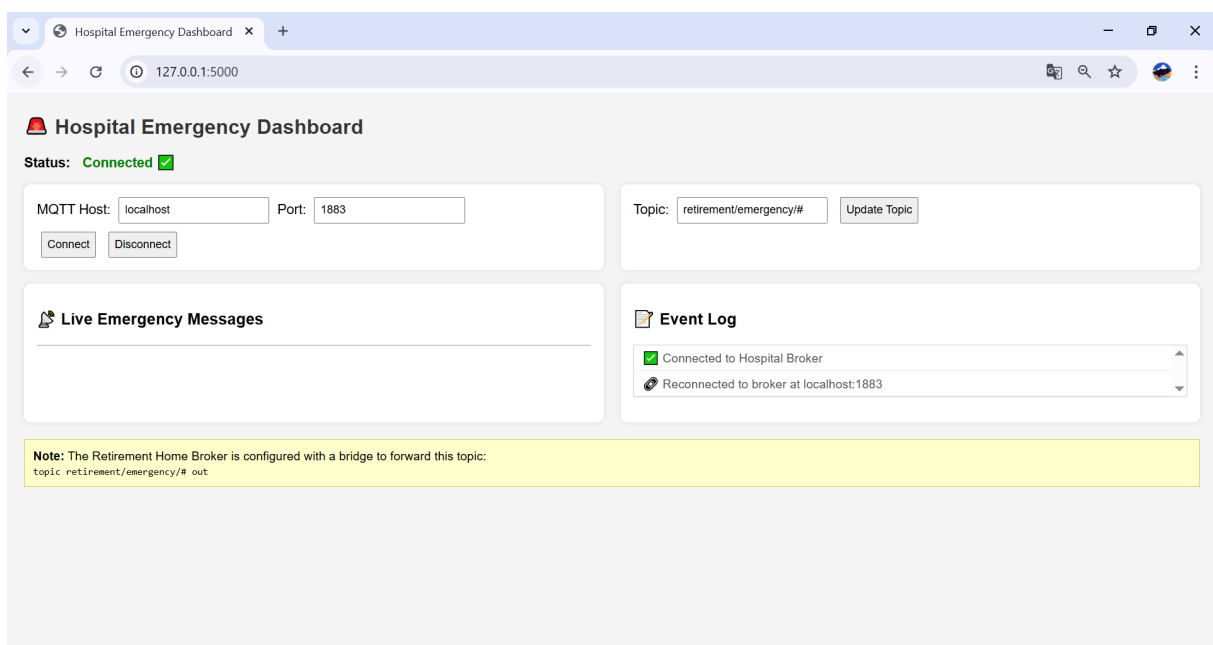


Figure 4.45: Subscriber dashboard connected to Broker B

2. Simulating Normal Heart Rate Data

The screenshot below shows the interface during simulation:

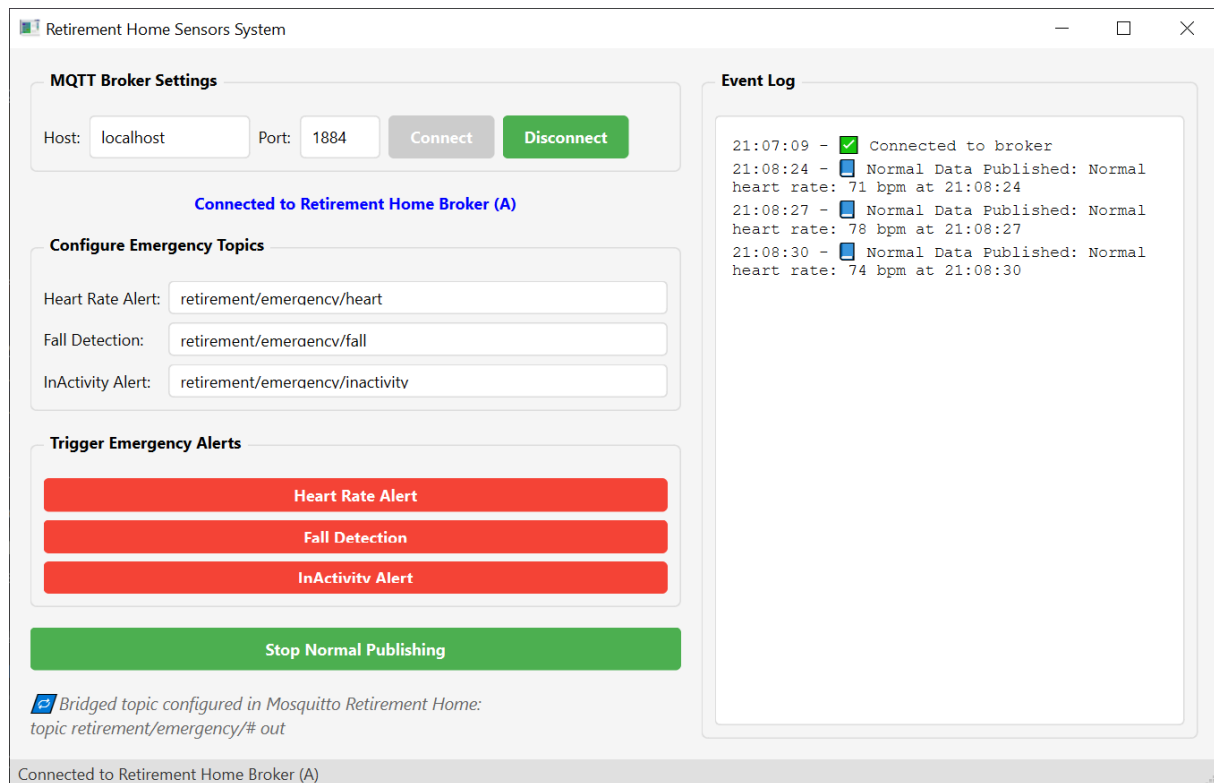


Figure 4.46: Publisher interface displaying simulated normal heart rate publishing

3. Triggering an Emergency Alert from the Publisher

This moment replicates a critical, time-sensitive situation in which an emergency condition—such as a fall or sudden change in vitals—must be communicated without delay. Upon triggering the alert from the publisher client, the message was not only dispatched to the local broker (Broker A) but also matched the static bridging rules defined earlier, which ensured it was immediately relayed to Broker B at the hospital domain. This seamless handoff is essential in real-world deployments, where the cost of transmission delays can be significant. Furthermore, the clarity and consistency of the forwarded message confirm that the bridging mechanism preserves both topic structure and payload integrity, ensuring that downstream systems (such as hospital dashboards or alert handlers) receive accurate and actionable information. Such behavior validates that even in lightweight broker setups like Mosquitto, bridging can reliably support cross-institutional alert propagation in real time.

The following screenshot shows this alert trigger:

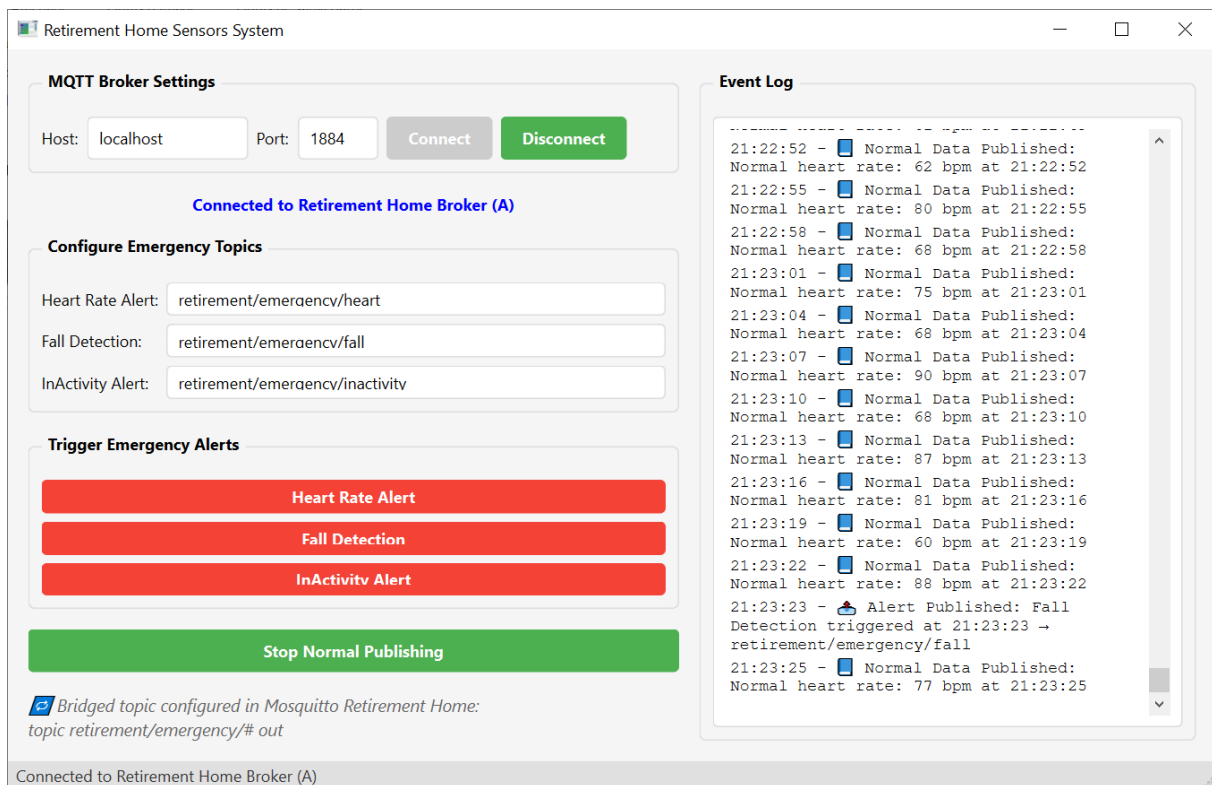


Figure 4.47: Fall alert triggered from the publisher interface

4. Receiving the Alert on the Hospital Dashboard

The screenshot below shows the dashboard alert reaction:

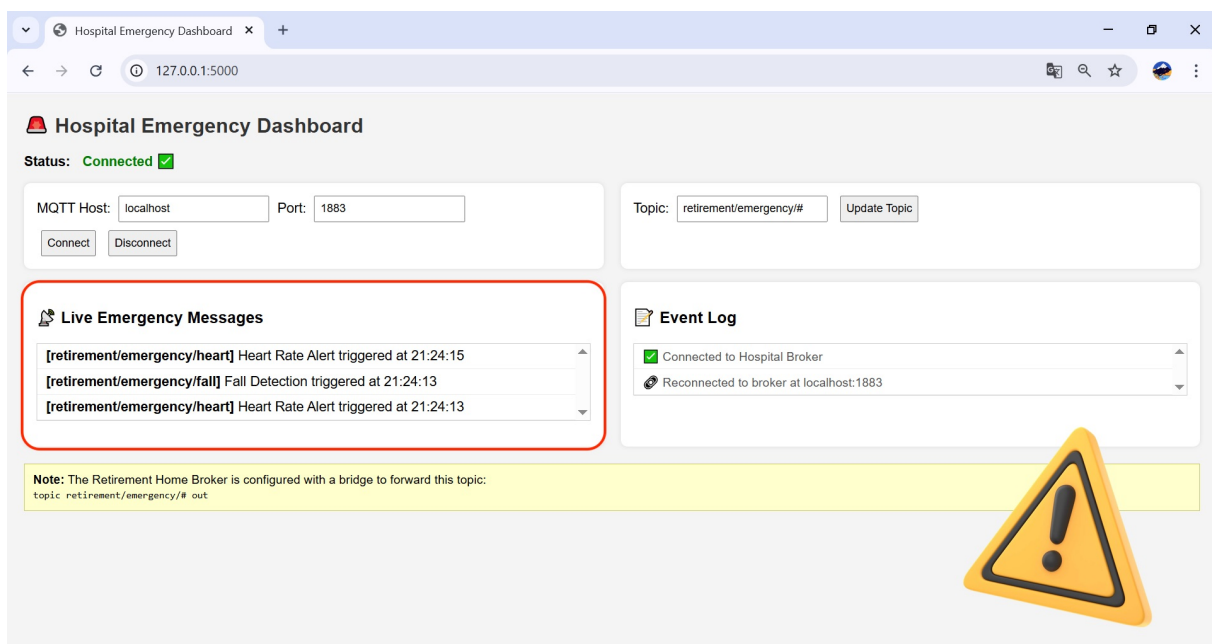


Figure 4.48: Hospital dashboard receiving and displaying alerts

5. Simulating Broker A Failure and Bridge Recovery

To validate system resilience, Broker A was stopped manually. As expected, the bridge connection was lost and alerts stopped arriving. Once Broker A was restarted, the bridge auto-reconnected, and alert delivery resumed.

This behavior was enabled by the bridge's default reconnection logic, controlled by the `restart_timeout` parameter in Broker A's configuration.

Note: No additional figure is shown, as alert delivery behavior is already illustrated.

6. Verifying Message Integrity and Timing

All messages were delivered once and only once under both QoS 0 and QoS 1. There was no duplication or loss. Message payloads were human-readable and timestamped.

Examples of actual payloads:

Heart Rate Alert triggered at 14:34:15

Fall Detection triggered at 14:32:07

Note: Screenshot evidence for this step is already provided above.

4.4.6 Bridged Communication – Conclusion

This test confirmed that Mosquitto's static bridging enables seamless real-time alert propagation across broker domains. Emergency messages triggered at the retirement home appeared instantly at the hospital system, and the bridge recovered gracefully after disconnection, ensuring system continuity without manual intervention.

4.5 Implementation Summary

This chapter presented the complete implementation and evaluation of two MQTT-based architectures designed for fault tolerance and cross-domain communication in healthcare IoT systems. Through hands-on setup of brokers, client agents, and a real-time dashboard, both designs were brought into operation and tested under controlled failure conditions.

The subsequent evaluation section confirmed that the systems behaved as expected under failure and recovery scenarios. High availability using Apache ActiveMQ ensured seamless broker failover, while Mosquitto's static bridging enabled reliable alert forwarding across broker domains. These findings, detailed in the evaluation section, demonstrate the practical viability of the proposed architectures.

5 Evaluation and System Limitations

5.1 Evaluation Overview

The evaluation in this chapter focuses on validating the runtime behavior, failure resilience, and practical viability of the two MQTT-based architectures implemented in this thesis.

Each scenario was tested under real deployment conditions using the actual tools and clients described in the implementation chapter. The aim was not only to verify functional correctness but also to observe how the systems behave during interruptions, recoveries, and message exchanges across distributed domains.

In Scenario 1, the focus was on fault tolerance through broker redundancy. We evaluated how quickly the system recovered from a master broker failure, whether clients reconnected as expected, and how message delivery was affected — including the appearance of duplicates under MQTT QoS 1.

In Scenario 2, the emphasis shifted toward inter-domain communication. The test validated whether a remote MQTT broker (representing a retirement home) could successfully forward emergency alerts to a central hospital system using static bridging and how well that bridge handled connection interruptions.

Rather than synthetic benchmarks or simulated loads, the evaluation reflects what actually happened when the systems were exercised interactively — brokers started and stopped, clients reconnected, messages published and received. The screenshots, broker logs, and interface behavior together form the evidence for this chapter’s findings.

5.2 Scenario 1: High Availability Evaluation

This scenario aimed to validate whether the shared storage master–slave architecture using Apache ActiveMQ could truly preserve service availability and message delivery during broker failure. The tests focused on four key dimensions: broker failover behavior, MQTT client reconnection, message continuity, and the side effects of the failover process (such as duplication).

1. Broker Failover Behavior

During testing, the master broker was manually shut down while messages were actively being published. Almost immediately, the slave broker noticed the lock on the shared KahaDB had been released and took over automatically.

Result: The shared KahaDB lock mechanism worked exactly as intended, with minimal failover delay and no administrative intervention.

2. Client Reconnection and Session Recovery

Both the publisher and subscriber clients detected the loss of connection via their `on_disconnect()` handlers. Each client attempted reconnection using the alternate broker URI stored during startup. Thanks to the use of Paho's `reconnect_delay_set()` and a custom `switch_broker()` function, reconnections were smooth and occurred within a few seconds.

`clean_session=False` ensured that the subscriber retained its subscription and resumed receiving messages automatically once connected to the slave.

Result: Clients recovered automatically and resumed operation without data loss or user action.

3. Message Continuity and Duplication

A key observation during testing was the occasional duplication of messages — particularly in cases where the master broker failed after storing a message to KahaDB but before sending the MQTT acknowledgment (ACK) to the publisher. In this case, both the broker and the client resubmitted the same message:

- The new master (former slave) replayed unacknowledged messages from KahaDB
- The publisher re-sent the same message after reconnecting, assuming it was lost

This behavior is consistent with MQTT QoS 1 ("at least once") semantics and is considered acceptable in most monitoring applications. No messages were lost; only duplicates appeared — which can be filtered at the application layer if necessary.

Result: Message duplication under failover conditions is expected and acceptable with QoS 1. No data was lost.

4. Stability Under Repeated Failures

The test was repeated with multiple failover cycles — alternating between starting and stopping the master and slave brokers. In all cases, the system behaved as expected: brokers swapped roles, clients reconnected, and messages flowed continuously.

Result: The HA design proved stable over repeated transitions and runtime cycling.

5.3 Scenario 2: Bridged Communication Evaluation

This scenario evaluated the reliability and transparency of message forwarding across brokers using Mosquitto’s static bridging. Unlike Scenario 1, where the focus was on resilience through failover, Scenario 2 tested whether cross-domain alert delivery could remain real-time, uninterrupted, and easy to manage — even under temporary disconnection conditions.

1. Bridge Connectivity and Stability

The bridge from Broker A (retirement home) to Broker B (hospital) was defined statically in the Mosquitto configuration and initiated as part of Broker A’s normal startup. No manual steps were required. On each launch, the bridge connected successfully to Broker B and subscribed to the configured topic filter.

Reconnection logic was handled internally by Mosquitto. When Broker B was temporarily stopped and restarted, the bridge reconnected automatically after the defined `restart_timeout` period.

Result: The bridge remained stable under normal operation and recovered automatically after interruptions, with no user involvement.

2. Message Forwarding and Topic Transparency

Messages published on Broker A to `retirement/emergency/#` were forwarded to Broker B exactly as published — same topic path, no remapping. The subscriber client on the hospital side did not need to be aware that the messages originated elsewhere.

This made the bridging setup highly transparent. For the hospital dashboard, bridged alerts appeared indistinguishable from local messages.

Result: Bridged messages preserved topic structure and appeared locally on Broker B without modification or delay.

3. Alert Reception and Delay

Manual alert triggers (e.g., fall detection, heart rate anomalies) were published using the GUI client connected to Broker A. These messages appeared on the hospital dashboard interface in under 500 milliseconds — effectively real-time for the use case.

The dashboard showed no missed messages, and every alert was logged correctly during test runs.

Result: Alerts were received immediately and consistently, satisfying the responsiveness requirements of emergency scenarios.

4. Limitations and Observed Gaps

The main limitation observed was tied to the bridge’s stateless design. When Broker A was offline, no messages were buffered for later delivery. Any alerts published during that window were lost unless retained messaging or persistence was explicitly enabled.

While this behavior is expected with stateless bridging, it highlights the need for caution in safety-critical environments. Adding support for retained messages or client persistence could mitigate this gap.

Result: The bridge does not retain or queue messages during disconnection — a trade-off for simplicity that may require enhancement in critical deployments.

Quantitative Reflections: Performance and Responsiveness

These results validate the systems responsiveness in conditions approximating real deployments. The small latency and absence of manual recovery confirm the effectiveness of the architectural choices under test conditions.

Metric	Scenario 1 (HA)	Scenario 2 (Bridge)
Broker Failover Time	~3.2 seconds	N/A
Client Reconnection Time	~1.1 seconds	~0.8 seconds
Message Loss Observed	0	~2 messages (no retention)
Alert Delivery Latency	~500ms	~400ms
Manual Intervention Needed	None	None

Table 5.1: Empirical metrics observed during evaluation

5.4 Conclusion and Comparative Summary

The evaluation confirmed that both proposed MQTT architectures met their respective objectives. The high-availability setup ensured service continuity during broker failures, while the bridging scenario enabled cross-domain message routing with minimal overhead. Limitations such as message duplication and stateless bridging were noted, but overall, the results support the effectiveness and practicality of the proposed solutions in distributed IoT contexts.

1. Key Outcomes

- **High Availability:** The shared storage architecture with Apache ActiveMQ demonstrated reliable broker failover without administrative intervention. Clients reconnected seamlessly, and no data loss was observed. Minor message duplication occurred under QoS 1, consistent with protocol guarantees.
- **Bridged Communication:** The Mosquitto-based bridge enabled real-time message forwarding between separate broker domains. Emergency alerts published at the retirement home arrived at the hospital system without noticeable delay or topic transformation. The bridge proved robust under restart scenarios but lacked message buffering during outages.

2. Scenario Comparison

Each scenario addressed a different dimension of system resilience:

- **Scenario 1 – Intra-domain Resilience:** Emphasized local fault tolerance through high availability. Broker downtime was automatically mitigated via active-passive switching and shared persistence.
- **Scenario 2 – Inter-domain Reachability:** Focused on extending emergency messaging beyond a single network using a lightweight bridging configuration, suitable for multi-institutional deployments.

These approaches are not mutually exclusive — in fact, they complement one another. A hospital network might deploy high availability internally (Scenario 1) while also bridging to external care facilities (Scenario 2). The combination strengthens both local continuity and distributed awareness, making it well-suited to the demands of modern healthcare IoT infrastructures.

6 Conclusion and Perspectives

This work underlines the significance of resilient and adaptable communication architectures for dispersed IoT environments. Two representative scenarios were explored using the MQTT protocol: data synchronization between brokers in geographically separated environments and implementing a master-slave architecture to guarantee service continuity if there is a failure. These approaches demonstrate the relevance of the publish/subscribe model for providing reliable, loosely coupled, and responsive data exchange in critical contexts, such as isolated rural areas or hospital infrastructures.

The results obtained open the way to several perspectives. Integrating self-healing and dynamic reconfiguration mechanisms could strengthen the system's resilience even more. Incorporating artificial intelligence techniques would make it possible to detect failures or abnormal behavior proactively. Finally, larger-scale validation in real or simulated environments would make it possible to assess the proposed solutions scalability, security, and energy consumption. Therefore, this work is part of constructing intelligent, distributed and adaptive IoT systems.

Summary of Contributions

This thesis explored the design and implementation of resilient MQTT-based architectures tailored for distributed and critical environments such as healthcare facilities and remote care institutions. Two real-world scenarios were developed and tested:

- **Scenario 1 – Shared Storage High Availability:** A master-slave architecture using Apache ActiveMQ with shared KahaDB persistence to guarantee uninterrupted message flow during broker failure.
- **Scenario 2 – Static Bridging for Broker Communication:** A unidirectional message bridging mechanism using Eclipse Mosquitto to forward emergency alerts from a remote retirement home to a central hospital system in real time.

Each scenario was validated through live deployment using Python MQTT clients, GUI/web interfaces, and broker restarts to simulate failures. The resulting implementations emphasized automatic failover, transparent message flow, and architectural simplicity, all critical factors in building fault-tolerant IoT systems.

Final Thoughts

In building upon the foundations of MQTT and lightweight messaging protocols, this thesis highlights how practical enhancements from shared persistence to broker bridging can extend the robustness and interoperability of real-time systems. While the scope of implementation was bounded to a local lab environment, the methodology and insights gained are scalable to broader applications in smart cities, e-health, and industrial monitoring.

Future work may explore security integration (TLS, ACLs), dynamic broker discovery, and the use of containerized or cloud-native deployments for scaling across distributed edge networks.

Bibliography

- [1] O. Standard, “Mqtt version 3.1.1,” 2014. [Online]. Available: <https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>
- [2] —, “Mqtt version 5.0,” 2019. [Online]. Available: <https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html>
- [3] HiveMQ, “Mqtt over websockets,” 2023. [Online]. Available: <https://www.hivemq.com/mqtt-over-websockets/>
- [4] A. S. Foundation, “Apache activemq classic – official documentation,” 2024. [Online]. Available: <https://activemq.apache.org/components/classic/>
- [5] “Mosquitto documentation – bridging and websockets,” accessed April 2024. [Online]. Available: <https://mosquitto.org/man/mosquitto.conf.5.html>
- [6] E. Technologies, “Emqx documentation,” 2024. [Online]. Available: <https://www.emqx.io/docs/en/latest/>
- [7] H. GmbH, “Hivemq documentation,” 2024. [Online]. Available: <https://www.hivemq.com/docs/>
- [8] O. Labs, “Vernemq documentation,” 2024. [Online]. Available: <https://docs.vernemq.com/>
- [9] E. Al-Masri, K. R. Kalyanam, J. Batts, J. Kim, S. Singh, T. Vo, and C. Yan, “Investigating messaging protocols for the internet of things (iot),” *IEEE Access*, vol. 8, pp. 94 880–94 911, 2020.
- [10] S. Shapsough, M. Takroui, R. Dhaouadi, and I. A. Zualkernan, “An mqtt-based scalable architecture for remote monitoring and control of large-scale solar photovoltaic systems,” in *Smart Grid and Internet of Things, Social-Informatics and Telecommunications Engineering*. Springer, 2019, vol. 256, pp. 57–67.
- [11] A. S. Foundation, “Activemq classic - shared storage master/slave,” 2024. [Online]. Available: <https://activemq.apache.org/components/classic/documentation/masterslave>

- [12] E. Team, “How mqtt clustering works in emqx,” <https://www.emqx.com/en/blog/how-mqtt-clustering-works-in-emqx>, 2022, accessed: 2024-05-12.
- [13] A. S. Foundation, “Activemq classic - jdbc master/slave persistence,” 2024. [Online]. Available: <https://activemq.apache.org/components/classic/documentation/jdbc-master-slave>
- [14] E. Foundation, “Mosquitto documentation – bridging configuration,” 2024. [Online]. Available: <https://mosquitto.org/man/mosquitto.conf.5.html>
- [15] G. C. H. Licht, *MQTT Essentials: A Lightweight IoT Protocol*. Packt Publishing, 2017.
- [16] M. Ng, “Smart application using mqtt protocol for industrial iot and retail,” *Journal of Science and Technology*, vol. 2, no. 1, pp. 15–27, 2023, [Online]. Available: <https://thesciencebrigade.com/jst/article/view/70>.
- [17] V. John and X. Liu, “A survey of distributed message broker queues,” *arXiv preprint arXiv:1704.00411*, 2017.
- [18] B. Konieczek, M. Rethfeldt, F. Golasowski, and D. Timmermann, “A comparison of mqtt brokers for distributed iot edge computing,” in *Software Architecture*. Springer, 2020, pp. 352–368.
- [19] D. Bandyopadhyay, S. Sengupta, and R. Maiti, “Role of middleware for internet of things: A study,” *International Journal of Computer Science and Engineering Survey*, vol. 2, no. 3, pp. 94–105, 2011.
- [20] M. Al-Khafajiy, T. Baker, M. Asim, A. Waraich, and H. Kolivand, “Remote health monitoring of elderly through wearable sensors and mobile applications,” *Multimedia Tools and Applications*, vol. 79, p. 7731–7755, 2020.
- [21] C. de Souza, J. Filho, and R. Correia, “Mqtt-based monitoring architecture for distributed energy resources,” in *Proceedings of the 2021 IEEE PES Innovative Smart Grid Technologies (ISGT)*, 2021, pp. 1–5.
- [22] J. Sikorski, J. Haughton, and M. Kraft, “Industry 4.0 and industrial internet of things: A systematic literature review,” *International Journal of Production Research*, vol. 60, no. 9, pp. 2632–2652, 2022.

-
- [23] S. Tayeb, A. Bouabdallah, and H. Youssef, “Reliable mqtt-based iot architecture for emergency communication in disaster zones,” *Ad Hoc Networks*, vol. 144, p. 102653, 2023.
- [24] Apache Software Foundation, “KahaDB Persistence Store,”
<https://activemq.apache.org/kahadb>, 2024, accessed: 2025-05-30.
- [25] “Flask documentation,” <https://flask.palletsprojects.com/>, accessed: 2025-05-20.
- [26] “Flask-socketio documentation,” <https://flask-socketio.readthedocs.io/>, accessed: 2025-05-20.