Optimization of Energy and QoS for IoT devices running MQTT

by

Chineme Uba

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
(Artificial Intelligence)
in the University of Michigan-Dearborn
2026

Master's Thesis Committee:

Assitant Professor Zheng Song, Chair
Assitant Professor Srijita Das,
Assitant Professor Ang Li


DEARBORN

Chineme Uba
cduba@umich.edu
ORCID iD: 0009-0005-8369-3686

© Chineme Uba 2026

DEDICATION

This thesis is dedicated to my parents, family, advisor, and friends for their unwavering support and sacrifice. To those who believed in me even when I doubted myself, and those who encouraged me to keep going when this seemed impossible.

ACKNOWLEDGEMENTS

I would like to express my sincere and profound gratitude to my advisor, Dr. Zheng Song, for his exceptional guidance, patience, and unwavering support throughout the entirety of this research. His extensive knowledge, thoughtful feedback, and consistent encouragement were invaluable in shaping the direction, depth, and rigor of this work. Dr. Song's mentorship not only contributed significantly to the successful completion of this thesis but also played a vital role in my academic growth and development as a researcher.

I am also sincerely grateful to Ang Li and Srijita Das for their time, insightful feedback, and constructive suggestions. Their careful review and thoughtful perspectives significantly strengthened this research and enhanced the overall quality of the final manuscript. I deeply appreciate their generosity in sharing their expertise and their guidance at critical stages of this work.

I would like to acknowledge the Department of Computer Science at the University of Michigan–Dearborn for providing the academic resources, facilities, and supportive environment necessary to complete this research. The department's commitment to academic excellence, along with the knowledge and training gained through coursework and research opportunities, contributed greatly to my intellectual development and the successful completion of this thesis.

Finally, I extend my deepest gratitude to my family and friends for their unwavering encouragement, understanding, and constant support throughout this journey. Their patience, motivation, and belief in me provided the strength and perseverance needed to overcome challenges and remain focused during demanding times. This accomplishment would not have been possible without their continued support and reassurance.

PREFACE

This thesis is based on research originally presented in the paper entitled *"Smart-MQTT: QoS-Adaptive MQTT Middleware for Energy-Efficient IoT Systems,"* authored by Chineme Uba, Salatiel Martinez, Nico Bokhari, Zhengquan Li, Ang Li, and Zheng Song.

The author of this thesis contributed to the design, implementation, and evaluation of the Smart-MQTT system. Specifically, the author was responsible for the development of the publisher-side and broker-side middleware components, the formulation and implementation of the task allocation model, and the execution and analysis of experimental evaluations conducted on both a real-world IoT testbed and in large-scale simulation environments. Additional contributions include the interpretation of results and the preparation of technical documentation describing the system architecture and optimization methodology.

This thesis expands upon the original publication by providing a more comprehensive discussion of the system design rationale, a detailed explanation of the optimization model, and an in-depth analysis of experimental results. Furthermore, this document includes extended background material and a broader review of related work to place Smart-MQTT within the context of existing research on energy-efficient IoT systems and MQTT-based communication frameworks.

All content presented in this thesis has been written to satisfy the requirements of the Master's degree program and reflects the author's independent academic work, except where explicitly noted. Portions of the material included herein may overlap with the previously published paper; however, the presentation, organization, and elaboration of the content have been substantially extended to meet the scope and depth expected of a master's thesis.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

PREFACE iv

Preface . . . . . **iv**

LIST OF FIGURES viii

LIST OF TABLES ix

LIST OF ACRONYMS x

ABSTRACT xi

CHAPTER

1 Introduction **1**

1.1 Background and Motivation 1
1.2 Problem Statement 2
1.3 Research Objectives 3
1.4 Contributions 3
1.5 Thesis Organization 4

2 Background and Related Work **5**

2.1 Internet of Things Communication Model 5
2.2 MQTT Protocol Overview 7
2.3 Energy Consumption Characteristics of IoT Devices 7
2.4 Energy Optimization in MQTT-Based IoT Systems 8
2.4.1 Protocol-Level Optimization 8
2.4.2 Hardware–Software Co-Design 10
2.4.3 Context-Aware and Adaptive Approaches 11
2.5 QoS-Aware Middleware for IoT Systems 13
2.6 Limitations of Existing MQTT QoS Mechanisms 15
2.7 Task Allocation and Optimization in IoT Systems 16
2.8 Summary and Research Gap 17

3 Smart-MQTT System Architecture and Design **19**
3.1 Design Goals and Assumptions 19
3.2 System Architecture Overview 19
3.3 Subscriber QoS Requirement Handling 20
3.4 Publisher-Side Add-On Design 21
3.5 Broker-Side Add-On and Control Flow 21
3.6 System Workflow Example 21
3.6.1 Subscriber QoS Request 22
3.6.2 Publisher Capability Reporting 22
3.6.3 Task Allocation and Control Flow 23
3.7 Chapter Summary 24
4 QoS-Aware Task Allocation Model **25**
4.1 System Model and Definitions 25
4.2 Task Utility and QoS Satisfaction 25
4.3 Optimization Objective 26
4.4 MILP Formulation 26
4.5 Algorithm Rationale and Complexity 27
4.6 Chapter Summary 28
5 System Implementation **29**
5.1 Implementation Overview 29
5.2 Broker-Side Implementation 29
5.3 Publisher-Side Implementation 30
5.4 Communication and Control Flow 30
5.5 Runtime Overhead and Practical Considerations 30
6 Experimental Evaluation **32**
6.1 Evaluation Objectives 32
6.2 Baseline Approaches 32
6.3 Experimental Setup 33
6.3.1 Small-Scale IoT Testbed 33
6.3.2 Large-Scale Simulation 34
6.4 Evaluation Metrics 34
6.4.1 System Lifetime 34
6.4.2 QoS Satisfaction Ratio 35
6.4.3 Execution Latency 35
6.5 Results and Analysis 35
6.5.1 Testbed Results 35
6.5.2 Large-Scale Simulation Results 36
6.6 Discussion 37
6.6.1 Interpretation of Results 37
6.6.2 Overhead Analysis 38
6.6.3 Threats to Validity 38
7 Conclusion and Future Work **39**

7.1 Summary of the Research 39
7.2 Summary of Contributions (*in the first chapter, you summarized the contributions already, this looks duplicated*) 39
7.3 Key Experimental Insights 40
7.4 Limitations 41
7.5 Future Research Directions 41

BIBLIOGRAPHY 43

LIST OF FIGURES

FIGURE

3.1 High-level architecture of the Smart-MQTT system. 20
3.2 Smart-MQTT System Components and Workflow 22
3.3 Subscription Request with QoS Embedded 22
3.4 SmartMQTT-Device-Status Message 23

6.1 Smart-MQTT testbed deployment. 33
6.2 Performance comparison on physical testbed. 35
6.3 Performance under varying system scale and α. 37

LIST OF TABLES

TABLE

6.1 Default parameter settings for real-world testbed. 34
6.2 Execution latency per algorithm. 36

LIST OF ACRONYMS

AI Artificial Intelligence

IoT Internet of Things

MILP Mixed Integer Linear Programming

CoAP Constrained Application Protocol

QoS Quality of Service

HVAC Heating, Ventilation, and Air Conditioning

UDP User Datagram Protocol

MAC Media Access Control

MQTT Message Queuing Telemetry Transport

ABSTRACT

The Message Queuing Telemetry Transport (MQTT) protocol is widely used in Internet of Things (IoT) systems due to its lightweight design and energy efficiency. In typical MQTT-based deployments, data-producing IoT devices publish sensor data to a broker, which forwards the data to subscribing applications. While MQTT provides basic message delivery guarantees, it lacks mechanisms for expressing and enforcing application-level Quality of Service (QoS) requirements such as sensing frequency, accuracy, and latency. As a result, IoT devices often continue sensing and publishing data at unnecessarily high quality levels, leading to excessive energy consumption and reduced system lifetime.

This thesis presents *Smart-MQTT*, a lightweight middleware framework that improves energy efficiency in MQTT-based IoT systems by aligning sensing behavior with application QoS requirements. Smart-MQTT operates as an add-on to existing MQTT components, requiring minimal modifications while maintaining compatibility with standard MQTT clients and brokers. Applications express QoS requirements through subscription requests, while IoT devices periodically report their sensing capabilities and energy status. Based on this information, Smart-MQTT dynamically assigns sensing tasks to devices using a QoS-aware and energy-efficient task allocation strategy.

The task allocation problem is formulated as a multi-objective optimization model that jointly considers system lifetime and QoS satisfaction. A task utility metric is introduced to quantify the degree to which device capabilities meet application requirements, and a Mixed Integer Linear Programming (MILP) approach is employed to compute efficient task assignments with practical runtime overhead. The proposed system is implemented using a modified MQTT broker and lightweight publisher-side add-ons.

Experimental evaluation on a real-world IoT testbed and through large-scale simulations demonstrates that Smart-MQTT significantly extends system lifetime while maintaining high QoS satisfaction. Compared to baseline MQTT-based approaches, Smart-MQTT achieves substantial energy savings with modest execution latency and communication overhead, highlighting its effectiveness as a practical and scalable solution for energy-constrained IoT environments.

CHAPTER 1

Introduction

1.1 Background and Motivation

The rapid growth of the Internet of Things (IoT) has led to the widespread deployment of interconnected sensing and actuating devices across diverse domains, including smart homes, healthcare, industrial automation, and environmental monitoring [2]. These systems rely on large numbers of resource-constrained devices that continuously collect and transmit data to applications running at the edge or in the cloud. As the scale and complexity of IoT deployments increase, energy efficiency has become a critical concern, particularly for battery-powered devices that are expected to operate autonomously for extended periods of time.

Among the communication protocols designed for IoT environments, Message Queuing Telemetry Transport (MQTT) has emerged as one of the most widely adopted solutions. MQTT is a lightweight, application-layer publish–subscribe protocol specifically designed to minimize communication overhead and energy consumption [16]. Its simplicity, low bandwidth requirements, and support for unreliable networks make it well suited for energy-constrained IoT devices [20]. In a typical MQTT system, sensor-equipped IoT devices act as publishers that transmit data to a central broker, while applications subscribe to specific topics to receive the published data.

Although MQTT is inherently more energy-efficient than many alternative protocols, energy waste remains prevalent in real-world deployments. In practice, Internet of Things (IoT) devices often continue to sense and publish data at fixed or unnecessarily high quality levels, regardless of the actual needs of data-consuming applications. This behavior leads to unnecessary sensing operations and transmissions that can significantly shorten device lifetimes and reduce the overall sustainability of IoT systems.

The root cause of this inefficiency lies in a structural disconnect between application-level Quality of Service (QoS) requirements and the sensing behavior of IoT devices. While

applications may have explicit requirements regarding data frequency, latency, reliability, or accuracy, these requirements are not effectively communicated to or enforced by data publishers. As a result, devices frequently produce data at quality levels that exceed actual application needs, consuming energy without providing commensurate utility. Addressing this mismatch represents a significant and largely unexplored opportunity to extend the operational lifetime of IoT deployments.

1.2 Problem Statement

Despite its widespread adoption, the current MQTT protocol provides limited support for application-level QoS management. MQTT defines a built-in QoS parameter [22] that governs message delivery guarantees—such as "at most once," "at least once," and "exactly once"—but this mechanism only addresses communication reliability. It does not capture broader QoS dimensions that are critical to many IoT applications, including sensing frequency, data accuracy, or end-to-end latency.

In existing MQTT-based systems, subscribing applications have no standardized way to express their QoS requirements to data-publishing IoT devices. Consequently, publishers are unaware of how their sensing behavior impacts application utility and continue operating according to static configurations or worst-case assumptions. This lack of coordination leads to redundant data production, excessive sensing frequency, and unnecessary energy consumption, particularly in environments where multiple devices can perform similar sensing tasks.

Furthermore, prior research on energy optimization in MQTT-enabled IoT systems has largely focused on communication-layer improvements, hardware–software co-design, or subscriber-side adaptations [28, 32, 24]. While these approaches offer valuable energy savings, they overlook the significant potential for optimization at the publisher side—specifically, by dynamically controlling which devices perform sensing tasks and at what quality levels based on application needs and device energy status.

The core problem addressed in this thesis is therefore the absence of a backward-compatible mechanism in MQTT systems that enables energy-efficient, QoS-aware coordination between data-consuming applications and data-producing IoT devices [22]. Without such a mechanism, IoT systems cannot systematically balance application QoS requirements against device energy constraints, resulting in suboptimal system lifetime and inefficient resource utilization. The following section defines the research objectives that guide this work.

1.3 Research Objectives

The primary objective of this thesis is to design, implement, and evaluate a middleware solution that improves the energy efficiency of MQTT-based IoT systems by aligning sensing behavior with application-level QoS requirements. Specifically, this thesis aims to:

- Design a QoS-aware MQTT middleware that enables applications to express sensing requirements without modifying the core MQTT protocol.
- Develop an energy-efficient task allocation mechanism that dynamically assigns sensing tasks to IoT devices based on both QoS satisfaction and device energy status.
- Formulate the task allocation problem as an optimization model that balances system lifetime and application QoS using a unified objective function.
- Implement the proposed solution in a real MQTT-based system while maintaining compatibility with existing MQTT clients and brokers.
- Evaluate the effectiveness and overhead of the proposed approach through real-world testbed experiments and large-scale simulations.

By achieving these objectives, this thesis seeks to demonstrate that intelligent, publisher-side QoS-aware control can significantly extend the operational lifetime of IoT systems while maintaining high levels of application satisfaction.

1.4 Contributions

This thesis makes the following key contributions to the design and optimization of energy-efficient IoT systems based on the MQTT protocol:

1. **QoS-Aware MQTT Middleware Design.** This thesis presents *Smart-MQTT*, a lightweight middleware framework that enables applications to express application-level Quality of Service (QoS) requirements—such as sensing frequency, accuracy, and reliability—without modifying the core MQTT protocol. By operating as an add-on to existing MQTT components, Smart-MQTT preserves backward compatibility while introducing intelligent control over sensing behavior.
2. **Publisher-Side Energy Optimization.** Unlike prior work that primarily focuses on communication efficiency or subscriber-side adaptations, this thesis explicitly targets energy optimization at the publisher side. Smart-MQTT dynamically controls

which IoT devices perform sensing tasks and at what quality levels, thereby reducing redundant sensing operations and unnecessary energy consumption.

3. **QoS-Aware Task Allocation Model.** This thesis formulates the sensing task allocation problem as a multi-objective optimization problem that jointly considers system lifetime and QoS satisfaction. A task utility metric is introduced to quantify the degree to which application QoS requirements are met, enabling a principled trade-off between energy efficiency and service quality.

4. **MILP-Based Optimization Algorithm.** To efficiently solve the task allocation problem, this work adopts a Mixed Integer Linear Programming (MILP) formulation. The proposed approach avoids the combinatorial explosion of brute-force methods and produces high-quality task assignments with practical computational overhead.

5. **Prototype Implementation and Experimental Evaluation.** Smart-MQTT is implemented using a modified MQTT broker and lightweight publisher-side add-ons. The system is evaluated through real-world testbed experiments and large-scale simulations, demonstrating significant improvements in system lifetime while maintaining high QoS satisfaction and acceptable runtime overhead.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides the technical background necessary to understand the design and evaluation of Smart-MQTT, surveys related work on energy optimization and QoS management in MQTT-based IoT systems, and identifies the research gap that motivates this work. Chapter 3 presents the Smart-MQTT system architecture and design, describing the subscriber QoS specification mechanism, the publisher-side add-on, the broker-side control logic, and the system workflow. Chapter 4 formalizes the QoS-aware sensing task allocation problem as a multi-objective optimization model and describes the Mixed Integer Programming formulation used to solve it. Chapter 5 details the prototype implementation of the Smart-MQTT framework, covering the broker-side and publisher-side components, control message design, and practical deployment considerations. Chapter 6 presents the experimental evaluation, including results from both a real-world IoT testbed and large-scale simulations, with analysis of system lifetime, QoS satisfaction, and computational overhead. Chapter 7 summarizes the contributions of this thesis, reflects on its limitations, and outlines directions for future research.

CHAPTER 2

Background and Related Work

This chapter provides the technical background necessary to understand the design motivations of Smart-MQTT and surveys existing research related to energy efficiency and Quality of Service (QoS) management in MQTT-based IoT systems. The chapter begins with an overview of IoT communication models and the MQTT protocol, followed by a discussion of energy consumption characteristics in IoT devices. It then reviews prior work on energy optimization in MQTT systems and QoS-aware middleware, and concludes by identifying the research gap addressed by this thesis.

2.1 Internet of Things Communication Model

The Internet of Things (IoT) refers to large-scale distributed systems composed of interconnected sensing, computing, and actuating devices that interact with the physical environment. These devices produce data, data producers, from their surroundings, process it locally or remotely, and may trigger actions based on observed conditions. IoT systems are deployed across diverse domains, including smart homes, industrial automation, healthcare, precision agriculture, environmental monitoring, and smart cities.

In a smart home environment, for example, temperature sensors monitor indoor climate, motion detectors detect occupancy, and smart thermostats adjust heating or cooling accordingly. In agriculture, soil moisture sensors measure hydration levels to optimize irrigation. In industrial settings, vibration and temperature sensors monitor machinery health to predict failures. In healthcare, wearable devices track heart rate, oxygen saturation, or activity levels and transmit data to remote monitoring platforms. These examples illustrate how IoT systems bridge the physical and digital worlds by continuously sensing real-world phenomena and enabling data-driven decision-making.

Despite their wide applicability, IoT devices typically operate under strict resource constraints. Many devices are battery-powered and must operate for months or years without maintenance. Their microcontrollers provide limited computational power and memory,

and they frequently rely on low-bandwidth wireless communication technologies such as IEEE 802.15.4, Bluetooth Low Energy (BLE), or LoRaWAN. Since wireless communication is significantly more energy-intensive than local computation, IoT system design often prioritizes lightweight communication protocols, such as MQTT and CoAP, along with decentralized data collection strategies to minimize communication overhead and prolong device lifetime [16, 20, 21, 31].

A common architectural pattern in IoT systems is the separation between *data producers* and *data consumers*. Data producers are sensor-equipped devices that generate raw or partially processed data. Examples of data producers include: a temperature sensor node in a building automation system, a wearable fitness tracker measuring heart rate, and a soil moisture sensor in a smart irrigation network.

Data consumers, on the other hand, are applications or services that process, analyze, or visualize the data generated by producers. These consumers may run on edge gateways, fog nodes, or cloud platforms. Examples of data consumers include: a building energy management system optimizing HVAC operation, a mobile health application analyzing patient vital signs, and a cloud-based analytics dashboard for smart agriculture.

This separation between data producers and data consumers enables scalable data processing and flexible application development. Multiple data consumers may subscribe to the same data stream, and data producers may serve multiple applications simultaneously. However, this architectural decoupling introduces challenges related to efficient communication, synchronization, quality-of-service differentiation, and energy management.

In many deployments, multiple devices are capable of producing similar or redundant data. For example, several temperature sensors may be deployed within the same building zone, or multiple air quality sensors may overlap in coverage. Similarly, multiple applications may consume the same data stream but with different requirements. A real-time control application may require high-frequency updates with strict latency constraints, whereas a historical analytics application may only need periodic summaries. Without proper coordination, such scenarios can lead to inefficient system behavior. Consider this example of an inefficient data producer and an inefficient data consumer.

Suppose a battery-powered soil moisture sensor is configured to transmit readings every second, regardless of environmental stability. Meanwhile, the cloud-based irrigation management application only evaluates soil conditions once every 30 minutes. In this case, the data producer generates high-frequency data that the consumer neither requires nor processes in real time. The result is excessive wireless transmissions, increased network congestion, and rapid battery depletion — all without improving application performance. An even more problematic scenario occurs when multiple nearby sensors independently transmit identical

environmental readings to a cloud server, and multiple consumer applications separately request the same data stream without aggregation. This redundancy results in duplicated sensing, unnecessary communication overhead, increased energy consumption, and reduced system lifetime.

Efficient coordination between data producers and data consumers is therefore essential to avoid redundant sensing and unnecessary communication. Mechanisms such as adaptive sampling, context-aware data reporting, task allocation strategies, and in-network aggregation can significantly reduce energy consumption while maintaining application-level performance requirements. By intelligently aligning sensing frequency, communication patterns, and consumer needs, IoT systems can achieve improved sustainability, scalability, and operational efficiency.

2.2 MQTT Protocol Overview

Message Queuing Telemetry Transport (MQTT) is a lightweight, application-layer publish–subscribe protocol specifically designed for resource-constrained environments. In MQTT, publishers send messages to a central broker under specific topics, while subscribers express interest in those topics. The broker is responsible for managing subscriptions and forwarding messages accordingly.

MQTT offers several advantages for IoT systems, including low protocol overhead, minimal message headers, and asynchronous communication [22, 20]. These features make it particularly suitable for low-power devices and unreliable networks [16, 20]. MQTT clients and brokers are widely available through mature implementations such as Eclipse Paho [8], Mosquitto [19], HiveMQ [10, 11], and MQTT.js [6].

The protocol defines three levels of Quality of Service (QoS) that govern message delivery reliability: at most once, at least once, and exactly once [22]. While these QoS levels provide basic delivery guarantees, they do not capture broader application-level requirements such as sensing frequency, latency constraints, or data accuracy [22]. Consequently, MQTT's built-in QoS mechanism is insufficient for expressing the diverse and application-specific requirements commonly found in IoT deployments.

2.3 Energy Consumption Characteristics of IoT Devices

Energy efficiency is a primary concern in IoT systems, particularly for battery-powered devices deployed in remote or inaccessible environments [20, 16]. The energy consumption of an IoT device is typically dominated by three components: sensing operations, computation,

and communication [31, 4].

While communication is often considered the primary energy cost, sensing operations themselves can also be energy-intensive, especially for high-frequency or high-precision sensors such as thermal cameras, motion detectors, or environmental sensors [5]. In addition, redundant sensing—where multiple devices collect identical data for the same application—can significantly reduce system lifetime without providing additional utility [5].

Energy consumption in IoT systems is further influenced by sensing frequency and data quality [31, 17]. Higher sampling rates, increased accuracy, and stricter reliability requirements generally lead to higher energy usage [31, 17]. Therefore, aligning sensing behavior with actual application needs is critical for achieving energy-efficient operation.

2.4 Energy Optimization in MQTT-Based IoT Systems

Prior research on improving energy efficiency in MQTT-enabled IoT systems can be broadly categorized into three principal research directions: protocol-level optimization, hardware–software co-design, and context-aware adaptation. Each of these approaches addresses a distinct layer of the IoT stack and contributes valuable insights into energy-aware system design. However, as discussed in detail below, these approaches typically focus on localized optimizations rather than holistic coordination between application-level QoS requirements and publisher-side sensing behavior.

2.4.1 Protocol-Level Optimization

The MQTT protocol defines how publishers and subscribers exchange information within a distributed IoT system [22]. It specifies the rules governing message formatting, session establishment, reliability guarantees, and topic-based data routing through a centralized broker [22]. Protocol-level optimizations seek to reduce energy consumption by modifying, extending, or reconfiguring the underlying MQTT communication stack.

The MQTT communication stack refers to the layered set of protocols that collectively enable data transmission from one device to another. Rather than operating independently, MQTT relies on multiple networking layers, each responsible for a specific function in the communication process. A typical MQTT stack consists of the application layer, the MQTT protocol layer, the transport layer, the network layer, the link layer, and the physical layer [22, 20]. Each layer introduces processing and communication overhead, thereby providing potential opportunities for energy optimization [31, 4].

The application layer runs directly on the IoT device and is responsible for sensing or

collecting data from the physical environment [22]. It determines when data should be transmitted and forwards the sensed information to the MQTT protocol layer [22].

The MQTT protocol layer formats the data according to MQTT specifications. It defines packet types (e.g., CONNECT, PUBLISH, SUBSCRIBE), manages the topic namespace, enforces Quality of Service (QoS) rules, handles keep-alive intervals, and maintains session state [22]. This layer encapsulates application data into MQTT control packets composed of a fixed header, variable header, and payload, which are then transmitted to the broker for routing [22].

The transport layer is responsible for communication reliability [15]. In standard deployments, MQTT operates over TCP. The transport layer manages connection establishment, ordered delivery, retransmission of lost packets, congestion control, and flow control [15]. Although these mechanisms enhance robustness, they introduce additional energy overhead due to acknowledgment packets, retransmissions [15], and persistent connection maintenance. Consequently, the transport layer is often a significant contributor to communication-related energy consumption [15, 4].

The network layer, typically implemented using IP, determines how packets are routed from the sensor node to the broker across interconnected networks and routers [20]. Beneath it, the link layer manages local frame transmission and medium access control, while the physical layer handles the actual transmission of electrical or radio signals [28]. These lower layers directly influence radio duty cycling and transmission power, which are critical factors in battery-powered wireless sensor networks [31, 4].

Because communication overhead is traditionally viewed as a dominant source of energy consumption in wireless sensor networks, many early optimization efforts focused on reducing transmission cost and improving network efficiency, particularly at the protocol and transport layers [20, 31]. By minimizing control traffic, reducing packet size, or altering transport behavior, protocol-level optimizations aim to decrease radio usage and extend device lifetime [4, 12].

One common strategy involves replacing the standard TCP in the transport layer with lighter alternatives. For example, MQTT-SN (MQTT for Sensor Networks) adapts MQTT to operate over UDP, thereby eliminating the overhead associated(3-way handshake) with TCP connection establishment, congestion control, and reliability management [14]. By minimizing packet headers and handshake procedures, such approaches reduce communication latency and energy expenditure [4], particularly in lossy or bandwidth-constrained networks.

Additional protocol-level strategies include:

- Reducing acknowledgment traffic through selective confirmation mechanisms [4].

- Aggregating multiple data samples into a single publish message [15].
- Optimizing keep-alive intervals to reduce unnecessary heartbeat transmissions [12].

These techniques are effective in scenarios where communication dominates energy consumption. However, they implicitly assume that sensing behavior remains fixed. In many practical IoT deployments, sensing operations themselves consume significant energy, especially when involving high-frequency sampling or computationally intensive sensors such as imaging or environmental monitoring modules [5].

Moreover, protocol-level optimizations often require modifications to protocol semantics, custom broker implementations, or changes to transport-layer behavior. Such requirements may reduce backward compatibility and complicate deployment in existing infrastructures. Consequently, while protocol-level improvements enhance communication efficiency, they do not directly address the coordination of sensing workloads across multiple publishers.

2.4.2 Hardware–Software Co-Design

Hardware–software co-design primarily operates at the lower layers of the communication stack, particularly at the physical layer, link layer, and application layer, while leveraging interactions across layers through cross-layer optimization [28, 12].

At the physical layer, energy efficiency techniques such as dynamic voltage and frequency scaling (DVFS), transmission power control, and radio duty cycling directly reduce the energy consumed during signal transmission and processing [28]. Since radio transmission is one of the most energy-intensive operations in wireless sensor nodes, physical-layer control mechanisms play a central role in extending battery lifetime [4, 31].

At the link layer, mechanisms such as Medium Access Control (MAC) scheduling, low-power listening, time synchronization, and coordinated sleep scheduling reduce idle listening and packet collisions [28]. By carefully managing when the radio is active, these techniques significantly decrease unnecessary energy expenditure caused by channel contention and overhearing [28].

At the application layer, sensing intervals, data sampling frequency, and local processing decisions are adjusted to balance sensing accuracy with energy consumption [12]. Application-driven scheduling determines when data should be generated, which directly influences how often lower-layer communication mechanisms are triggered [12].

Cross-layer optimization [28] connects these layers by coordinating sensing, processing, and communication phases. For example, application-layer data generation may be synchronized with MAC-layer wake-up schedules to prevent redundant radio activations. This

integration ensures that devices wake, sense, process, and transmit in tightly controlled intervals, thereby minimizing idle energy consumption [12].

Importantly, hardware–software co-design generally operates at the individual device level [28, 12]. Each node independently optimizes its own physical, link, and application-layer behavior without global awareness of system-wide task allocation or neighboring device capabilities [5]. Consequently, while such approaches significantly improve local energy efficiency, they do not inherently provide mechanisms for coordinated sensing or workload distribution across multiple devices in a distributed IoT system [5].

2.4.3 Context-Aware and Adaptive Approaches

Context-aware optimization represents a dynamic class of energy-aware strategies that adapt system behavior based on runtime conditions such as network congestion, workload fluctuations, environmental variability, or evolving application requirements [9]. Unlike static protocol tuning or fixed duty-cycling strategies, context-aware approaches introduce feedback mechanisms that allow different layers of the communication stack to adjust their behavior in response to observed operating conditions [9, 23].

Within the MQTT communication stack, context-aware adaptations can occur at multiple layers [9, 23].

At the **application layer**, sensing frequency and data reporting intervals may be adjusted based on environmental stability or event detection. For example, if measured values remain within a stable range, the system may reduce sampling frequency to conserve energy. Conversely, sudden environmental changes may trigger higher sampling or immediate reporting.

At the **MQTT protocol layer**, adaptive mechanisms may dynamically select appropriate Quality of Service (QoS) levels depending on data criticality, reliability requirements, or network conditions. Less critical data may be transmitted using QoS 0 to reduce acknowledgment overhead, while important alerts may use higher QoS levels to guarantee delivery.

At the **transport and network layers**, context-aware strategies may include congestion-aware transmission control, dynamic broker placement [13], broker clustering, or load balancing across distributed brokers to reduce latency and minimize retransmissions. By optimizing routing paths or redistributing traffic, these methods aim to decrease communication overhead and improve overall system responsiveness [13, 27].

At the **link and physical layers**, adaptive transmission power control or channel selection may be employed in response to interference levels or signal quality, thereby reducing retransmissions and conserving radio energy.

In MQTT-based systems, representative context-aware techniques include:

- Adaptive selection of MQTT delivery QoS levels [31, 33].
- Dynamic broker placement or clustering to reduce latency and transmission distance [13].
- Load balancing across distributed brokers [27, 25].
- Adjustment of sensing or reporting frequency based on detected environmental stability [23, 9].

These mechanisms introduce runtime adaptability, enabling systems to respond to changing operational conditions across multiple layers of the communication stack [9, 23]. Performance evaluation studies of publish-subscribe systems in IoT have further demonstrated that the energy savings achievable through context-aware QoS adaptation are sensitive to the degree of coordination between application requirements and device behavior [9]. However, many existing approaches primarily emphasize communication-layer optimization or subscriber-side adaptations [5]. In numerous deployments, publishers continue sensing at fixed or maximum capability levels regardless of the actual utility, redundancy, or system-level necessity of the generated data [5]. Recent work has begun exploring sensing-task aggregation and coordination at the device level to improve energy efficiency [5], but such approaches remain limited in their ability to incorporate dynamic application-level QoS requirements and system-wide coordination.

An emerging class of context-aware approaches adopts adaptive or event-driven sensing strategies, in which devices dynamically adjust their sampling frequency in response to environmental changes or detected events [23, 9]. In threshold-based approaches, a device reduces its reporting rate when measured values remain stable and increases sampling upon detecting significant changes [23]. While these per-device strategies improve individual energy efficiency, they do not account for redundant coverage across multiple nearby devices or for system-wide application QoS requirements [5]. Coordinating adaptive sensing across multiple publishers in response to evolving subscriber demands remains an open challenge that motivates the centralized coordination approach proposed in this thesis.

This implicit assumption of static or independently adaptive publisher behavior constrains the achievable energy savings. While adaptive transmission strategies reduce network overhead and communication cost, they do not eliminate redundant sensing operations at the source [5]. Since sensing, sampling, and local processing themselves consume non-trivial energy—particularly in high-frequency or computationally intensive sensors—failure to coordinate sensing behavior at the system level limits overall efficiency [5].

Consequently, context-aware methods partially address energy optimization across the MQTT stack but often leave significant untapped potential at the application-layer sensing stage, where system-wide coordination could further reduce redundant data generation [13] and uneven energy depletion.

2.5 QoS-Aware Middleware for IoT Systems

QoS-aware middleware seeks to bridge the semantic gap between application-level requirements and underlying system behavior [27]. Rather than treating the network as a passive message relay, middleware introduces an intermediate abstraction layer that translates high-level performance objectives—such as latency, reliability, throughput, or priority—into concrete operational decisions across the communication stack [27].

Conceptually, middleware operates between the application layer and the MQTT protocol layer, while exerting indirect influence over transport, network, and broker-level behaviors. It does not replace the MQTT protocol itself, but instead orchestrates how the protocol and lower layers are utilized in order to meet specified Quality of Service (QoS) constraints.

In general distributed systems, QoS-aware middleware may regulate bandwidth allocation at the network layer, enforce latency constraints through transport-layer configuration, or manage priority scheduling at the application layer [27, 7]. In IoT environments, middleware solutions have been proposed to coordinate data aggregation at the application layer, enforce reliability policies at the MQTT protocol layer, and adapt delivery guarantees by dynamically selecting QoS levels. For example, EMMA [27] introduces a distributed MQTT middleware architecture designed for edge computing environments, enabling QoS-aware message processing and scalable data distribution across edge nodes. Such systems demonstrate how middleware can enhance QoS management without modifying the underlying MQTT protocol semantics, instead introducing orchestration mechanisms at the middleware and broker layers.

Within MQTT-based IoT systems specifically, existing QoS-aware middleware solutions typically focus on:

- Enhancing message delivery reliability through dynamic QoS selection at the MQTT protocol layer [23].
- Prioritizing subscriber requests via broker-side scheduling mechanisms [7].
- Performing load balancing across distributed brokers at the network layer [27, 25].

- Managing topic-based access control and authorization policies at the application and broker layers [18].

Representative examples illustrate the range of these approaches. Palmese et al. [23] propose an adaptive QoS control mechanism for MQTT-SN that dynamically adjusts the delivery QoS level based on observed network conditions and reliability requirements, reducing unnecessary acknowledgment traffic while preserving message delivery guarantees. De Caldas Filho et al. [7] present a QoS scheduling algorithm for fog IoT gateways that prioritizes message handling based on subscriber requirements and available gateway resources, improving responsiveness for latency-sensitive applications. La Marra et al. [18] introduce usage control mechanisms into the MQTT broker, enabling policy-based enforcement of data access and quality constraints without modifying the core protocol semantics. In the context of mobile crowdsensing, Podnar Žarko et al. [26] demonstrate how publish-subscribe middleware-level coordination can reduce energy consumption while maintaining service quality, highlighting the potential of middleware-driven approaches to bridge application requirements and device behavior.

Through these mechanisms, middleware primarily influences the MQTT protocol layer and broker infrastructure, with secondary effects on transport-layer performance and network routing efficiency [27, 23]. As a result, application-facing performance metrics such as latency, reliability, and fairness are improved without requiring direct modification of the underlying hardware [27, 7].

However, most existing QoS-aware middleware architectures exert little to no control over publisher-side sensing behavior at the application layer [5]. In typical deployments, sensing configurations—including sampling frequency, sensing modality, and reporting intervals—are statically defined during device setup [20, 5]. Even when subscriber demand decreases, reliability constraints are relaxed, or environmental stability reduces the need for high-frequency updates, publishers often continue operating at fixed sensing rates [5].

This absence of feedback between subscriber requirements and publisher sensing behavior creates structural inefficiencies across the stack. While middleware may successfully optimize communication-layer performance and broker-level resource allocation, it does not necessarily reduce energy expenditure at the source of data generation.

From an energy perspective, this disconnect results in excessive sensing frequency, redundant measurements across multiple capable devices, unnecessary packet generation at the MQTT layer, and accelerated battery depletion at the physical layer [5, 31]. Consequently, QoS-aware middleware improves semantic alignment between applications and communication behavior but often stops short of enabling full system-wide coordination of sensing workloads.

2.6 Limitations of Existing MQTT QoS Mechanisms

Despite its widespread adoption in IoT systems, the MQTT protocol provides limited support for application-level Quality of Service (QoS) management [22, 20]. MQTT defines three built-in QoS levels that govern message delivery semantics at the protocol layer: at most once (QoS 0), at least once (QoS 1), and exactly once (QoS 2) [22]. These levels regulate reliability of message transmission between clients and brokers, operating primarily at the MQTT protocol and transport layers of the communication stack [22].

While effective for ensuring delivery guarantees, these QoS levels do not express higher-level service characteristics relevant to IoT sensing applications. In other words, MQTT's QoS mechanisms address *how reliably messages are delivered*, but not *what quality of data is required* at the application layer.

Critical QoS dimensions for IoT applications extend beyond delivery reliability and include:

- Sensing frequency (sampling rate) at the application layer.
- Data accuracy or precision determined by sensor configuration.
- Freshness and latency constraints spanning application, protocol, and network layers.
- Energy-aware prioritization of sensing and transmission tasks.
- Redundancy tolerance across multiple publishers observing the same phenomenon.

These parameters originate at the application layer but influence behavior throughout the communication stack [22]. For example, higher sampling rates increase MQTT publish frequency at the protocol layer, which in turn increases transport-layer acknowledgments, network-layer routing activity, and physical-layer radio usage. Similarly, stricter latency constraints may require higher QoS levels, reduced buffering, or more frequent wake cycles at the link layer.

However, the MQTT standard does not provide mechanisms for encoding, negotiating, or dynamically adapting these application-level QoS parameters between subscribers and publishers [22]. Subscribers can request topic data, but they cannot formally specify sensing requirements such as acceptable sampling intervals or redundancy constraints [22, 5]. Publishers, in turn, lack visibility into how their sensing configuration affects overall application utility or system-wide efficiency.

This architectural separation between application-level quality requirements and protocol-level delivery semantics leads to conservative provisioning strategies. In the absence of

explicit coordination, devices often operate under worst-case assumptions—maintaining high sampling rates, strict reliability levels, and frequent transmissions to ensure that all potential subscribers are satisfied.

From a system perspective, this over-provisioning propagates inefficiencies across the stack. Excessive sensing increases application-layer energy consumption; higher publish rates amplify protocol- and transport-layer overhead; and increased transmission activity accelerates battery depletion at the physical layer. Consequently, while MQTT provides reliable communication primitives, its limited support for application-aware QoS negotiation constrains opportunities for holistic, energy-efficient system optimization.

2.7 Task Allocation and Optimization in IoT Systems

Efficient allocation of sensing responsibilities across multiple heterogeneous IoT devices is fundamentally a combinatorial optimization problem. Given a set of sensing tasks and a set of capable devices, the system must determine which device should perform which task in order to optimize one or more system-level objectives, such as minimizing energy consumption, maximizing QoS satisfaction, or extending device lifetime.

Classical assignment problems have been extensively studied in operations research. The Hungarian algorithm, for example, provides an efficient polynomial-time solution to linear assignment problems, enabling optimal matching between agents and tasks under linear cost structures [29]. However, IoT sensing environments introduce additional complexity: devices are heterogeneous in sensing capability and remaining energy level, tasks may have multiple concurrent QoS dimensions, and the system must jointly satisfy energy and quality constraints simultaneously rather than optimizing a single objective.

Energy-efficient task scheduling has been explored in various IoT contexts. Abdullah and Yang [1] propose an energy-efficient message scheduling algorithm for IoT environments that reduces transmission energy by coordinating message delivery timing across devices. Singh and Das [30] address energy-efficient data offloading for IoT medical devices, demonstrating that task-level assignment decisions significantly influence both energy consumption and application quality. These works collectively show that coordinating task execution across devices yields substantial energy savings compared to independent, uncoordinated operation.

At the broker level, dynamic resource allocation and workload distribution strategies have also been proposed to improve scalability and QoS. Pham et al. [25] introduce a multi-level elasticity mechanism for MQTT brokers that dynamically adjusts broker resources in response to fluctuating workloads, improving system responsiveness under varying subscriber demands. While this approach addresses broker-side scalability rather than publisher-side

sensing coordination, it illustrates the broader benefit of dynamic resource management in MQTT-based systems.

Despite these contributions, prior work on task allocation in IoT systems has not addressed the joint problem of assigning publisher-side sensing tasks to heterogeneous devices under both QoS satisfaction and energy constraints within an MQTT middleware framework. The thesis addresses this gap by formulating the sensing task allocation problem as a Mixed Integer Linear Programming (MILP) model, enabling principled, computationally efficient assignment of sensing responsibilities while balancing application-level QoS utility and device energy levels.

2.8 Summary and Research Gap

The literature reviewed in this chapter demonstrates substantial progress in improving energy efficiency and Quality of Service (QoS) management in MQTT-based IoT systems. Protocol-level optimizations reduce communication overhead at the MQTT and transport layers [4, 15]. Hardware–software co-design enhances device-level efficiency by optimizing physical, link, and application-layer behavior [28, 12]. Context-aware adaptations introduce runtime flexibility across multiple layers of the communication stack [9, 23]. QoS-aware middleware improves subscriber-facing performance by translating high-level requirements into communication-layer adjustments [27, 7].

Despite these advances, these approaches share a fundamental architectural limitation: they do not explicitly coordinate publisher-side sensing behavior based on application-level QoS requirements and real-time device energy status.

Existing solutions either optimize communication after data has already been generated or improve local device efficiency in isolation. Few mechanisms directly address the relationship between *what data is needed*, *which device should produce it*, and *how sensing responsibilities should be distributed* across multiple capable publishers. As a result, sensing decisions at the application layer remain largely static, even as network conditions, subscriber demand, and device energy levels evolve.

This gap manifests in several system-level inefficiencies:

- Subscribers cannot formally express rich QoS requirements beyond message delivery guarantees.
- Publishers lack visibility into application-level quality needs and system-wide redundancy.

- Multiple devices capable of sensing the same phenomenon operate independently, leading to redundant measurements.
- Energy consumption is unevenly distributed, accelerating depletion of heavily utilized nodes and reducing overall system lifetime.

The work most closely related to this thesis is MQTT-EES [5], which proposes aggregating sensing tasks across IoT devices to improve energy efficiency at the device level. However, MQTT-EES addresses task aggregation as a local, device-side optimization and does not incorporate dynamic application-level QoS requirements, subscriber-driven sensing coordination, or system-wide energy balancing across all available publishers. Smart-MQTT extends beyond this approach by introducing an expressive QoS specification mechanism for subscribers, a utility-based task allocation model that jointly considers QoS satisfaction and per-device energy state, and a centralized MILP-based optimization framework that coordinates sensing responsibilities across the entire publisher population.

Consequently, IoT systems struggle to balance application satisfaction with long-term energy sustainability. Publishers continue operating under conservative, worst-case configurations; redundant sensing persists across devices; and battery resources are depleted inefficiently at the physical layer.

The core research problem addressed in this thesis is therefore the absence of a backward-compatible, middleware-based mechanism that enables energy-aware, QoS-driven coordination between data-consuming applications and data-producing IoT devices within MQTT systems.

To address this gap, this thesis proposes *Smart-MQTT*, a middleware framework that integrates expressive application-level QoS specification, publisher-side energy awareness, and centralized sensing task allocation. By aligning sensing behavior with application utility and device constraints, Smart-MQTT enables dynamic workload distribution across publishers, reduces redundant sensing, and balances energy consumption to extend overall system lifetime while maintaining high QoS satisfaction. In doing so, it advances the state of energy-aware IoT system design beyond communication-centric optimization toward system-wide sensing coordination.

CHAPTER 3

Smart-MQTT System Architecture and Design

3.1 Design Goals and Assumptions

The design of Smart-MQTT is guided by several key goals intended to ensure practicality and deployability in real-world IoT systems. First, the system aims to remain fully compatible with existing MQTT implementations. Rather than modifying the MQTT protocol itself, Smart-MQTT operates as a middleware layer that augments standard MQTT components with minimal changes. This design choice allows the proposed system to be integrated into existing deployments without requiring changes to client applications [3] or protocol semantics.

Second, Smart-MQTT is designed to be lightweight and energy-efficient. Any additional communication or computation introduced by the middleware must incur minimal overhead relative to the energy savings achieved through optimized sensing behavior. Third, the system is designed to support dynamic environments in which subscriber requirements and device conditions may change over time. Finally, the system assumes that IoT devices can periodically report basic status information, such as remaining energy and sensing capabilities, to the MQTT broker.

3.2 System Architecture Overview

Figure 3.1 illustrates the high-level architecture of the Smart-MQTT system. The architecture consists of three main components: subscribers (applications), publishers (IoT devices), and an MQTT broker. Smart-MQTT introduces lightweight add-on components at the broker and publisher sides, while leaving subscriber applications largely unmodified.

Subscribers interact with the system through standard MQTT subscription requests. To express application-level QoS requirements, subscribers embed QoS parameters—such as required sensing frequency or reliability—directly into the subscription topic string. On

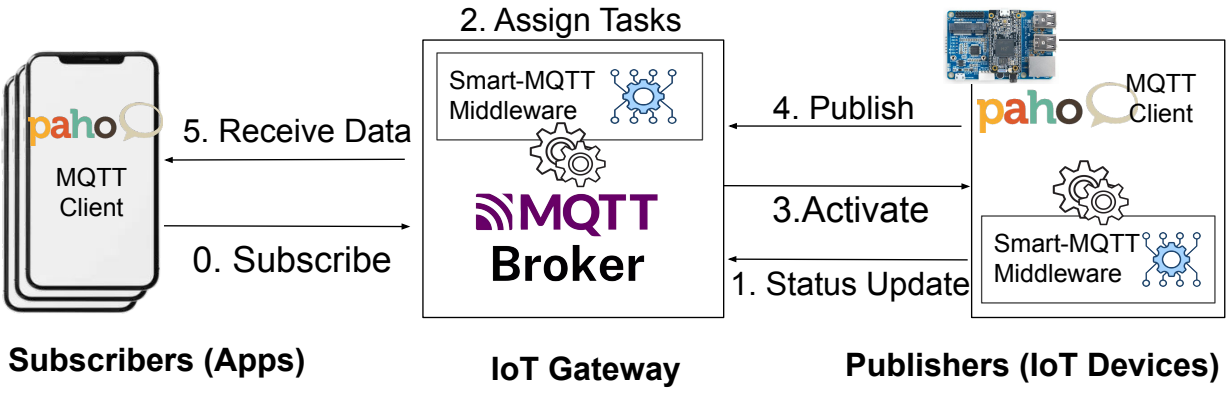


Figure 3.1: High-level architecture of the Smart-MQTT system.

the broker side, a modified subscription handler intercepts these requests, extracts the QoS requirements, and stores them in a local database. The subscription then proceeds through the standard MQTT workflow after the QoS information is removed from the topic string.

On the publisher side, each IoT device runs a Smart-MQTT publisher add-on that interfaces with onboard sensors. This add-on periodically collects device status information, including sensing capabilities, ongoing tasks, and remaining energy. Based on the collected subscriber requirements and device status, the broker executes a task allocation algorithm and instructs selected devices to activate or deactivate specific sensing tasks.

3.3 Subscriber QoS Requirement Handling

Smart-MQTT enables applications to express QoS requirements without modifying the MQTT protocol by embedding QoS constraints into the subscription topic. Upon receiving a subscription request, the broker's modified subscription handler parses the embedded QoS requirements and stores them in a database associated with the subscribed topic.

If multiple subscribers request the same topic with different QoS requirements, the system aggregates these requirements and retains the most stringent constraint for each QoS dimension. This ensures that the delivered sensing data satisfies all subscribing applications. Applications may dynamically update their QoS requirements by issuing new subscription requests, allowing Smart-MQTT to adapt sensing behavior at runtime.

3.4 Publisher-Side Add-On Design

The Smart-MQTT publisher add-on is responsible for monitoring device status and executing sensing tasks assigned by the broker. The add-on interfaces with sensor executables to determine sensing capabilities, such as maximum frequency, accuracy, and energy consumption per sensing operation.

In addition to reporting capabilities, the add-on tracks the device's remaining energy and any ongoing sensing tasks. This information is periodically reported to the broker using side-channel MQTT messages. Upon receiving control messages from the broker, the add-on activates or deactivates specific sensors and adjusts sensing parameters as instructed. Sensing results are published through the standard MQTT publishing mechanism, ensuring compatibility with existing subscribers.

3.5 Broker-Side Add-On and Control Flow

The broker-side Smart-MQTT add-on maintains a lightweight database that stores subscriber QoS requirements and publisher status information. At predefined intervals or upon significant system events, such as a new subscription or device joining the system, the broker triggers the task allocation algorithm.

The resulting task assignments are communicated back to the publishers via control messages. By centralizing decision-making at the broker, Smart-MQTT enables global optimization across devices while keeping individual IoT devices lightweight and simple.

3.6 System Workflow Example

While the previous sections described the individual components of Smart-MQTT, it is useful to illustrate how these components interact during runtime. Figure 3.2 presents the overall workflow of the system, highlighting the interaction between subscribers, the MQTT broker, and IoT publishers equipped with the Smart-MQTT add-on.

The workflow begins when an application subscribes to a sensing topic while embedding QoS requirements in the subscription request. The broker intercepts these requirements and stores them for later task allocation. IoT devices periodically report their sensing capabilities and energy status to the broker. Based on this information, the broker executes the QoS-aware task allocation algorithm and dispatches control messages to the selected devices. The devices then perform the assigned sensing operations and publish the resulting data through the standard MQTT publishing mechanism.

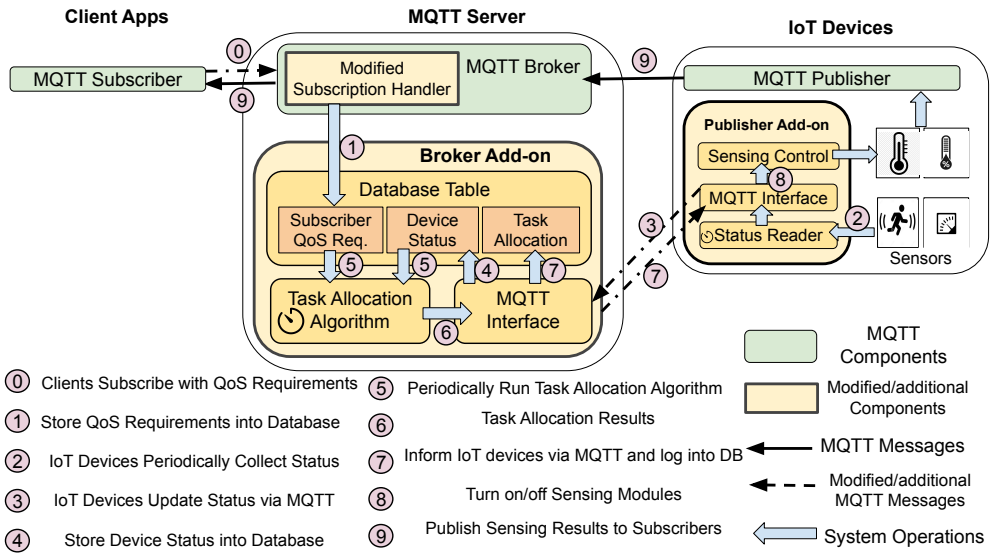


Figure 3.2: Smart-MQTT System Components and Workflow

The following subsections describe each step of the workflow in detail using a temperature sensing example.

3.6.1 Subscriber QoS Request

The system's operation is demonstrated through an example involving temperature sensing. As shown in Step 0, an air-conditioner control application subscribes to the temperature topic for a given room, embedding QoS requirements directly in the subscription string. Upon receiving this request, the broker's modified subscription handler intercepts the QoS requirements, stores them in the internal database, removes them from the subscription string, and proceeds with the standard MQTT subscription process (Step 1).

If the application subsequently needs to update its QoS requirements, it issues a new subscription request; the new QoS values override the previously stored ones, as illustrated in Fig. 3.3.

```
subscribe temp:reliability>=70,frequency=50
```

Figure 3.3: Subscription Request with QoS Embedded

3.6.2 Publisher Capability Reporting

On the IoT device side, the Smart-MQTT publisher add-on periodically queries the executables of all onboard sensors, including the temperature sensor, to retrieve their sensing

capabilities (Step 2).

For example, the executable of a temperature sensor may specify its reliability as “95%”, meaning it successfully returns readings upon request 95% of the time. It may also report its maximum sensing frequency (e.g., 60 Hz) and the energy consumed per sensing operation (e.g., 0.04 W).

The add-on aggregates these capabilities together with information on ongoing sensing tasks and the device's remaining energy. This information is then reported to the broker add-on via side-channel MQTT messages (Step 3). These status updates are transmitted using the same MQTT broker by publishing to the topic `SmartMQTT-Device-Status`, as illustrated in Fig. 3.4.

```
{ "mac": "AA:BB:CC:DD:EE:FF"
  "topics": ["temperature", "humidity"]
  "ongoing_tasks": [
      "temperature"
  ]
  "temperature_qos": {
    "reliability": "95%"
    "accuracy": 0.9
    "max_frequency": 60Hz
    "energy_per_sensing": 0.04W
  }
  "humidity_qos": {
    "reliability": 0.5
    "max_frequency": 30Hz
    "energy_per_sensing": 0.05W
  }
  "current_energy": 77Wh}
```

Figure 3.4: SmartMQTT-Device-Status Message

3.6.3 Task Allocation and Control Flow

On the broker side, the Smart-MQTT add-on receives the device status updates and stores them in its internal database (Step 4). Using the aggregated information from both subscribers and publishers, the broker triggers the task allocation algorithm (Step 5).

The optimization algorithm [29] takes as input the QoS requirements of all active subscribers together with the capability profiles and energy status of all registered IoT devices. It then computes an optimal or near-optimal assignment of sensing tasks to devices, determining which device should perform which task and at what sensing frequency (Step 6).

The resulting task assignments are communicated back to IoT devices via control messages published on the dedicated topic `Smart-MQTT-Control` (Step 7). Upon receiving a control message, the publisher add-on parses the assignment and prepares the relevant sensor executables (Step 8). Once activated, the sensor executables begin generating sensing

data and publishing it to the MQTT broker using the standard MQTT publishing workflow (Step 9). From the perspective of subscriber applications, this process is fully transparent, preserving compatibility with existing MQTT clients.

3.7 Chapter Summary

This chapter has presented the architecture and design of the Smart-MQTT system, describing how subscribers express QoS requirements, how publishers report capabilities and energy status, and how the broker orchestrates sensing task allocation. The design achieves backward compatibility by operating entirely as a middleware overlay on standard MQTT components, with all customization confined to the broker add-on, publisher add-on, and a set of dedicated control topics. The formalization of the task allocation problem underlying Step 6—including the mathematical model, decision variables, and optimization objective—is developed in Chapter 4.

CHAPTER 4

QoS-Aware Task Allocation Model

Chapter 3 described how Smart-MQTT collects QoS requirements from subscribers and capability reports from publishers, and how the broker periodically triggers a task allocation algorithm to determine sensing assignments. This chapter formalizes that allocation problem. The system model, optimization objective, and solution approach are defined in turn, providing the mathematical foundation for the experimental evaluation presented in Chapter 6.

4.1 System Model and Definitions

Let $T = \{t_1, t_2, \ldots, t_m\}$ denote the set of sensing tasks, where each task corresponds to a topic with associated QoS requirements. The QoS requirements for task $t \in T$ are denoted by $Q(t)$, which may include constraints on sensing frequency, reliability, accuracy, or latency.

Let $P = \{p_1, p_2, \ldots, p_n\}$ represent the set of available IoT devices. Each device $p \in P$ has a remaining energy level denoted by $E(p)$. To capture device–task compatibility, a capability matrix C is defined such that

$$c[p][t] = \begin{cases} 1, & \text{if device } p \text{ can perform task } t, \\ 0, & \text{otherwise.} \end{cases}$$

An energy cost matrix $e[p][t]$ represents the energy consumed by device p when executing task t. Additionally, a QoS satisfaction matrix $u[p][t] \in [0, 1]$ quantifies how well device p satisfies the QoS requirements of task t.

4.2 Task Utility and QoS Satisfaction

To evaluate the suitability of assigning a device to a task, this thesis introduces a task utility function. If a device fully satisfies all QoS requirements of a task, the utility is set to one.

Otherwise, the utility decreases as the sensing capability diverges from the required QoS level. This penalty structure reflects the principle that severe QoS violations should be penalized more heavily than minor deviations, while still permitting near-feasible assignments in cases where strict satisfaction is unattainable. The resulting utility values populate the matrix $u[p][t]$ and are used directly in the optimization objective defined in the following section.

4.3 Optimization Objective

The objective of the task allocation algorithm is to maximize the operational lifetime of the IoT system while maintaining high QoS satisfaction. System lifetime is defined as the elapsed time until the system can no longer guarantee sensing coverage. The system remains operaional as long as every active topic has at least one eligible publisher, and it terminates when any topic loses all capable publishers.

To balance energy efficiency and QoS satisfaction, a weighted objective function is employed. A parameter $\alpha \in [0, 1]$ controls the relative importance of QoS satisfaction versus energy preservation. Lower values of α prioritize energy efficiency, while higher values emphasize QoS fulfillment.

4.4 MILP Formulation

Let $A[p][t]$ be a binary decision variable such that

$$A[p][t] = \begin{cases} 1, & \text{if task } t \text{ is assigned to device } p, \\ 0, & \text{otherwise.} \end{cases}$$

Each task must be assigned to exactly one device:

$$\sum_{p \in P} A[p][t] = 1, \quad \forall t \in T.$$

A task may only be assigned to a device capable of performing it:

$$A[p][t] \leq c[p][t], \quad \forall p \in P, \forall t \in T.$$

The optimization objective is defined as:

$$\begin{aligned}\mathcal{A}^* = \arg\max \Bigg(& (1-\alpha) \min_{\forall p \in \mathcal{P}} \frac{E(p)}{\sum_{\forall t \in \mathcal{T}} A[p][t] \cdot e[p][t]} \\ & + \alpha \sum_{\forall t \in \mathcal{T}} u[p][t] \Bigg) \\ \text{s.t.} \quad & \sum_{\forall p \in \mathcal{P}} A[p][t] = 1, \quad \forall t \in \mathcal{T}, \\ & c[p][t] \cdot A[p][t] = 1, \quad \forall t \in \mathcal{T}, \\ & 0 \leq \alpha \leq 1\end{aligned} \tag{4.1}$$

The first term measures how long each publisher can sustain its assigned workload by taking the ratio of its available energy $E(p)$ to the total energy it would spend on its assigned tasks, $\sum_{\forall t \in \mathcal{T}} A[p][t] \cdot e[p][t]$. This captures how long a device can continue operating given the assignment, making it a direct representation of device lifetime. The minimization over all publishers ensures that the assignment favors solutions where no device is assigned a workload that would cause it to deplete energy too quickly. This term introduces a nonlinear dependence on the binary assignment variables $A[p][t]$; in practice, it is typically reformulated (e.g., via variable substitution) so it can be handled by mixed-integer optimization solvers. The second term sums the utility $u[p][t]$ of the assigned tasks, directly capturing the overall quality of service (QoS) provided by the system. It is linear in the decision variables and rewards assignments that yield higher sensing value.

Together, these two terms define a mixed-integer optimization problem that explicitly trades off between preserving node lifetime (by protecting the weakest publisher assigned a sensing task) and maximizing sensing utility, with the balance controlled by the parameter α.

4.5 Algorithm Rationale and Complexity

A brute-force solution to the task allocation problem would require evaluating all possible task–device combinations, resulting in combinatorial complexity that becomes computationally intractable even for modest system sizes. In contrast, the optimization formulation [29] permits the use of branch-and-bound and cutting-plane techniques that systematically reduce the search space without exhaustive enumeration.

In practice, the proposed approach produces high-quality task assignments with computational overhead that remains acceptable for the device and task counts typical of real-world

IoT deployments, as confirmed by the runtime measurements reported in Chapter 6.

4.6 Chapter Summary

This chapter has formalized the sensing task allocation problem as a multi-objective optimization model that jointly maximizes system lifetime and QoS satisfaction. The key components—the device and task model, the task utility function, the weighted objective function, and the binary assignment constraints—together define a complete mathematical specification of the allocation problem. With this model established, Chapter 5 describes how the model is implemented within the Smart-MQTT prototype and how the broker invokes the solver at runtime.

CHAPTER 5

System Implementation

Chapter 3 introduced the architectural design of Smart-MQTT and Chapter 4 formalized the optimization model it relies upon. This chapter bridges design and evaluation by describing the concrete realization of that design in a working prototype. Where Chapter 3 focused on *what* each component does, this chapter addresses *how* it was built: the broker and publisher add-ons, the communication mechanisms between them, and the practical engineering considerations encountered during deployment. This implementation provides the foundation for the experimental evaluation presented in Chapter 6.

5.1 Implementation Overview

The Smart-MQTT prototype consists of two primary add-on modules—a broker-side component and a publisher-side component—layered on top of an unmodified MQTT infrastructure. As shown in Fig. 3.2 in Chapter 3, the unmodified components (subscriber, broker core, and publisher) are retained without change, while the broker add-on and publisher add-on introduce the QoS management and sensing coordination capabilities. The broker add-on maintains a lightweight database storing subscriber QoS requirements and publisher device status, periodically invokes the task allocation algorithm, and dispatches control messages to IoT devices. The publisher add-on queries onboard sensor executables for capability information, reports device status to the broker, and activates or deactivates sensors in response to received control messages. Sensing data produced by activated sensors is forwarded through the standard MQTT publish path, remaining fully transparent to subscribing applications.

5.2 Broker-Side Implementation

The broker-side Smart-MQTT add-on extends a standard MQTT broker by intercepting subscription and control messages. Upon receiving a subscription request, the add-on parses embedded QoS parameters and stores them in a local data structure indexed by topic.

The broker maintains a global view of subscriber QoS requirements and publisher device status. When a triggering event occurs—such as a new subscription, a device joining the system, or a periodic update—the broker invokes the task allocation algorithm described in Chapter 4. The resulting task assignments are translated into control messages and disseminated to the appropriate publishers.

5.3 Publisher-Side Implementation

Each IoT device runs a Smart-MQTT publisher add-on that operates alongside the standard MQTT client shown in Fig. 3.2. This add-on is responsible for monitoring device energy levels, tracking available sensing capabilities, and executing sensing tasks assigned by the broker.

The publisher add-on periodically reports device status information, including remaining energy and active tasks, to the broker. Upon receiving a control message, the add-on activates or deactivates sensors and adjusts sensing parameters accordingly. Sensing data is published using standard MQTT publish operations, ensuring transparency to subscribing applications.

5.4 Communication and Control Flow

Smart-MQTT introduces a small set of auxiliary control messages exchanged between the broker and publishers. These messages are transmitted over dedicated MQTT topics to avoid interference with application data streams.

Three types of auxiliary control messages are exchanged over the dedicated side-channel topics: task activation commands, task deactivation commands, and device status reports. These messages are intentionally kept small in payload size to minimize the communication overhead they introduce relative to the application data streams they coordinate.

Task allocation decisions are performed centrally at the broker, while sensing execution remains decentralized at the device level. This separation enables efficient global optimization while maintaining lightweight device operation.

5.5 Runtime Overhead and Practical Considerations

The computational overhead of Smart-MQTT is primarily concentrated at the broker, where the task allocation algorithm is executed. Publisher-side overhead is limited to simple status reporting and control handling, making the system suitable for resource-constrained devices.

Several practical considerations were addressed during implementation. When a device fails to deliver a status update within the expected reporting window, the broker marks that device as unavailable and excludes it from the next allocation cycle. When a subscriber updates its QoS requirements, the broker treats the event as a reallocation trigger and recomputes assignments using the latest device status. When no feasible assignment exists for a task—because no available device meets the required QoS constraints—the broker retains the previous assignment until a capable device becomes available or requirements are relaxed. If either of that isn't true, then the broker assigns the task to the publisher that can perform the task with the closest QoS.

Taken together, these design choices demonstrate that Smart-MQTT can be integrated into existing MQTT deployments with modest overhead and without requiring any modifications to the underlying protocol or to existing client applications. Chapter 6 evaluates the magnitude of that overhead and its impact on energy efficiency and QoS satisfaction under both small-scale and large-scale operating conditions.

CHAPTER 6

Experimental Evaluation

This chapter evaluates the performance of Smart-MQTT through both real-world testbed experiments and large-scale simulations. The objective is to determine whether the proposed QoS-aware and energy-aware task allocation mechanism improves system lifetime while maintaining high Quality of Service (QoS) satisfaction and acceptable computational overhead.

6.1 Evaluation Objectives

The evaluation is designed to answer the following research questions:

1. Does Smart-MQTT extend the operational lifetime of IoT systems compared to standard MQTT deployments?
2. How effectively does Smart-MQTT satisfy application-level QoS requirements?
3. What computational overhead is introduced by the proposed middleware, and what is the energy cost of periodic status reporting?
4. How does the system scale as the number of devices and tasks increases?

6.2 Baseline Approaches

To comprehensively evaluate Smart-MQTT, we compare it against five representative baselines:

- **Static - Duplicate:** This reflects the original MQTT model, where all IoT devices publish all data they can sense without coordination. This often leads to redundant transmissions and rapid energy depletion.

- **Static - Random:** All sensing tasks are randomly assigned to IoT devices once at system startup without considering QoS requirements or device energy levels. No dynamic reassignment occurs.
- **Static – High-QoS:** Tasks are assigned only once at startup to devices offering the highest QoS (Pareto-optimal QoS), without considering energy balance or dynamic adaptation.
- **Dynamic – Random:** Tasks are reassigned when subscription changes occur. However, assignments are randomly selected among devices that satisfy QoS requirements, without considering energy optimization.
- **Dynamic – QoS & Energy Aware (Smart-MQTT):** Our proposed approach dynamically reassigns tasks upon subscription changes and optimizes assignments using a multi-objective formulation that considers both QoS satisfaction and device energy levels.

6.3 Experimental Setup

6.3.1 Small-Scale IoT Testbed

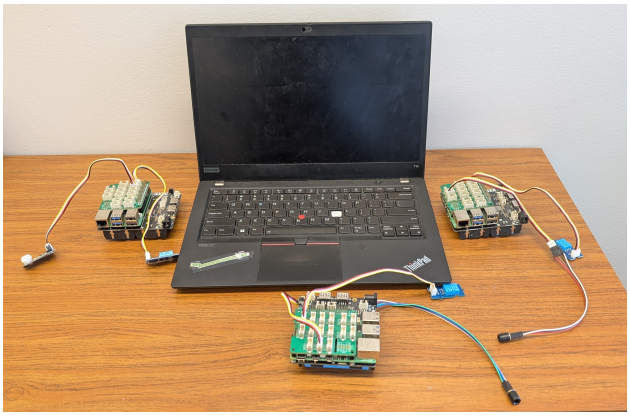


Figure 6.1: Smart-MQTT testbed deployment.

Testbed Deployment:

For the real-world deployment (Fig. 6.1), we utilized an Asustek G751JL computer running Windows Subsystem for Linux (WSL) as the MQTT broker and Smart-MQTT coordinator. The testbed included six Raspberry Pi devices—three equipped with battery monitoring capabilities and three without.

Devices were connected to three types of sensors:

- Temperature and humidity sensor
- Motion sensor
- Thermal sensor

Three publishers were configured, each equipped with two of the three sensors. The system supported three topics: *RM1-Motion*, *RM1-Temperature-and-Humidity*, and *RM1-Thermal*. Six subscribers generated QoS-aware subscription requests with varying requirements.

QoS capabilities and requirements were randomly generated within the ranges shown in Table 6.1. If no feasible assignment existed, values were regenerated. Each experiment was repeated 100 times. To simulate dynamic conditions, QoS requirements were regenerated hourly. The weighting parameter was set to $\alpha = 0.5$ unless otherwise specified.

Parameter	**Range**
# Publishers	3
# Subscribers	6
# Topics	3
Frequency QoS (Hz)	$[0.1, 6]$
Latency QoS (ms)	$[200, 560]$
Accuracy QoS (%)	$[84, 98]$

Table 6.1: Default parameter settings for real-world testbed.

6.3.2 Large-Scale Simulation

To evaluate scalability, we introduced a Scale Indicator (SI), representing the number of devices, subscribers, and sensing tasks. For example, SI=3 corresponds to three devices, three subscribers, and three tasks.

The SI was varied from 3 to 15 in increments of 3. Device capabilities, initial energy levels, and subscriber QoS requirements were randomly generated. The weighting parameter α was varied from 0.0 to 1.0 in increments of 0.2 to analyze trade-offs between energy efficiency and QoS satisfaction.

6.4 Evaluation Metrics

6.4.1 System Lifetime

System lifetime is defined as the elapsed time from initial task assignment until no eligible publisher remains capable of performing at least one active sensing task. A publisher is

considered eligible for a task if it has sufficient remaining energy and the sensing capability required to satisfy the task's QoS constraints. The system is operational as long as every active topic has at least one eligible publisher; it terminates when any topic loses all capable publishers due to energy depletion.

6.4.2 QoS Satisfaction Ratio

QoS satisfaction is computed as the average task utility across all active tasks, where task utility $u[p][t] \in [0, 1]$ is defined in Chapter 4 and reflects how closely a device's sensing capabilities match the QoS requirements of an assigned task.

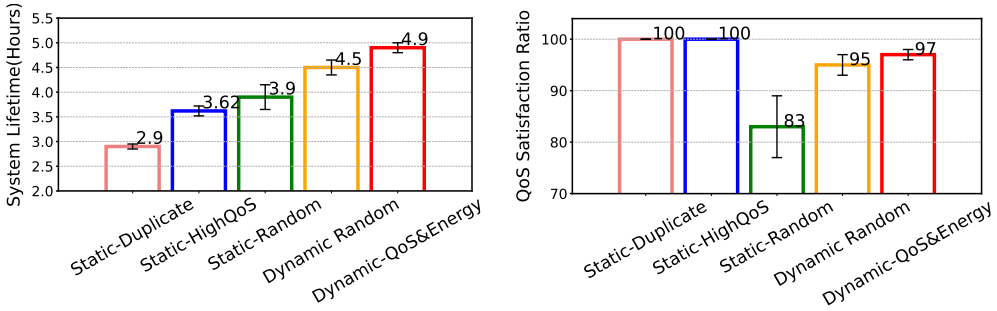
6.4.3 Execution Latency

Execution latency is defined as the time between the final subscriber's subscription request and the receipt of the first sensing message.

6.5 Results and Analysis

The following subsections report results for each of the four research questions defined in Section 6.1. Testbed results address research questions 1 through 3, while the large-scale simulation results address research question 4.

6.5.1 Testbed Results



(a) System Lifetime (b) QoS Satisfaction

Figure 6.2: Performance comparison on physical testbed.

Figure 6.2 presents the system lifetime and QoS satisfaction ratio for each approach across 100 experimental trials.

The Static-Duplicate baseline achieved the shortest system lifetime, as all devices simultaneously sense and transmit data for every topic regardless of subscriber demand, leading to rapid and uniform energy depletion. Despite this, it achieves 100% QoS satisfaction, since the highest-capability device is always actively publishing.

Static-Random and Static-HighQoS improved lifetime by approximately 35% and 25%, respectively. Notably, Static-HighQoS performs *worse* than Static-Random, despite always selecting the device with the best QoS capability. This counterintuitive result arises because Static-HighQoS consistently assigns tasks to the same high-capability devices, concentrating energy consumption on a small subset of nodes and triggering early depletion. Static-Random, by contrast, distributes assignments more evenly across devices by chance, inadvertently extending system lifetime. This finding directly illustrates the cost of ignoring energy balance in task allocation and reinforces the motivation for the energy-aware design of Smart-MQTT.

Dynamic-Random increased lifetime by nearly 55% while maintaining 95% QoS satisfaction, demonstrating that even reactive reassignment without energy awareness provides substantial benefit over static approaches.

The proposed Dynamic-QoS&Energy approach achieved the best overall performance, extending lifetime by 69% compared to the Static-Duplicate baseline while maintaining 97% QoS satisfaction.

Algorithm	**Latency (ms)**
Static-Duplicate	1.0
Static-Random	5.08
Static-HighQoS	5.87
Dynamic-Random	6.4
Dynamic-QoS&Energy	30

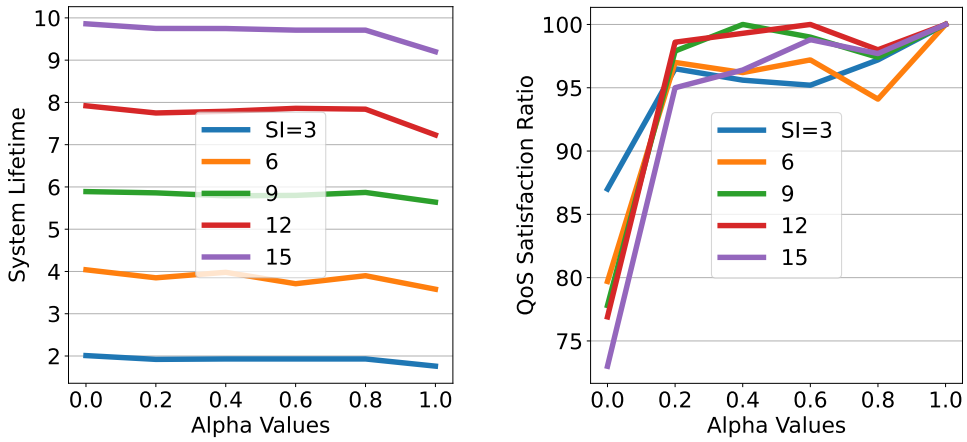
Table 6.2: Execution latency per algorithm.

Although the MILP-based optimization incurs higher latency (approximately 30 ms), this overhead remains acceptable given the significant gains in lifetime and QoS balance.

Energy overhead due to periodic status updates was also evaluated. The system operated for 5.0 hours without updates and 4.68 hours with updates, corresponding to an 8% energy overhead.

6.5.2 Large-Scale Simulation Results

Figure 6.3 presents system lifetime and QoS satisfaction as functions of system scale (SI) and the weighting parameter α. System lifetime generally increases with system size, since larger



(a) System Lifetime (b) QoS Satisfaction

Figure 6.3: Performance under varying system scale and α.

deployments provide greater flexibility for distributing sensing workloads across devices. However, increasing α reduces lifetime as the optimization prioritizes QoS satisfaction over energy preservation.

QoS satisfaction rises sharply once $\alpha > 0.2$ and remains high thereafter. At $\alpha = 0$, the algorithm ignores QoS entirely, and larger systems experience lower satisfaction due to increased task contention without quality-aware selection. At $\alpha = 0.2$, the algorithm strikes an effective balance between QoS and energy, producing stable performance across all tested scale levels. This result directly answers research question 4: Smart-MQTT scales favorably, with performance benefits amplifying rather than diminishing as deployment size grows.

6.6 Discussion

6.6.1 Interpretation of Results

The experimental results collectively validate the central claim of this thesis: that coordinating publisher-side sensing behavior based on application-level QoS requirements and device energy status substantially improves MQTT-based IoT system performance. The 69% lifetime improvement over the uncoordinated baseline is not merely a product of reducing redundant transmissions; it reflects a more fundamental redistribution of sensing workloads across devices, which slows the energy depletion of individual publishers(data producers) and extends the period during which every active topic retains at least one eligible publisher capable of performing it's assigned sensing task.

The 97% QoS satisfaction rate demonstrates that this energy saving is not achieved at the expense of application quality. The small gap from 100% satisfaction reflects the inherent cost of energy-aware task allocation: when the most capable device is conserved to preserve battery life, the second-best device may be assigned in its place, incurring a slight reduction in task utility. The weighting parameter α makes this trade-off explicit and configurable. The simulation results indicate that $\alpha = 0.2$ represents a practical operating point, achieving near-maximum QoS satisfaction while preserving meaningful energy savings across all tested system scales.

6.6.2 Overhead Analysis

The 30 ms execution latency of the MILP-based optimization is higher than all baselines but remains well within acceptable bounds for IoT sensing applications, where sensing intervals are typically measured in seconds or minutes rather than milliseconds. The 8% energy overhead introduced by periodic status reporting represents a modest cost relative to the lifetime improvements achieved; however, this figure is sensitive to the reporting interval and the number of devices, and may increase in larger deployments. Tuning the reporting interval to match the expected rate of environmental change is therefore an important deployment consideration.

6.6.3 Threats to Validity

Several factors limit the generalizability of these results. The testbed evaluation involved only three publishers, three topics, and six subscribers—a small-scale configuration that may not reflect the complexity of industrial deployments. The simulation study extends the evaluation to larger scales but relies on randomly generated capability and energy parameters rather than empirical IoT traces. Additionally, the evaluation assumes reliable network connectivity; in environments with high packet loss or intermittent connectivity, the performance of the status reporting mechanism and the accuracy of the broker's device model may degrade.

These limitations notwithstanding, the results demonstrate consistent and substantial improvements across both the physical testbed and simulation environments, providing confidence in the directional validity of the findings. Chapter 7 summarizes these contributions and discusses how future work can address the identified limitations.

CHAPTER 7

Conclusion and Future Work

This chapter synthesizes the contributions and findings of this thesis, reflects on its limitations, and identifies promising directions for future research.

7.1 Summary of the Research

This thesis presented Smart-MQTT, a middleware framework designed to improve energy efficiency in MQTT-based IoT systems through dynamic, QoS-aware sensing coordination at the publisher side. The work was motivated by a structural inefficiency in existing MQTT deployments: devices continuously sense and publish data at fixed rate, regardless of actual application demand, leading to redundant sensing operations, uneven energy depletion, and unnecessarily shortened device lifetimes.

Smart-MQTT addresses this problem by introducing a lightweight middleware layer that bridges the gap between application-level QoS requirements and device-level sensing behavior. Subscribers encode sensing requirements in standard MQTT subscription requests; IoT devices periodically report their capabilities and energy status to the broker; and the broker periodically solves a multi-objective optimization problem to assign sensing tasks to the most appropriate available device. The result is a system that reduces redundant sensing, distributes energy consumption more evenly across devices, and maintains high application-level service quality—all without modifying the underlying MQTT protocol or requiring changes to existing client applications.

7.2 Summary of Contributions (*in the first chapter, you summarized the contributions already, this looks duplicated*)

The five research objectives stated in Chapter 1 have been addressed as follows.

- **QoS-aware middleware architecture.** In response to the objective of designing a QoS-aware middleware that enables applications to express sensing requirements without modifying the core MQTT protocol, this work proposed Smart-MQTT —a lightweight framework that augments existing MQTT infrastructures via broker-side and publisher-side add-ons. The system preserves full backward compatibility with standard MQTT clients and brokers.

- **Publisher-side sensing coordination.** In response to the objective of developing an energy-efficient task allocation mechanism, this research introduced a dynamic task assignment strategy that controls which IoT devices perform which sensing tasks and at what quality levels, targeting energy optimization directly at the sensing source rather than the communication layer.

- **Optimization-based task allocation model.** In response to the objective of formulating the allocation problem as a principled optimization model, the sensing task allocation problem was cast as a multi-objective optimization that jointly balances system lifetime and QoS satisfaction, with a configurable weighting parameter enabling flexible trade-off control.

- **Prototype implementation.** In response to the objective of implementing the solution in a real MQTT-based system, a complete prototype was developed using a modified MQTT broker and lightweight publisher-side add-ons, and deployed on a physical testbed of Raspberry Pi devices.

- **Experimental evaluation.** In response to the objective of evaluating effectiveness and overhead, the system was assessed through 100-trial physical testbed experiments and large-scale simulations, demonstrating a 69% improvement in system lifetime over the uncoordinated baseline and 97% QoS satisfaction at the default operating configuration.

7.3 Key Experimental Insights

Beyond confirming that Smart-MQTT outperforms all baselines, the experimental evaluation produced several non-obvious findings with practical implications. First, the Static-HighQoS baseline—which always assigns tasks to the most capable device—achieved a *shorter* system lifetime than Static-Random assignment. This result reveals that capability-driven allocation without energy awareness concentrates workloads on a small subset of devices, accelerating

their depletion. This finding reinforces the importance of joint energy and QoS optimization rather than treating them as independent objectives.

Second, the simulation results show that the system lifetime benefit of Smart-MQTT grows with scale: larger deployments offer greater flexibility in distributing tasks across devices, amplifying the benefit of energy-aware allocation. This suggests that the approach is particularly well suited to the large IoT deployments where energy sustainability is most critical.

Third, the α sensitivity analysis demonstrates that $\alpha = 0.2$ provides an effective default operating point, achieving near-maximum QoS satisfaction while retaining substantial lifetime improvements. Values of α above 0.5 provide diminishing returns in QoS satisfaction while progressively sacrificing lifetime, making them appropriate only for applications with strict service requirements.

7.4 Limitations

While the proposed framework demonstrates promising results, several limitations remain. The task allocation algorithm relies on a centralized optimization process executed at the MQTT broker. Although this approach enables global coordination across devices, it may become computationally expensive in extremely large-scale IoT deployments involving thousands of devices and sensing tasks.

Furthermore, the current implementation assumes reliable periodic status updates from IoT devices. In highly dynamic or unreliable network environments, delayed or missing status reports may affect the accuracy of task allocation decisions. Additional mechanisms for handling incomplete or outdated device information could further improve system robustness.

Finally, the QoS model considered in this work focuses primarily on sensing frequency and reliability. Other QoS dimensions, such as data freshness, and network congestion awareness, were not explicitly incorporated into the optimization model.

7.5 Future Research Directions

Several promising directions for future research emerge from this work.

One potential extension involves developing heuristic or distributed task allocation algorithms to improve scalability in large-scale deployments. Such approaches could approximate the MILP-based solution while significantly reducing computational overhead.

Another direction involves expanding the QoS model to incorporate additional application requirements, such as data freshness guarantees, or context-aware sensing policies. Inte-

grating network-level information into the task allocation process may also further improve system efficiency.

Future work may also explore integrating Smart-MQTT with edge computing platforms. By distributing optimization logic across multiple edge nodes rather than a centralized broker, the system could achieve improved scalability and faster response times.

Finally, evaluating the framework in real-world industrial or smart-city deployments would provide valuable insights into its long-term operational performance and practical deployment challenges.

This thesis has demonstrated that the energy sustainability of MQTT-based IoT systems can be substantially improved by closing the feedback loop between application-level QoS requirements and publisher-side sensing behavior. The prevailing assumption that sensing is a fixed-cost background process—independent of what subscribers actually need—is shown to be a source of significant and avoidable energy waste. By making sensing behavior a first-class optimization target, Smart-MQTT achieves lifetime improvements that communication-layer optimizations alone cannot provide.

The system contributes to a broader shift in IoT system design: from treating the communication infrastructure as the sole lever for efficiency toward recognizing the sensing layer as an equally important site of optimization. As IoT deployments scale to encompass billions of devices across smart buildings, environmental monitoring networks, and industrial automation systems, the ability to coordinate sensing workloads intelligently—rather than simply transmit them efficiently—will be essential to achieving the energy sustainability that long-lived, battery-powered IoT systems demand.

BIBLIOGRAPHY

[1] Saima Abdullah and Kun Yang. An energy-efficient message scheduling algorithm in internet of things environment. In *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 311–316. IEEE, 2013.

[2] Shadi Al-Sarawi, Mohammed Anbar, Rosni Abdullah, and Ahmad B Al Hawari. Internet of things market analysis forecasts, 2020–2030. In *2020 Fourth World Conference on smart trends in systems, security and sustainability (WorldS4)*, pages 449–453. IEEE, 2020.

[3] Melvin Bender, Erkin Kirdan, Marc-Oliver Pahl, and Georg Carle. Open-source mqtt evaluation. In *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–4. IEEE, 2021.

[4] Pegah Nikbakht Bideh, Jonathan Sönnerup, and Martin Hell. Energy consumption for securing lightweight iot protocols. In *Proceedings of the 10th International Conference on the Internet of Things*, pages 1–8, 2020.

[5] Nico Bokhari, Zhengquan Li, and Zheng Song. Mqtt-ees: Optimizing energy efficiency by aggregating sensing tasks on iot devices. In *IEEE VTC 2024-Fall*, pages 1–6. IEEE, 2024.

[6] Matteo Collina and Contributors. Mqtt.js. https://github.com/mqttjs/MQTT.js, 2024. Accessed: 2025-05-05.

[7] Francisco L de Caldas Filho, Rodrigo L Rocha, Cláudia JB Abbas, Lucas MC E Martins, Edna Dias Canedo, and Rafael T de Sousa. Qos scheduling algorithm for a fog iot gateway. In *2019 Workshop on Communication Networks and Power Systems (WCNPS)*, pages 1–6. IEEE, 2019.

[8] Eclipse Foundation. Eclipse Paho MQTT Client Libraries. https://www.eclipse.org/paho/, 2014. Accessed: 2025-05-05.

[9] Norisvaldo Ferraz Junior, Anderson AA Silva, Adilson E Guelfi, and Sergio T Kofuji. Performance evaluation of publish-subscribe systems in iot using energy-efficient and context-aware secure messages. *Journal of Cloud Computing*, 11(1):6, 2022.

[10] HiveMQ GmbH. Hivemq: Enterprise mqtt broker for fast, secure and reliable data movement to and from connected devices. https://www.hivemq.com, 2024. Accessed: 2025-05-05.

[11] HiveMQ GmbH. Hivemq mqtt client. https://github.com/hivemq/hivemq-mqtt-client, 2024. Accessed: 2025-05-05.

[12] Saidou Haman, Djorwe Temoa, Eric Michel Deussom Djomadji, et al. Proposal for energy consumption reduction between connected objects in a network running on mqtt protocol. *Journal of Computer and Communications*, 12(10):177–188, 2024.

[13] K Hwang, J Moon Lee, and I Hwan Jung. Extension of reception frequency control to mqtt protocol. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, pages 2278–3075, 2019.

[14] IBM and Eurotech. MQTT for Sensor Networks (MQTT-SN): Protocol Specification Version 1.2. Technical report, MQTT.org, October 2013. Version 1.2, available via OASIS.

[15] Yerin Im and Mingyu Lim. E-mqtt: End-to-end synchronous and asynchronous communication mechanisms in mqtt protocol. *Applied Sciences*, 13(22):12419, 2023.

[16] Heriberto J Jara Ochoa, Raul Peña, Yoel Ledo Mezquita, Enrique Gonzalez, and Sergio Camacho-Leon. Comparative analysis of power consumption between mqtt and http protocols in an iot platform designed and implemented for remote real-time monitoring of long-term cold chain transport operations. *Sensors*, 23(10):4896, 2023.

[17] Sujita Jiwangkura, Peraphon Sophatsathit, and Achara Chandrachai. Mqtt of iot classification in energy saving. In *proceeding of 11th International Conference on Data Mining, Computers, Communication and Industrial Applications (DMCCIA-2017), Kuala Lumpur*, 2017.

[18] Antonio La Marra, Fabio Martinelli, Paolo Mori, Athanasios Rizos, and Andrea Saracino. Introducing usage control in mqtt. In *Computer Security: ESORICS 2017 International Workshops, CyberICPS 2017 and SECPRE 2017, Oslo, Norway, September 14-15, 2017, Revised Selected Papers 3*, pages 35–43. Springer, 2018.

[19] Roger Light and Eclipse Foundation. Eclipse mosquitto: An open source mqtt broker. https://mosquitto.org, 2024. Accessed: 2025-05-05.

[20] Biswajeeban Mishra and Attila Kertesz. The use of mqtt in m2m and iot systems: A survey. *Ieee Access*, 8:201071–201086, 2020.

[21] Dae-Hyeok Mun, Minh Le Dinh, and Young-Woo Kwon. An assessment of internet of things protocols for resource-constrained applications. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 555–560. IEEE, 2016.

[22] OASIS Standard. MQTT Version 5.0. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html, March 2019. OASIS Standard. Accessed: 2025-02-24.

[23] Fabio Palmese, Alessandro EC Redondi, and Matteo Cesana. Adaptive quality of service control for mqtt-sn. *Sensors*, 22(22):8852, 2022.

[24] Jun-Hong Park, Hyeong-Su Kim, and Won-Tae Kim. Dm-mqtt: An efficient mqtt based on sdn multicast for massive iot communications. *Sensors*, 18(9), 2018.

[25] Linh Manh Pham, Nguyen-Tuan-Thanh Le, and Xuan-Truong Nguyen. Multi-level just-enough elasticity for mqtt brokers of internet of things applications. *Cluster Computing*, 25(6):3961–3976, 2022.

[26] Ivana Podnar Zarko, Aleksandar Antonic, and Krešimir Pripužic. Publish/subscribe middleware for energy-efficient mobile crowdsensing. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, pages 1099–1110, 2013.

[27] Thomas Rausch, Stefan Nastic, and Schahram Dustdar. Emma: Distributed qos-aware mqtt middleware for edge computing applications. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 191–197. IEEE, 2018.

[28] Bertram Schütz, Jan Bauer, and Nils Aschenbruck. Improving energy efficiency of mqtt-sn in lossy environments using seed-based network coding. In *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*, pages 286–293. IEEE, 2017.

[29] Ventseslav Kirilov Shopov and Vanya Dimitrova Markova. Application of hungarian algorithm for assignment problem. In *2021 International Conference on Information Technologies (InfoTech)*, pages 1–4. IEEE, 2021.

[30] Namrata Singh and Ayan Kumar Das. Energy-efficient fuzzy data offloading for iomt. *Computer Networks*, 213:109127, 2022.

[31] Jevgenijus Toldinas, Borisas Lozinskis, Edgaras Baranauskas, and Algirdas Dobrovolskis. Mqtt quality of service versus energy consumption. In *2019 23rd international conference electronics*, pages 1–4. IEEE, 2019.

[32] Riccardo Venanzi, Burak Kantarci, Luca Foschini, and Paolo Bellavista. Mqtt-driven sustainable node discovery for internet of things-fog environments. In *2018 IEEE ICC*, pages 1–6, 2018.

[33] Abhishek Viswanathan. *Analysis of power consumption of the MQTT protocol*. PhD thesis, University of Pittsburgh, 2017.