UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DAVID JÚNIO MOTA CAVALCANTI

AN ENABLING FRAMEWORK FOR CUSTOMIZATION AND ADAPTATION OF MIDDLEWARE OF THINGS

Recife

2025

DAVID JÚNIO MOTA CAVALCANTI

AN ENABLING FRAMEWORK FOR CUSTOMIZATION AND ADAPTATION OF MIDDLEWARE OF THINGS

A PhD thesis presented by **David Junio Mota Cavalcanti** in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Post-graduate Program in Computer Science at the Universidade Federal do Pernambuco.

Research Areas: Distributed Systems and Internet of Things

Supervisor: Prof. Dr. Nelson Souto Rosa

Co-supervisor: Prof. Dr. Danny Hughes

Recife

2025

.Catalogação de Publicação na Fonte. UFPE - Biblioteca Central

Cavalcanti, David Junio Mota.
An enabling framework for customization and adaptation of middleware of things / David Junio Mota Cavalcanti. - Recife, 2025.
148f.: il.

Tese (Doutorado) - Universidade Federal de Pernambuco, Centro de Informática, Programa de Pós-graduação em Ciência da Computação, 2024.
Orientação: Nelson Souto Rosa.
Coorientação: Daniel Hughes.
Inclui referências e apêndices.

1. Internet of Things; 2. Adaptive Middleware; 3. Software Architecture; 4. Uncertainties; 5. Energy Saving; 6. Smart Water Management. I. Rosa, Nelson Souto. II. Hughes, Daniel. III. Título.

UFPE-Biblioteca Central

David Junio Mota Cavalcanti

"AN ENABLING FRAMEWORK FOR CUSTOMIZATION AND ADAPTATION OF MIDDLEWARE OF THINGS"

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação. Área de Concentração: Redes de Computadorea e Sistemas Distribuídos.

Aprovada em: 16/12/2024.

Orientador: Prof. Dr. Nelson Souto Rosa

BANCA EXAMINADORA

I dedicate this thesis to my late brother, Saudade.

AGRADECIMENTOS

À minha mãe, Suzana Mota, pelo seu imenso amor, cuidado e dedicação ao longo de toda a minha vida. Agradeço também por ser exemplo de vida, pelos sacrifícios na minha criação e por me ensinar que a educação é a ferramenta mais essencial que um ser humano pode ter.

À minha irmã, Dayse Mota, pela amizade, amor e respeito. Sou profundamente grato por todas as experiências que compartilhamos. E, especialmente, ao meu irmão, Dayvson Mota (*in memoriam*), que me ensinou que o que realmente importa na vida é o que fazemos pelo outro enquanto estamos vivos, o nosso impacto e as memórias que deixamos nas pessoas que amamos.

Ao meu orientador e amigo, Prof. Dr. Nelson Rosa, por todos os ensinamentos e pela oportunidade de realizar esta pesquisa. Agradeço por acreditar em mim e por me apoiar incansavelmente. Sou e serei eternamente grato pela paciência, amizade e dedicação exemplar ao longo dessa jornada, e pelo grande incentivo que tornou possível a realização deste trabalho.

Ao meu coorientador, Prof. Dr. Danny Hughes, pelas nossas reuniões e pelo impacto positivo na jornada desta pesquisa. Quero também agradecer a todos os colegas que conheci no *Networked Embedded Software Taskforce Laboratory* da *Katholieke Universiteit Leuven*, na Bélgica.

Aos meus amigos de pesquisa no laboratório, Ranieri Valença, Angelo Fernandes, Yeda Lima e Esdras Bispo, pelas nossas conversas e discussões esclarecedoras, que foram extremamente úteis ao longo do desenvolvimento deste trabalho. Estiveram sempre prontos para ajudar sempre que necessário.

Aos meus queridos amigos da vida, Eduardo Virães, Agnes Regina e Anderson Queiroz, que, cada um à sua maneira, contribuiu para tornar este trabalho possível.

Ao Instituto Federal de Pernambuco (Campus Igarassu) e aos meus colegas de trabalho, Ramon Mota, Lincoln Tavares e demais colegas, pelo incentivo e apoio que tornaram esta pesquisa possível, incluindo a valiosa experiência de estudar no exterior. Sem a ajuda de vocês, isso não teria sido possível.

À Universidade Federal de Pernambuco e ao Centro de Informática por me acolherem e me fornecerem apoio de várias formas ao longo do curso desta pesquisa.

Finalmente, a todos os amigos, colegas e irmãos que contribuíram, direta ou indiretamente, para a minha jornada, o meu sincero agradecimento.

ACKNOWLEDGEMENTS

To my mother, Suzana Mota, for her immense love, care and dedication throughout my life. I also thank her for being a life example, for her sacrifices in raising me, and for teaching me that education is the most essential tool a human can have.

To my sister, Dayse Mota, for her friendship, love, and respect. I am deeply grateful for all the experiences we've shared. And especially to my brother, Dayvson Mota (in memoriam), who taught me that what truly matters in life is what we do for others while we are alive, our impact and the memories we leave on those we love.

To my supervisor and friend, Prof. Dr. Nelson Rosa, for all the teachings and the opportunity to conduct this research. I thank him for believing in me and for supporting me tirelessly. I am and will be eternally grateful for his patience, friendship, and exemplary dedication throughout this journey, and the great encouragement that made completing this work possible.

To my co-supervisor, Prof. Dr. Danny Hughes, for our meetings and his positive impact on the journey of this research. I also want to thank all the colleagues I met at the Networked Embedded Software Taskforce Laboratory at the Katholieke Universiteit Leuven in Belgium.

To my research friends in the lab, Ranieri Valença, Angelo Fernandes, Yêda Lima and Esdras Bispo, for our insightful conversations and discussions, which were incredibly helpful throughout the development of this work. They were always ready to assist whenever needed.

To my dear friends in life, Eduardo Virães, Agnes Regina and Anderson Queiroz, who, each in their own way, contributed to making this work possible.

To the Instituto Federal de Pernambuco (Campus Igarassu) and my colleagues at work, Ramon Mota, Lincoln Tavares and others, for their encouragement and support, which made this research possible, including the valuable experience of studying abroad. Without your help, this would not have been possible.

To the Universidade Federal de Pernambuco and the Centro de Informática for welcoming me and providing support in various ways throughout the course of this research.

Finally, to all the friends, colleagues, and brothers who contributed, directly or indirectly, to my journey, my heartfelt thanks.

ABSTRACT

The Internet of Things (IoT) enables the development of applications using smart devices called things. The increasing processing, storage and communication capacities of devices boosted the growth of distributed IoT applications. IoT Middleware systems have become essential for developing these applications by facing distribution, device heterogeneity and application interoperability. However, IoT environments are highly dynamic and susceptible to changes, introducing uncertainties, such as changing user requirements (e.g., evolving applications), changing environmental conditions (e.g., network delays) and varying resource availability, e.g., battery levels. These uncertainties can lead to failures or compromise application functioning. Self-adaptive middleware systems have been responsible for dealing with uncertainties by dynamically adapting their behavior/structure and applications built atop them without system shutdowns. Managing uncertainties at various layers, each requiring a distinct adaptive action, making it challenging to manage them simultaneously. This thesis introduces *M*iddleware *Ex*tendify (MEx), a solution for building and executing IoT self-adaptive middleware systems. MEx simplifies the implementation of middleware and provides an execution environment for supporting a range of adaptation mechanisms, ensuring that the middleware meets the evolving demands of applications and copes with changes at runtime. Additionally, this thesis presents AquaMOM, an adaptive IoT system designed for monitoring water consumption in semi-arid regions, where frequent changes in water availability and usage patterns justify the need for an adaptive approach. It also includes a low-cost IoT device prototype equipped with water and energy monitoring sensors. Built using MEx, AquaMOM leverages MEx's capabilities to manage uncertainties, respond to dynamic changes, and meet application demands. The evaluation of MEx encompasses different adaptive middleware implementations to measure its adaptation mechanisms' impact while comparing its performance with a widely adopted MQTT-based middleware. Results indicate that adaptation comes with acceptable performance costs while providing significant benefits, such as fine-tuning middleware functionalities or enhancing application alignment, e.g., adaptation increases publishing time from 4.24 ms to 6.27 ms, while extending battery lifetime from 1.4 to 6.6 days. These findings show MEx's potential to enhance IoT middleware, making systems more adaptable and efficient in real scenarios.

Keywords: Internet of Things. Adaptive Middleware. Software Architecture. Uncertainties. Energy Saving. Smart Water Management. Challenging Environments.

LIST OF FIGURES

LIST OF SOURCE CODES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

***p*ADL**	Python-based Architecture Description Language
ADL	Architecture Description Language
API	Application Programming Interface
BLE	Bluetooth Low Energy
BNF	Backus-Naur Form
CoAP	Constrained Application Protocol
DCAM	Duty Cycle Adaptive Mechanism
DSL	Domain-Specific Language
ENO	Energy-Neutral Operation
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IIoT	Industrial IoT
IoT	Internet of Things
LoRa	Long Range
MAPE-K	Monitor-Analyzer-Planner-Executor + Knowledge
MEx	Middleware Extendify
MOM	Message-Oriented Middleware
MQTT	Message Queuing Telemetry Transport
OODA	Observe, Orient, Decide, Act
OS	Operating System
OSGi	Open Services Gateway Initiative
PC	Personal Computer
QoE	Quality of Experience
QoS	Quality of Service
RFID	Radio Frequency Identification

SAS	Self-Adaptive System
SMS	Short Message Service
TDCAM	Time-Based Duty Cycle Adaptive Mechanism
UML	Unified Modeling Language
UN	United Nations

CONTENTS

1 INTRODUCTION

"The utopia is on the horizon. I move two steps closer; it moves two steps further away. I walk another ten steps, and the horizon runs ten steps further away. As much as I may walk, I'll never reach it. So, what's the point of utopia? The point is this: to keep walking."

—Eduardo Galeano

This chapter introduces the research conducted in this thesis. First, it presents the context and motivation for the research. Then, it identifies the research problem and a summary of existing solutions and how they fail to solve it. Next, it outlines the proposed solution. Finally, the chapter summarizes the thesis's contributions and provides an overview of the document's organization.

1.1 CONTEXT AND MOTIVATION

The IoT is a technology that enables the development of systems using smart objects, known as "things". These include sensors, actuators, smartphones, vehicles, household appliances, or even virtual devices, such as software systems that simulate real environments (Atzori et al., 2019; Gubbi et al., 2013). Powered by communication, computing, and storage technologies, these devices collectively work to provide applications and services. Essentially, IoT devices collect and process data from their environment and exchange information with each other over the Internet to achieve common goals, making environments smarter and benefiting people (Alfonso et al., 2021; Qadri et al., 2020; Razzaque et al., 2016; Atzori; Iera; Morabito, 2010).

As IoT systems continue to increase in complexity and connectivity, the number of connected devices reached 20 billion in 2020 (Xu et al., 2020) and is expected to reach 500 billion by 2030 (Mahamuni, 2023; Mathur et al., 2023; Karie; Sahri; Haskell-Dowland, 2020). This growth, along with advances in device capabilities, such as processing, storage, and communication, has driven the development of distributed IoT applications in different domains, e.g., smart homes (Albany et al., 2022), smart cities (Costa et al., 2022), smart water management (Singh;

Ahmed, 2021), and Industrial IoT (IIoT) (Peter; Pradhan; Mbohwa, 2023).

In this context, middleware systems play an essential role in facilitating the development and deployment of IoT applications (Razzaque et al., 2016). Middleware bridges between IoT applications, services, and devices by enabling communication, dealing with heterogeneity, and allowing interoperability (Blair; Schmidt; Taconet, 2016). This support allows developers to focus primarily on the core system requirements, i.e., the business logic (Borges et al., 2023; Bandyopadhyay et al., 2011; Schmidt; Buschmann, 2003). IoT middleware addresses the heterogeneity of systems and services at the application level by supporting their interoperability while offering essential services such as data and resource management, event and context detection, and security features (Sethi; Sarangi, 2017). At the device level, the middleware abstracts device details from applications and facilitates seamless communication between devices.

As IoT applications continue to evolve in complexity and connectivity, recently, there has been growing interest in developing IoT applications for challenging environments, such as extreme locations (e.g., nuclear plant management and emergency responses) or urban settings, including remote and economically needy regions, e.g., semi-arid areas (Cavalcanti et al., 2024). In such settings, IoT applications facilitate monitoring critical parameters, such as extreme temperatures, radiation, and water levels. They can help relieve humans from costly and dangerous tasks, such as inspecting hazardous environments and disaster sites and monitoring water consumption in remote areas (Kant; Jolfaei; Moessner, 2024).

However, these systems often face unexpected harsh conditions due to aging infrastructure, demand exceeding capacity, and increasingly intense operating conditions brought on by climate change or the nature of the application (Kant; Jolfaei; Moessner, 2024). Addressing the advance of IoT applications brings new challenges, including diversity of communication protocols, technological diversity, and resource management considerations, especially concerning power consumption and strict demands for Quality of Service (QoS) (e.g., performance, scalability, and security) based on application requirements (Alfonso et al., 2021). Despite middleware support, fulfilling these commitments remains challenging due to the distributed nature of IoT systems. These systems are dynamic and subject to continuous changes, unexpected events, and harsh environmental conditions, introducing uncertainties at different layers of the IoT (Kant; Jolfaei; Moessner, 2024; Cavalcanti; Hughes; Rosa, 2023; Alfonso et al., 2021; Muccini et al., 2018). Uncertainties may appear due to changes in user needs, workloads, software aging (application), loss of connectivity or delays (communication). Additionally, devices may have fluctuations in resource availability, affecting sensors' reliability or degrading components (Alfonso et al., 2021; Moreno; Cámara Javier andGarlan; Schmerl, 2015). These uncertainties can impact the QoS of the IoT applications, affecting integrity and trustworthiness or significantly reducing devices operating lifetime (Kant; Jolfaei; Moessner, 2024; Cavalcanti; Hughes; Rosa, 2023; Weyns; Ramachandran; Singh, 2018).

Consider, for example, a simple, smart water application designed to monitor poorly structured water cisterns in rural areas. The application can use sensors to monitor water level and the device's battery, transmitting these data to a messaging service, which forwards them to the dashboard web application to display digital metering feedback. Reliable communication with low latency and efficient transmission power are crucial for sending packets, as they ensure accurate delivery and minimize delays between sending and receiving data, which is essential for dynamic applications. However, uncertainties like network interference (e.g., weather conditions or weak/nearby Wi-Fi signals), traffic load fluctuations (e.g., system monitoring rate congestion) and resource limitations (e.g., battery power) can affect the application's operation. These uncertainties can lead to data collection errors or system failures, compromising integrity and trust—critical factors in many IoT systems (Kant; Jolfaei; Moessner, 2024; Weyns; Ramachandran; Singh, 2018).

Given these complexities, it is essential for IoT middleware to also handle harsh conditions and uncertainties at runtime. Adopting self-adaptive principles is a crucial strategy to address such uncertainties by modifying IoT applications' behavior or structure in response to system or environmental changes (Weyns, 2020; Lemos et al., 2013). Recently, developers have incorporated adaptive capabilities into middleware systems to face uncertainties (Cavalcanti; Carvalho; Rosa, 2021; Soojin; Sungyong, 2019; Rausch; Nastic; Dustdar, 2018). The middleware becomes self-adaptive by adapting its behavior or structure to fix bugs and accommodate user and environmental changes.

1.2 THE PROBLEM

With the rapid evolution of technology, IoT significantly influences how people interact with devices in daily life. IoT allows the interconnection of devices from the digital, virtual, and physical world to the Internet with minimal human involvement. This connectivity allows users to control devices and access sensor data remotely. The data generated by these devices can unveil users' habits and trends, which can be utilized to enhance various services.

The inherent dynamic nature and need for continuous operation introduce a variety of uncertainties during the operation of the IoT applications. At the same time, IoT applications may face uncertainties caused by variations in user and application goals, changes in environmental conditions (e.g., network delays), and dynamics in resource availability. These uncertainties can lead to issues in data collection, resulting in failures in IoT application execution and ultimately putting these applications in insecure situations, such as loss of connection and incorrect results (Alagar; Wan, 2019; Magruk, 2015). Therefore, facing these uncertainties and assessing, adjusting, or enhancing the system before, during, and after uncertainties appear is essential to minimize service degradation, maximize coverage, and improve offered services (Kant; Jolfaei; Moessner, 2024; Sunny et al., 2021). IoT applications are expected to operate under conditions of uncertainty and without interruption.

The IoT middleware self-adaptation capabilities form the foundation for ensuring IoT applications function properly (Moghaddam; Rutten; Giraud, 2020; Nundloll et al., 2019). As challenges in developing adaptive middleware for IoT persist and continue to be an active research area, this thesis focuses on a novel middleware solution capable of exploring and tackling these challenges and provides new contributions to the field. This solution aims to develop and execute self-adaptive middleware systems for IoT.

With this context in mind, the research problem identified for this thesis is:

- **How can middleware systems for IoT be designed and implemented with adaptive capabilities as a first-class concept while addressing their fundamental requirements?**

These systems must meet common middleware requirements such as distribution support, interoperability, resources and data management, programming abstraction, and scalability while addressing common design issues associated with self-adaptive software, including: Why adapt the software? When is adaptation necessary? Where should changes occur? What artifacts need to be modified? How is the adaptation performed? (Rosa et al., 2020; Razzaque et al., 2016; Krupitzer et al., 2015; Salehie; Tahvildari, 2009).

Furthermore, the development of such middleware systems needs to consider the particularities of IoT environments, such as constrained resources (e.g., disk space, memory, and processing power), power consumption, dynamic changes, diversity of applications, and heterogeneity of devices (Razzaque et al., 2016). A key aspect is successfully combining adaptability concerns with these particular characteristics of IoT environments. Dealing with this aspect is crucial for creating effective middleware solutions operating efficiently in diverse and challenging settings.

Despite ongoing efforts to develop self-adaptive IoT middleware systems, IoT applications are subject to various uncertainties during operation, which increases the complexity of implementing middleware systems capable of effectively addressing all these uncertainties. Moreover, the uncertainties encountered may require distinct adaptation strategies. Therefore, a key research aspect of this thesis is enabling multiple comprehensive adaptation mechanisms to handle as many uncertainties as possible in IoT environments. The approach must provide the ability to use customizable mechanisms that can be selected and switched based on the specific needs of IoT applications, thereby enhancing the overall effectiveness of the middleware in addressing diverse uncertainties.

In this context, the main objective of this thesis is to propose a middleware-based solution to facilitate the development and execution of self-adaptive middleware systems for IoT that address uncertainty in IoT applications. The research hypothesis is that self-adaptive middleware systems offer an adequate approach to managing these uncertainties. To achieve this, the proposed solution needs to enable the development of customizable and dynamically adaptable self-adaptive IoT middleware, allowing it to evolve in response to changes and uncertainties in IoT environments.

This adaptability involves adjusting to varying environmental conditions, enhancing application functionalities, and addressing changes. The solution incorporates adaptation mechanisms that ensure the middleware can handle a wide range of uncertainties during application operation, guaranteeing the proper functioning of IoT applications in diverse and unpredictable settings.

1.3 PARTIAL EXISTING SOLUTIONS

Developing middleware for IoT has been extensively investigated but remains an active field due to challenges such as distribution, heterogeneity, interoperability, and communication. Despite advancements, addressing the dynamism and resource limitations of IoT environments continues to be a significant open issue (Taconet et al., 2023; Medeiros; Fernandes; Queiroz, 2022; Razzaque et al., 2016).

Recent studies have explored self-adaptive systems, with IoT emerging as a critical area for their application (Pekaric et al., 2023; Wong; Wagner; Treude, 2022). Middleware for IoT has become an essential approach for integrating self-adaptive features into these systems.

Existing IoT middleware solutions remain static, either have no support for adaptation, or provide limited adaptability, focusing only on specific application- or network-level goals. Moreover, these solutions frequently fail to address the dynamic and unpredictable nature of IoT environments.

This thesis categorizes existing IoT middleware into three groups: off-device IoT middleware, device-based IoT middleware, and adaptive IoT middleware, with the latter being the most significant for the research. Off-device IoT middleware (Agostinho et al., 2018; Joseph et al., 2017; Elkhodr; Shahrestani; Cheung, 2016) typically operates in the cloud, handles distribution aspects, and requires a gateway for device integration.

Device-based IoT middleware systems such as those using MQTT (e.g., Mosquitto (Light, 2017) and HiveMQ[1]), run on devices but lack the adaptability needed to handle IoT's dynamic nature. They have static configurations, i.e., they do not support adaptations. They cannot handle the dynamics of IoT environments, as updates and reconfigurations after deployment are only possible by stopping the device.

Finally, adaptive IoT middleware systems include some adaptive capabilities. However, such adaptations target specific adaptation goals and fail to address the broader uncertainties inherent to IoT systems.

For example, solutions focused on adaptation, such as ARM (Achilleos et al., 2017), SAMSON (Portocarrero et al., 2016), and InteropAdapt (Mohalik et al., 2016), are designed to enhance interaction and efficiency by adapting application parameters or structure. These solutions typically incorporate runtime adaptations to adjust to changing dynamically environments. For instance, some systems emphasize distributed IoT applications, facilitating dynamic adaptation through modular components that can be adjusted to meet immediate needs. These adaptations often include saving energy consumption, fine-tuning communication protocols, ensuring device interoperability, improving system performance and extending the longevity of devices, particularly in wireless sensor networks.

Similarly, some middleware systems focused on adaptation at the application level, focusing on user interface (Uribarren et al., 2008; Park; Song, 2015) aim to enhance user experience and ensure that applications remain relevant to users' needs. These solutions dynamically adjust mobile application interfaces and IoT applications based on user behavior and contextual changes. This adaptation ensures the user interface is continually optimized for the best

[1] https://www.hivemq.com/

possible experience, responding to user preferences and shifting environmental factors.

There are middleware solutions (Pradeep; Krishnamoorthy; Vasilakos, 2021; Hassan et al., 2023) that focus on contextual and integration adaptation; the emphasis shifts toward ensuring interoperability across various IoT systems and applications. AUM-IoT (Pradeep; Krishnamoorthy; Vasilakos, 2021), for example, adapts services based on changes in context, managing different domains to ensure optimal performance. Similarly, PlanIoT (Hassan et al., 2023) focus on adapting to evolving requirements, such as QoS or resource availability, facilitating seamless integration across various devices and services. These solutions are vital to maintaining fluid data exchange and ensuring diverse IoT systems can work together effectively, even as conditions change.

To adapt to the network level, solutions such as PlanEMQX (Hassan et al., 2024), MPaS (Ahmed, 2022), Ermis (Peros; Joosen; Hughes, 2021), and EMMA (Rausch; Nastic; Dustdar, 2018) aim to enhance the efficiency of data management and communication. These systems implement strategies like QoS planning, fractal replication, and adaptive sensor sampling to optimize data flow and network efficiency. Such approaches are vital in large-scale IoT deployments, where balancing system responsiveness with resource use is essential to maintaining long-term functionality.

Finally, some solutions focus on specific environmental adaptations to address unique challenges within particular domains. AquaEIS (Han; Mehrotra; Venkatasubramanian, 2019), for example, adapts water infrastructure networks to prevent system failures and optimize resource management, making water systems more resilient and efficient. Similarly, ImmunoPlane (Jung et al., 2024) designs adaptive deployment plans that minimise potential failures and optimize resource usage in IoT systems, ensuring greater robustness and fault tolerance in specific applications.

Currently, no solution handles the different uncertainties that arise during the operation of the IoT systems. This limitation significantly increases the complexity of developing adaptive middleware capable of addressing uncertainties across different layers of the IoT architecture.

1.4 MIDDLEWARE EXTENDIFY

This thesis proposes Middleware Extendify (MEx)[2], a comprehensive solution designed to assist developers in implementing and executing self-adaptive middleware systems tailored for IoT environments. MEx consists of a *framework* and an underlying *execution environment*. The *framework* facilitates the development of self-adaptive IoT middleware systems, and the *execution environment* manages the functioning and adaptations of the middleware system and applications relying on it.

MEx's *framework* uses software architecture principles as an enabling technology for developing adaptive middleware systems and incorporating adaptability issues. MEx simplifies the development process by providing a library of pre-implemented middleware components and an Architecture Description Language (ADL), namely Python-based Architecture Description Language (*p*ADL), which allows developers to define middleware software architectures using middleware components belonging to the library. Developers build architectures as a collection of loosely coupled components having well-defined functionalities. Such organizational strategy is widely recognized as facilitating adaptation (Rosa; Campos; Cavalcanti, 2017; Garlan; Schmerl; Cheng, 2009a).

Components of the MEx's *framework* provide functionalities of message-oriented middleware systems (Goel; Sharda; Taniar, 2003), structured using the publish/subscribe pattern (Tarkoma, 2012), which is commonly found in IoT applications (Al-Fuqaha et al., 2015). The middleware components of MEx are specifically designed to run directly on IoT devices, possessing an awareness of IoT 's resource limitations and the ability to evolve dynamically. Developers implement these components using low-cost, low-power, and low-CPU-consuming operations whenever possible. MEx is customizable and extensible, allowing developers to combine components to create middleware systems in diverse ways. Developers can also implement and integrate new components as needed.

MEx's underlying *execution environment* manages the execution and adaptation of the middleware systems and applications built atop them. Developers only need to deploy the middleware architecture, its components, and associated applications into this *execution environment* for their operation and adaptation.

Regarding adaptations, the *execution environment* includes a managing system that acts

[2] MEx is an acronym for "*M*iddleware *Ex*tendify", where "Extendify" is formed by blending the words "extend" and "modify."

as an adaptation manager. This managing system manages all actions related to adaptive capabilities at the middleware and application levels during runtime. It triggers adaptations through adaptation mechanisms based on predefined goals to address uncertainties, which may arise from user goal variations, resource availability fluctuations, or changes in the application context.

Different uncertainties may require distinct adaptation goals. Hence, the managing system is a central hub for adaptation mechanisms to deal with as many uncertainties as possible. Hence, MEx supports various customizable adaptation mechanisms that can be switched to enhance middleware performance, meet application demands, and handle continuous changes and uncertainties at runtime. This flexibility allows for integrating new adaptation mechanisms, making the *framework* customizable and adaptable as IoT environments evolve.

The adaptation mechanisms, operating at the application level, enable the (re)configuration of execution parameters without stopping or recompiling the application, enhancing operational efficiency and adjusting to environmental conditions or behavioral changes based on context. At the middleware level, dynamic changes focus on improving middleware operations to respond to evolving conditions. These enhancements may involve fixing bugs, adding new functionality, or updating the middleware with the latest improvements.

1.5 SUMMARY OF CONTRIBUTIONS

The contributions of this thesis are summarized as follows:

- **Middleware framework**: MEx is the main contribution, a framework designed to simplify the development of self-adaptive middleware systems for IoT. This simplification is achieved through pre-implemented and loosely coupled middleware components so that developers only have to define a high-level artifact of the implemented middleware, i.e., software architecture. MEx also supports adding customized components, enabling developers to implement various self-adaptive middleware configurations by reusing existing components. Leveraging software architecture principles, MEx enables developers to easily define middleware architectures.

  The development of MEx followed a design science research approach (Lacerda et al., 2012), iteratively refining its architecture and adaptation capabilities based on experimental validation. MEx was evaluated in a real practical IoT scenario (see Chapter 5), demonstrating its effectiveness in managing uncertainties, adjusting energy consumption, and enabling dynamic middleware adaptation.

- **Architecture Description Language**: MEx includes *p*ADL, a Python-based architecture description language designed to describe and facilitate the implementation of self-adaptive middleware architectures. It serves as a tool for outlining the structure of middleware systems, aiding in the development process. Developers only need to use it to define and deploy different self-adaptive middleware architectures in the execution environment. The results showed that *p*ADL simplifies the definition and deployment of self-adaptive middleware architectures (see Chapter 5), reducing development effort, requiring no programming from the developer, and improving maintainability.

- **Adaptive Middleware Execution Environment**: MEx provides an execution environment responsible for executing IoT applications and their self-adaptive middleware systems. This environment manages applications and orchestrates the adaptation process, ensuring smooth operation in dynamic IoT environments. As a key contribution of this execution environment, MEx currently includes three adaptation mechanisms: *Evolutive*, *Duty Cycle Adaptation*, and the *Time-based Duty Cycle Adaptation*. The *Evolutive* mechanism allows structural adaptation by allowing the middleware to be continuously updated, incorporating new features, fixing bugs, or enhancing performance. The other two mechanisms are parametric, enabling the adjustment of application parameters to save energy—a critical uncertainty for IoT applications. In addition, MEx provides developers with the flexibility to integrate custom adaptation mechanisms, thereby expanding its capacity to meet various adaptation goals and address different uncertainties.

Considering the contributions of MEx, particularly in the dynamic nature and the uncertainties they face, this thesis presents additional contributions that show the capabilities of MEx.

- **AquaMOM**: A self-adaptive smart water consumption monitoring system designed to assist humans (e.g., customers) in improving efficient water conservation in challenging environments, such as semi-arid regions. Leveraging MEx's capabilities, AquaMOM utilizes the IoT device to collect, transmit data, and automatically adapt, focusing on managing water and energy consumption, ensuring efficient use, and effective monitoring.

- **IoT Device**: A specialized low-cost IoT device equipped with sensors for measuring water level and energy consumption was designed and built. This device can execute MEx, run IoT applications, and connect to the Internet to transmit data, serving as essential components in IoT systems.

The results of this research were validated through experiments (see Chapter 5), including the deployment of AquaMOM, performance benchmarks of the adaptation mechanisms, and analysis of middleware overhead. These evaluations confirmed that MEx effectively supports self-adaptive IoT middleware while maintaining low computational costs. Specifically, AquaMOM demonstrated its ability to reduce water waste and energy consumption by dynamically adjusting operational parameters based on environmental conditions.

The IoT device was also designed and tested to run MEx efficiently on low-power hardware. Performance tests confirmed that it can collect and transmit data with minimal energy consumption while executing self-adaptive middleware operations.

1.6 THESIS ORGANIZATION

The remaining chapters are organized as follows:

- **Chapter 2** introduces the basic concepts necessary for understanding the research: IoT, middleware for IoT, self-adaptive systems, and software architecture.
- **Chapter 3** presents details of MEx, including its design decisions, software modules, architecture, components, and adaptation mechanisms. In addition, the details of implementation and operation are presented.
- **Chapter 4** presents AquaMOM an IoT application that uses MEx capabilities. It is used as a proof-of-concept of how MEx can be utilized in practice.
- **Chapter 5** presents the experimental evaluation conducted with the MEx solution and its results.
- **Chapter 6** discusses related work on self-adaptive middleware for IoT and self-adaptive system solutions for IoT, which have contributed to the development of MEx.
- **Chapter 7** concludes the thesis, presenting the main contributions to self-adaptive middleware for IoT. It also discusses limitations of the MEx and possible future work.

2 BACKGROUND

"Ideas improve. The meaning of words participates in the improvement. Plagiarism is necessary. Progress implies it. It embraces an author's phrase, uses his expressions, erases a false idea, and replaces it with the right idea."

—Guy Debord

This chapter introduces the concepts necessary for understanding MEx. It initially gives the basic definitions of IoT, followed by the elements of self-adaptive systems and middleware. Finally, it presents the definition of software architecture.

2.1 INTERNET OF THINGS

The Internet of Things (IoT) has been defined in various ways by various authors (Atzori; Iera; Morabito, 2010; Bandyopadhyay et al., 2011; Razzaque et al., 2016; Cruz et al., 2018; Cavalcanti et al., 2024). In simple terms, IoT connects physical and virtual devices, such as sensors, actuators, smartphones, vehicles, home appliances, and software systems, enabling them to work together over the Internet. These connected devices are often referred to as *things* that can interact with each other anytime, anywhere, creating services and applications to achieve shared common goals. This connectivity has introduced a new dimension to information and communication technologies, allowing not only communication at any time, anywhere, and with anyone but also with anything (ITU-T Study Group 20, 2012)

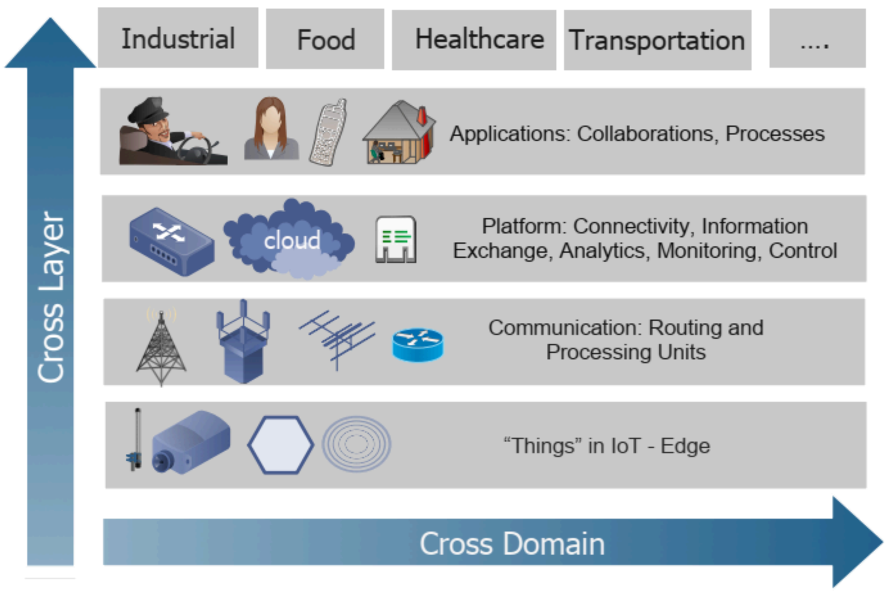
IoT has received great attention from academia and industry due to its potential to improve various aspects of society and interact with people, thereby enhancing infrastructure efficiency and quality of life (Nižetić et al., 2020a). IoT advances have enabled developments of smart homes (Agostinho et al., 2018), smart cities (Costa et al., 2022), digital health (eHealth) (Mukhopadhyay; Sreenadh; Anoop, 2020), smart water management (Singh; Ahmed, 2021) and Industrial IoT (IIoT) (Sisinni et al., 2018)).

The diversity of applications results in the interconnection of different devices and communication technologies, creating challenges, such as limited interoperability, lack of standardization, and uncertainties in these environments. These factors complicate IoT application development, as different architectures, functionalities, and components are often without standardization, regulations, and security (Medeiros; Fernandes; Queiroz, 2022; Al-Fuqaha et al., 2015).

2.1.1 IoT Architecture

While no single globally accepted IoT architecture exists (Burhan et al., 2018), various proposals have been presented (Karie; Sahri; Haskell-Dowland, 2020; Khan et al., 2012; Yang et al., 2011; Wu et al., 2010). In this thesis, the IoT architecture consists of four layers, as shown in Figure 2.1.

Figure 2.1 – IoT Architecture


Industrial
Food
Healthcare
Transportation
....
Applications: Collaborations, Processes
cloud
Platform: Connectivity, Information Exchange, Analytics, Monitoring, Control
Communication: Routing and Processing Units
"Things" in IoT - Edge
Cross Layer
Cross Domain

Source: (Weyns; Ramachandran; Singh, 2018)

The *Business layer* (top), although not explicitly indicated, represents the IoT domains, such as Industrial, Healthcare, Smart City, Smart Water, among others. It manages overall activities within these domains, including the development of business models, flowcharts, and ensuring application integrity (Burhan et al., 2018; Al-Fuqaha et al., 2015; Khan et al., 2012). Additionally, it also enables decision-making and monitoring of other layers, e.g., by implementing solutions that compare the output of each layer with the expected service results (Burhan et al., 2018; Al-Fuqaha et al., 2015).

The *Application layer* encompasses the applications and services provided by IoT applications. These applications must be reliable to meet diverse user needs in various areas, such as smart homes, smart cities, transportation, smart water management, healthcare, and IIoT (Nižetić et al., 2020b; Kavre; Gadekar; Gadhade, 2019; Khan et al., 2012).

The *Platform layer*, also known as *the Middleware layer*, hides the complexity of distribution among IoT applications. It enables interoperability and integration of devices and applications and hides heterogeneity. Additionally, the middleware can also process data and events from IoT devices, provide data management services (e.g., collecting and maintaining data), make decisions regarding these data (e.g., storing or sending them to a requesting application), and ensures scalability and security (Nižetić et al., 2020b; Burhan et al., 2018; Razzaque et al., 2016).

The *Communication layer* (or *Network layer*) handles data transmission between IoT devices, services and applications. It also gets data produced by the things layer and sends it to the information processing system. This layer can also function as a gateway, supporting various communication technologies, such as Radio Frequency Identification (RFID), Bluetooth, Long Range (LoRa), ZigBee, 5G, and Wi-Fi. Other functions, such as processing and managing data in the cloud, can also be performed in this layer (Nižetić et al., 2020b; Burhan et al., 2018; Al-Fuqaha et al., 2015).

The *Things layer*, also known as *Perception layer* (or *Device layer*) includes IoT devices responsible for data collection and processing, e.g., sensors, actuators, or micro-controllers for measuring temperature, humidity, weight, and so on. Data are measured, digitized, and passed to the communication layer for further processing (Nižetić et al., 2020b; Burhan et al., 2018; Al-Fuqaha et al., 2015).

2.1.2 IoT Uncertainty

Another characteristic of IoT is uncertainty (Weyns, 2021; Magruk, 2015), as it experiences continuous changes, primarily due to context modifications. Uncertainty refers to any unanticipated event, deviation, change, or error that unexpectedly manifests in a system, making predicting the system's behavior challenging (Hezavehi et al., 2021; Baresi; Ghezzi, 2010). An indication of uncertainty occurs when the results of ongoing activities cannot be determined with absolute confidence. In this sense, uncertainty concerns changes challenging to estimate or events whose probability cannot be predicted because the available information is too limited (Magruk, 2015; Calinescu et al., 2012; Baresi; Ghezzi, 2010).

Due to their complexity, high dynamism, inherent resource constraints, and integration with other systems, IoT applications are especially susceptible to several sources of uncertainty. These uncertainties can appear at different layers of the IoT technology stack during operation, requiring considerable effort in management (Weyns; Ramachandran; Singh, 2018; Magruk, 2015).

IoT applications usually have several kinds of elements, such as sensors and actuator things, storage and data processing units, compute servers, and communication technologies (Sethi; Sarangi, 2017). Each of these elements represents a potential source of uncertainty. For example, the heterogeneity of things may require diverse processing capabilities and support for different communication protocols (e.g., Wi-Fi and Bluetooth). The heterogeneity can also introduce uncertainty through measurement errors, environmental variations, packet loss in the network, and thing malfunctions. The dynamism of IoT applications can also introduce uncertainty related to the evolution of software. New components may become available, enabling the system to adapt to the evolving capabilities of IoT applications, devices, and protocols while maintaining compatibility and optimizing performance. Continuous middleware updates are essential to manage these uncertainties, ensuring the system remains adaptable, resilient, and effective as the IoT environment evolves.

To better illustrate the uncertainty of IoT environments, consider an IoT application that monitors water. Each sensor transmits data directly to the IoT gateway when in range. Otherwise, the sensor can relay data through other sensors until it reaches the gateway. In such a case, this application demands reliable communication with low latency and transmission power and a selected path for relaying packets towards the gateway. Ensuring the required communication quality is challenging, given that the application is susceptible to various uncertainties. Two primary uncertainties include network interference, encompassing factors like weather conditions, and wireless signals such as Wi-Fi in the neighborhood, which may lead to potential packet loss. The second uncertainty involves fluctuating traffic loads influenced by the rate of data generation or network congestion, which can also introduce uncertainty. Furthermore, changes in application requirements, such as adding user feedback dashboards or alert systems (e.g., email or Short Message Service (SMS)), can introduce new uncertainties by altering the original system design.

Finally, a typical source of uncertainty in IoT applications is the need for devices to remain operational for long periods, often powered by batteries. Power consumption is a critical uncertainty in this context, as these systems operate in unpredictable environments, and replacing batteries can be costly or sometimes even impossible (Cavalcanti; Hughes; Rosa, 2023; Weyns, 2018). Moreover, the applications themselves can be subject to change, which may affect the configuration of underlying layers and cause uncertainties in the system's state and surrounding environment. Furthermore, when several systems communicate, uncertainty caused in a particular system may have a cascading effect and threaten other interconnected systems (Weyns, 2018; Magruk, 2015).

2.2 SELF-ADAPTIVE SYSTEMS

A Self-Adaptive System (SAS) adapts its behavior or structure at runtime in response to system or environmental changes. Adaptations allow the system to evolve or even fix bugs during execution, ensuring it continues to achieve its goals with minimal or no human intervention while maintaining a certain QoS (Wong; Wagner; Treude, 2022; Alfonso et al., 2021; Weyns, 2020; Weyns; Ramachandran; Singh, 2018; Krupitzer et al., 2015; Kephart; Chess, 2003).

The concepts of SAS emerged as a response to the software complexity crisis at the beginning of 2000 (Sinreich, 2005; Kephart; Chess, 2003). As systems became more interconnected, distributed, and heterogeneous, developers found it increasingly difficult to predict and design interactions among components, leading to system failures and issues with hardware and software. These complexities required highly skilled system developers to install, configure, operate, tune, and maintain the systems. The development of SAS relieves developers from the maintenance burden, which requires human intervention and is both costly and time-consuming (Salehie; Tahvildari, 2009). Self-adaptive systems help solve this maintenance problem autonomously and at runtime.

Self-adaptation is essential for configuring and maintaining IoT applications, which are highly dynamic and prone to uncertainties. IoT applications need to operate continuously in environments that frequently change their conditions. Unexpected events can occur anytime, potentially compromising QoS (Alfonso et al., 2021; Patel; Ali; Sheth, 2017). In this context, the SAS collects data about itself and its environment and uses those data to reason and adapt itself to meet the required QoS (Weyns; Ramachandran; Singh, 2018; Salehie; Tahvildari, 2009; Kephart; Chess, 2003).

More formally, the adaptation can be understood as defined in the following equation:

$$S, D \models R \tag{2.1}$$

where, $\models$ denotes satisfaction or logical consequence, S represents the system specification, D denotes the domain assumptions, and R represents the system requirements. This equation means that the system specification S and the domain assumptions D together satisfy the system requirements R.

In software evolution, changes in requirements (R) as new business rules are usually handled through perfective maintenance and done offline through human intervention during the development phase. Changes in the specification (S) must accommodate evolving requirements (R) or adapt to changing domain assumptions (D). These assumptions (D) are highly uncertain and can change dynamically. In this case, changes are done by the software itself; they use sensors that monitor the environment, detect such changes and trigger self-adaptive mechanisms. The software's ability to meet its requirements (R) depends on the accuracy of its domain assumptions (D). These assumptions are defined in terms of adaptation goals and need to be corrected due to flawed analysis or environmental changes; the requirements might no longer be satisfied, necessitating adaptation to the software.

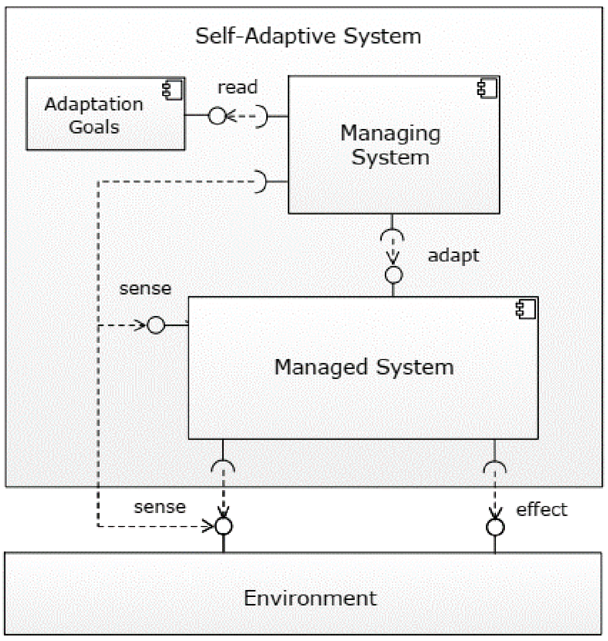
Finally, SAS has four self-management properties that a self-adaptive system can fulfill: self-configuration, self-healing, self-optimization, and self-protection. *Self-configuration* refers to systems that automatically and dynamically reconfigure themselves in response to changes by installing, updating, integrating and composing software entities. *Self-healing* describes systems capable of detecting, diagnosing, and reacting to errors and failures, including the ability to predict potential problems and take proactive actions to prevent a failure. *Self-optimization* involves systems continuously seeking ways to improve and evolve their functionalities, quality, and resource utilization. Finally, *self-protection* systems can defend themselves against malicious attacks or failures.

2.2.1 Model for Self-Adaptive Systems

Weyns (2018) introduced two fundamental principles that complement each other and help define a self-adaptive system. The first principle, the *External* principle, defines that a SAS can autonomously handle uncertainties and changes within the system, its goals, or its environment with minimal or no human intervention. The second principle, the *Internal* principle, mentions that a SAS consists of two distinct parts. The first part is responsible for the domain concerns (i.e., business logic) and interacts with the environment. The second part manages the first part and is responsible for the adaptation concerns.

Adopting these principles, Weyns (2018) proposed a conceptual model for a self-adaptive system, as shown in Figure 2.2. This model describes the basic concepts and elements of a self-adaptive system, its components, and the relationship between them.

Figure 2.2 – Conceptual model of a self-adaptive system



Source: (Weyns, 2018)

The first component is the *Environment* that comprises external elements interacting with the self-adaptive system. The environment can include physical and virtual elements where the effects of adaptation can be observed and evaluated. In IoT environments, these elements might be devices and applications or the broader environment in which the system operates, e.g., smart cities, smart homes, smart water systems, and so on. The environment can be sensed and effected through sensors and effectors, respectively. However, as the SAS does not control the environment, there may be uncertainty regarding what is sensed or what the outcomes will affect.

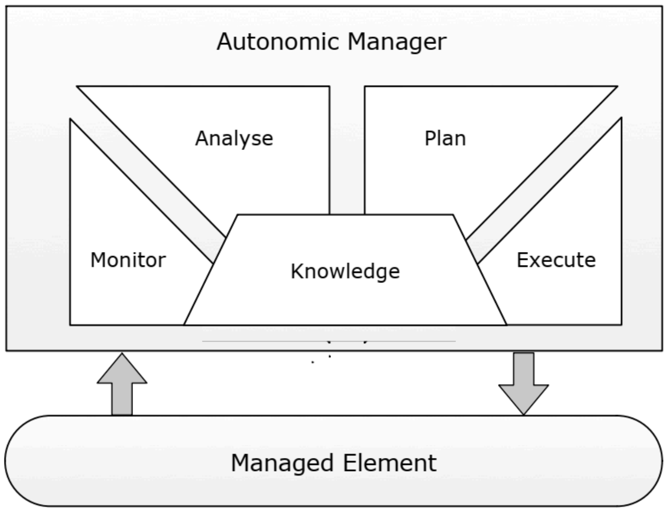
The second component of the model is the *Managed System*. It is part of the self-adaptive system that implements the application's domain functionality (business logic) and suffers the adaptation processes. To support adaptations, sensors must be used to collect information, and actuators to execute the adaptations. It is essential to ensure that adaptations in managed systems are safe, meaning they should occur when the system is idle or in stable conditions to avoid interfering with its functionality, i.e., the system is in a quiescence state (Kramer; Magee, 2007).

The third component is the *Adaptation Goals*, usually related to the quality and proper functioning of the managed system. If a goal is violated, an adaptation should be triggered. For example, a self-optimisation goal might specify a maximum execution time. Similarly, a self-configuration goal could ensure that the managed system is always up to date. It is important to note that adaptation goals can change over time. Adding or removing goals during operation requires updates to the managing system and may also necessitate updates in sensors and effectors.

The last component is the *Managing System*, responsible for implementing the adaptation logic and ensuring that the managed system meets its adaptation goals. To achieve this, the managing system contains subsystems that monitor and analyze both the environment and the managed system to decide the need for adaptation. To work autonomously with minimal human intervention, the managing system is typically implemented using autonomic feedback control loops, known as feedback adaptation loops (Alfonso et al., 2021; Weyns, 2020; Muccini et al., 2018; Salehie; Tahvildari, 2009; Kephart; Chess, 2003). These loops have been identified as crucial elements in developing self-adaptation systems. Existing control loops include Observe, Orient, Decide, Act (OODA) (Muccini; Moghaddam, 2017), Cognitive Cycle (sensing, analysis, decision, action) (Muccini; Moghaddam, 2017) and the Monitor-Analyzer-Planner-Executor + Knowledge (MAPE-K) (Sinreich, 2005).

The MAPE-K is the most popular control loop (Muccini et al., 2018; Muccini; Sharaf; Weyns, 2016) and consists of the elements shown in Figure 2.3.

Figure 2.3 – MAPE-K



Source: (Sinreich, 2005)

The *Monitor* element collects data from the managed system and its environment. It can also aggregate and filter this data to align it with the adaptation goals for analysis. Additionally, the *Monitor* updates the *Knowledge* base with newly collected data.

After collected, monitored data are forwarded to the *Analyzer*. The *Analyzer* evaluates the collected data and determines if an adaptation is needed. It identifies when the managed system is not meeting adaptation goals. Common techniques used by the *Analyzer* include directly checking system parameters, time series analysis, and rule engines. The analysis relies on data stored in the *Knowledge*. If an adaptation is necessary, the *Analyzer* informs this need to the *Planner*.

Having received the Analyzer decision, the *Planner* creates an adaptation plan for the managed system. The plan can range from a simple command action to change a variable to a series of procedures that must be executed in the managed system. Once the plan is created, it is sent to the *Executor* responsible for executing it.

The *Executor* performs the required adaptations in the managed system. Adaptations may involve changing the managed system's parameters, components, or both.

Finally, the *Knowledge* base is a repository that stores data about the whole adaptation process. This repository is shared between all MAPE-K elements.

2.3 MIDDLEWARE SYSTEMS

Middleware facilitates interaction or communication between applications, networks, or operating systems in distributed systems. It hides the complex connections required in such systems, easing the implementation process for developers (Schmidt; Buschmann, 2003; Bishop; Karne, 2003; Bernstein, 1996). This way, middleware simplifies the complexities of developing distributed applications by managing communication, heterogeneity, and interoperability issues. It allows developers to focus on application-specific concerns, such as business logic, rather than dealing with complex and error-prone details associated with underlying programming infrastructure (Cavalcanti; Carvalho; Rosa, 2021; Schmidt; Buschmann, 2003; Bishop; Karne, 2003).

There are various models of middleware (Bishop; Karne, 2003), each designed to facilitate communication and interoperability between distributed applications. One of these models is *Procedure Oriented Middleware*, which uses synchronous methods to request remote service execution. This model employs client stubs and server skeletons to manage interactions between applications. The client stub converts the procedure parameters into messages and sends them to the server, which converts them back into procedures for processing. This conversion process is called marshaling. After processing, the reverse process returns the results. This middleware model has advantages, such as using standard naming services and supporting multiple data formats. However, it is not very scalable or flexible due to its tight coupling with the procedure, since clients must wait until they receive a reply from the server before continuing (Menasce, 2005; Bishop; Karne, 2003).

Another model is *Object Oriented Middleware*, which supports synchronous and asynchronous communication between client and server objects. In this case, the client makes a method call on a remote object, and a local proxy marshals and sends the data to a broker. The broker contacts data sources, organizes and forwards the data to the server, which processes it and returns results. This model supports load management, scalability, and multi-threading but may require prelinked execution and wrapper code for legacy systems (Voelter; Kircher; Zdun, 2004; Zdun; Kircher; Völter, 2004; Bishop; Karne, 2003).

Finally, *Message-Oriented Middleware (MOM)* can be divided into Message Passing/Queuing and Publish/Subscribe. In Message Queuing, applications send messages using the client MOM. A MOM server picks these messages in a predetermined order and routes them via a message broker to the appropriate clients through a queue. The MOM server functions primarily as a message router and typically does not interact with the content of the messages. This approach enables asynchronous messaging. In the publish/subscribe pattern, which is event-driven, clients can act as publishers, subscribers, or both. Publishers send messages to topics managed by a broker. The broker manages topics, messages, and subscribers and notifies subscribers. Subscribers request or are notified of specific data from the publisher, sent via the broker (Rausch; Dustdar; Ranjan, 2018; Tarkoma, 2012; Bishop; Karne, 2003).

Middleware usually operates between applications and operating systems and provides transparencies in different aspects of distributed systems (Coulouris et al., 2011; Tanenbaum; Steen, 2006):

- **Access Transparency**: Users can access the application without knowing the system details, such as how data is represented or a resource is accessed.
- **Location Transparency**: Developers do not need to know where the resources are located; even if the resource is remote, it should appear local to the user.
- **Technology Transparency**: The underlying technology used in the system is hidden from users, including differences in operating systems, programming languages, plat-

forms, and communication technologies.

- **Concurrency Transparency**: The system allows multiple users to access the same resources simultaneously without conflicts, with little or no perception of other users.

In addition, middleware systems can also offer services, such as queuing, security, and concurrency control for applications built on top of them (Cavalcanti; Carvalho; Rosa, 2021). For example, it can encapsulate and enhance native Operating System (OS) mechanisms, providing reusable event (de)multiplexing, inter-process communication, and synchronization objects. These characteristics allow developers to create applications without hard-coding dependencies on specific locations, programming languages, OS platforms, or communication protocols.

2.3.1 Middleware for IoT

IoT middleware plays a crucial role in addressing challenges related to distribution, heterogeneity, interoperability, and communication in IoT environments. It facilitates communication between devices and applications (Razzaque et al., 2016). IoT middleware hides the heterogeneity of devices, making communication possible among them. In practice, IoT middleware provides connectivity for devices. It also enables interaction between various devices while working as a layer for data management, security and concurrency control, and scalability for IoT applications (Sethi; Sarangi, 2017).

IoT middleware is essential for several reasons: (1) it allows interoperability between heterogeneous devices belonging to different IoT domains; (2) it also acts as an abstraction layer for data communication and representation, allowing transparent communication between diverse applications and (3) it provides an Application Programming Interface (API) for communicating between the physical layer, providing the necessary services for applications while hiding the complexity of devices and services (Zhang et al., 2021; Kassab; Darabkh, 2020; Bandyopadhyay et al., 2011).

IoT middleware systems are versatile, functioning across various layers of the IoT application architecture (see Section 2.1.1). At the application layer, they can offer mechanisms that support efficient and secure processing of streaming data from many sensors. It also provides services and other functionalities linked to the application level, which help application developers add new components, integrate additional functionalities, and fine-tune their resource capabilities. These mechanisms may encompass several elements, such as a runtime environment and programming APIs.

At the platform layer, middleware simplifies the development process by integrating heterogeneous computing and communication things, managing resources to ensure acceptable QoS for all applications and environments, and promoting interoperability among diverse applications and services. IoT middleware manages the communication protocols, such as Wi-Fi and Bluetooth, within the communication layer. Finally, the thing layer abstracts the heterogeneity and details of things from different vendors, irrespective of their connectivity protocols, ensuring compatibility and seamless operation.

2.4 SOFTWARE ARCHITECTURE

Software Architecture is a crucial concept in software engineering, serving as a bridge between requirements and implementation. Although no universal definition exists, it is generally understood as a high-level abstract representation of the software's structure (Medvidovic; Taylor, 2010; Fuxman, 1999; Garlan; Shaw, 1993). *Software Architecture* is described using various methods, including informal diagrams, descriptive terms, module interconnection languages, domain-specific frameworks, and formal models (Mckenzie; Petty; Xu, 2004; Medvidovic; Taylor, 2000).

Software Architecture defines the system's organization, components, and interactions. As software systems grow in size and complexity, new challenges arise, extending beyond algorithms and data structures. These challenges include designing and specifying the overall system structure, including software organization, communication protocols, synchronization, data access, and functionality assignment.

In practice, *software architecture* is often developed by composing three essential elements: components, connectors, and configurations (Rosa et al., 2020; Medvidovic; Taylor, 2010; Medvidovic; Taylor, 2000). Components represent a system's computation units, such as application clients, servers, functions, or entire applications and data storage systems like databases. Connectors describe the interactions between software components, such as function calls, database queries, or pipelines. Finally, configurations describe information about the system, such as how components are connected, including any constraints, rules, settings, and non-functional properties. Architectural patterns guide the composition of these elements.

There are different ways of representing the system's architecture. Informal graphical notations like *box-and-line* diagrams are typically used. In these diagrams, boxes usually represent system components, while lines represent some data flow or control connections between the components. Although helpful, these diagrams usually lack detail (Medvidovic; Taylor, 2010; Mckenzie; Petty; Xu, 2004; Garlan; Shaw, 1993).

A more structured approach for describing a software architecture is Unified Modeling Language (UML), a graphical language that provides diagrams representing different aspects of a system. For example, class diagrams show structural aspects, while interaction diagrams represent behavioral aspects. Despite the availability of such tools, informal textual descriptions are still commonly used in practice, often referring to architectural patterns, such as *client-server*, without providing detailed explanations (Mckenzie; Petty; Xu, 2004; Medvidovic; Taylor, 2010).

Architecture Description Languages (ADLs) provide a more formal and precise representation. As the name suggests, an ADL is a declarative language that offers features like a defined syntax and a conceptual framework for describing software architectures (Rosa; Campos; Cavalcanti, 2017; Mckenzie; Petty; Xu, 2004; Garlan; Shaw, 1993). ADLs facilitate the validation and analysis of the architecture before implementation, making it easier to communicate with stakeholders and maintain the system over time. ADLs and their associated frameworks and tool-sets offer a solution to the limitations of informal methods by providing a formal approach to architecture-based software development. These languages and tools can be processed computationally to support the creation, analysis, and refinement of architectural specifications, ensuring a robust foundation for the software system.

Considering these characteristics, ADLs and their frameworks provide reusable infrastructures with well-defined components, allowing developers to customize systems to systematically meet specific goals and concerns. This capability allows some approaches to leverage software architecture concepts to guide self-adaptation logic during the system adaptation process, a practice referred to as architecture-based self-adaptation (Rosa et al., 2020; Rosa; Campos; Cavalcanti, 2017; Garlan; Schmerl; Cheng, 2009b). These approaches offer structured frameworks to monitor the system and its environment, reflecting observations in the system's architecture model, detecting opportunities for improvement, selecting actions, and implementing changes in a closed loop.

2.5 CONCLUDING REMARKS

This chapter presented the fundamental concepts necessary to understand the proposed MEx system. Initially, fundamental IoT concepts were introduced. Next, SAS concepts and middleware for IoT were discussed. Finally, the concept of software architecture was presented.

3 MIDDLEWARE EXTENDIFY

"You may learn something, and whether what you see be fair or evil, that may be profitable, and yet it may not. Seeing is both good and perilous. Yet I think, Frodo, that you have courage and wisdom enough for the venture, or I would not have brought you here. Do as you will!"

—J.R.R. Tolkien

This chapter presents MEx. Firstly, it presents an overview of the solution, followed by a discussion of the design principles considered in its development. Then, the main modules and components of MEx are detailed. Finally, implementation details are presented.

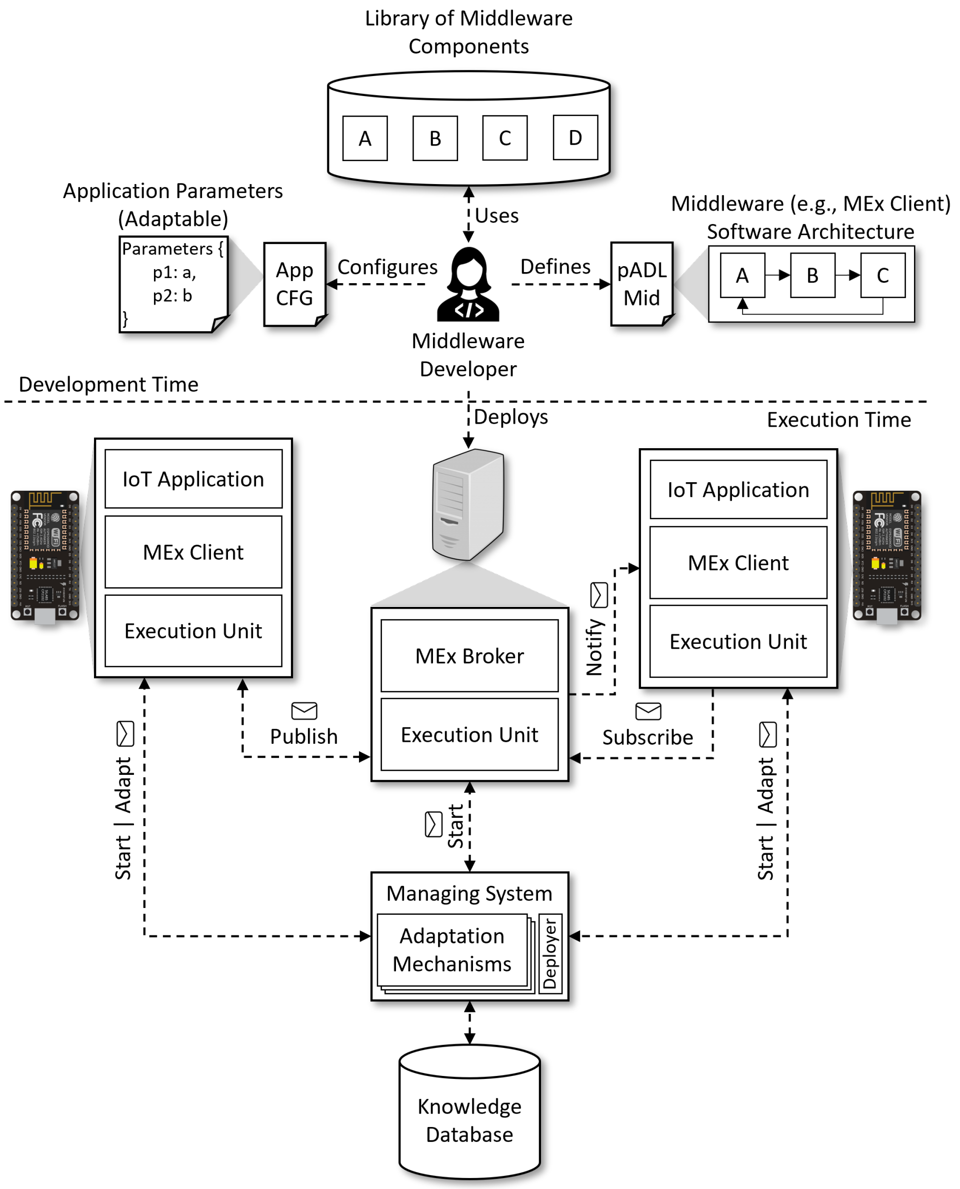
3.1 OVERVIEW OF MEX

MEx (*M*iddleware *Ex*tendify) is a comprehensive, customiZable and adaptable solution for designing and implementing self-adaptive middleware systems tailored to IoT. *Comprehensive* means that MEx offers tools and guidelines needed for developing and adapting these middleware systems and the IoT applications atop them. *Customizable* means that MEx allows the addition of new components and adaptation mechanisms, which can be added and removed at any time. *Adaptable* refers to the ability of middleware systems to be adjusted for different IoT contexts, combining components in various ways to create different types of self-adaptive middleware systems (Cavalcanti; Rosa, 2024).

MEx consists of a middleware *framework* and an underlying *execution environment*, which help developers create and manage self-adaptive middleware systems and distributed applications. Figure 3.1 shows a general overview of MEx, along with its support in developing (*Development Time*) and executing (*Execution Time*) self-adaptive middleware systems.

At *Development time*, MEx has a *framework* that simplifies the development of middleware systems. It uses software architecture principles (see Section 2.4) as an enabling technology for implementing the middleware and its associated adaptation mechanisms. The middleware is structured as a collection of loosely coupled components with well-defined func-

Figure 3.1 – General Overview of MEx



Source: Author

tionalities. This organizational strategy facilitates adaptations, as these components can be easily manipulated (e.g., swapped out) at runtime (Rosa; Campos; Cavalcanti, 2017; Garlan; Schmerl; Cheng, 2009a).

The *framework* provides a *Library of Middleware Components* used by *Middleware Developers* for implementing the middleware. This library includes pre-implemented middleware components supporting communication, distribution, and adaptation. It also has an ADL, named *p*ADL, for describing middleware software architectures. Developers only *use* components from MEx's library and *define* the software architecture using *p*ADL. MEx supports adaptability at the middleware and application levels. In this sense, application-level adaptations are parametric, so developers can also *configure* adaptable parameters adjusted at runtime.

The components in MEx have been designed for the development of message-oriented middleware (Goel; Sharda; Taniar, 2003) structured using the publish/subscribe pattern (Tarkoma, 2012), which has been widely used to support communication and distribution of IoT applications. Furthermore, MEx supports the automated deployment of software architectures into the execution environment. This automatic process is crucial for scalability and error reduction, as manual configuration of numerous IoT devices can be costly and error-prone. Thus, the middleware architecture, IoT applications, and configurations are stored in a database linked to each device's unique identifier. Upon connection, the device initiates the deployment process by sending a start message. Once the deployment concludes, the execution begins immediately.

At *Execution Time*, the *execution environment* manages the execution and adaptation of middleware and applications. It comprises five main modules: MEx *Client*, MEx *Broker*, *Execution Unit*, *Managing System*, and a *Knowledge Database*. Each module ensures efficient deployment, execution, communication, and adaption of middleware systems and applications.

The *Execution Unit*, a central module of MEx, manages the automated deployment and execution of middleware architectures and applications. This component triggers the deployment process and acts as the execution engine, managing the life-cycle of each component, including automatic loading, instantiation, starting, and execution. It also handles adaptations by sending *adapt* messages to implement runtime changes, such as adding, replacing, or removing components or adjusting adaptable parameters. Furthermore, it actively manages shared resources, fostering interactions among these components.

The MEx *Client* is the middleware component executed on IoT devices. It provides communication and distribution services to achieve interoperability between IoT devices and applications. MEx *Client* provides the traditional operations of a publish/subscribe middleware to applications executing in the device, e.g., *publish* and *subscribe*. The MEx *Broker* is the messaging service that mediates asynchronous message exchanges between MEx *Clients*. *IoT applications* built on MEx *Client* use the MEx *Broker* to *publish* messages without explicitly specifying recipients. Likewise, subscribers use the MEx *Broker* in the subscription process to receive messages. MEx *Broker* stores messages, manages topics and subscribers and *notifies* subscribers interested in a particular topic.

The *Managing System* is the main module in the cloud side of MEx. It is responsible for initializing the execution and coordinating all actions related to the adaptation process. It receives *start* messages from the *Execution Unit* and orchestrates automatic deployment into the IoT device. The *Knowledge Database* assists the *Managing System* in the initialization and adaptation process.

Managing System manages adaptations at middleware and application levels. Adaptation at the application level allows (re)configuration of execution parameters without stopping or recompiling the application. The dynamic changes of these parameters aim to improve the application's operation and adjust it to the environment conditions or its behavior based on the context. At the middleware level, dynamic changes are intended to improve the operation of the middleware and adjust it to changing environmental conditions. These middleware improvements include the ability to address bugs, add new functionality, or keep the middleware up to date with the latest advancements. It receives *adapt* messages from the *Execution Unit* and triggers adaptations using its *Adaptation Mechanisms*, which determine the necessary changes. These mechanisms are developed according to predefined goals to address uncertainties (see Section 2.1.2) inherent to IoT environments. These uncertainties may arise from modifications in user goals (e.g., varying user requirements and workloads), changes in operational environments (e.g., temporary loss of connectivity and degraded performance or security), and fluctuating resource availability, such as decreasing battery levels. However, different uncertainties may require other different adaptation goals. For this reason, the *Managing System* is designed as a central hub for *Adaptation Mechanisms*. It allows the integration of custom mechanisms according to the needs of the IoT environment, making MEx customizable and flexible as IoT environments evolve. Integrating different mechanisms is crucial for dealing with as many uncertainties as possible, as no single mechanism can be considered the best in all situations.

For example, in a smart water application for monitoring cistern consumption, a device monitoring the water level initially sends a *start* message to the *Managing System*. The *Managing System* then accesses the *Knowledge Database* and retrieves the necessary components and configurations (e.g., adaptation interval and adaptive mechanism) to deploy on the device. While executing, the device measures the execution cycle times and at the end of each cycle, the *Execution Unit* sends an *adapt* message. Different devices may require distinct adaptation mechanisms. Therefore, the *Managing System* instantiates the needed adaptation mechanism (e.g., a *parametric adaptation*) to trigger and execute adaptations. For instance, an adaptation might be adjusting the monitoring interval based on the cistern's level. Suppose the cistern's level is critical; the system might change the monitoring frequency to every minute.

Conversely, suppose the water level is at peace level. In that case, the system might change the frequency to every 60 minutes to save energy. Another potential adaptation mechanism could dynamically adjust the operation frequency based on the battery level.

At this point, it is worth noting that all adaptation mechanisms in MEx are implemented according to the MAPE-K (Monitor, Analyzer, Planner, Executor, and Knowledge Database) feedback loop (Sinreich, 2005) (see Section 3.4.4). The Monitor monitors the execution environment, and the Analyzer receives monitored data and decides whether an adaptation is necessary. If an adaptation is required, the Planner creates an adaptation plan, and the Executor carries it out. The *Knowledge Database*, also part of MAPE-K and shared among all elements, stores relevant information to support the adaptation process.

In the following sections, the design decisions of MEx (Section 3.2) are discussed, followed by the details of its development time (Section 3.3) and execution time (Section 3.4) support, along with implementation details (Section 3.5).

3.2 DESIGN DECISIONS

The following sections present design decisions established to guide MEx development. These decisions were based on a systematic review of related works and surveys on implementing self-adaptive middleware for IoT (Al-Fuqaha et al., 2015; Razzaque et al., 2016; Moghaddam; Rutten; Giraud, 2020; Zhang et al., 2021; Medeiros; Fernandes; Queiroz, 2022).

3.2.1 MEx Middleware is MOM

As mentioned in Section 2.3, middleware systems can vary in their models and implementations. Among these, MOM (see Section 2.3) is a crucial technology for today's IoT environments (Rausch; Dustdar; Ranjan, 2018; Razzaque et al., 2016; Al-Fuqaha et al., 2015). A message-oriented approach is a popular choice for developing IoT middleware for several reasons, which is why it was also selected for this thesis.

Firstly, the distributed nature of IoT applications means that devices are often in different locations and must communicate efficiently. MEx asynchronous communication allows devices to exchange data without requiring a direct connection, which is essential in scenarios where connectivity may be intermittent.

Secondly, MOM offers scalability, allowing new devices to be integrated into the architecture without the need for reconfiguration (Sheltami; Al-Roubaiey; Mahmoud, 2016; Razzaque et al., 2016). Each device using MEx client can independently send and receive messages, simplifying communication. This approach also enhances system resilience; if one device fails, messages can still be delivered to others, allowing the overall system to continue operating.

MOM provides a centralized message broker enabling decoupled device-to-device communication. MEx broker decouples publishers and subscribers, meaning that devices do not need to be aware of each other to communicate. This feature facilitates maintenance and updates without disrupting operations. Additionally, it is crucial for lightweight, resource-constrained IoT environments. It supports asynchronous communication, essential for distributed systems where components may move, fail, or disconnect from the network.

Moreover, a MOM allows for greater flexibility in integrating various types of devices and communication protocols, resulting in diverse technology choices (e.g., programming languages) and platform options (e.g., micro-controllers and Arduino) (Rausch; Dustdar; Ranjan, 2018; Razzaque et al., 2016; Al-Fuqaha et al., 2015). Lastly, it efficiently manages data generated by IoT devices, enabling dynamic data collection and processing. The MEx framework addresses several issues by supporting different programming languages, such as C, Python and JavaScript, implementing its communication protocol, and enabling communication with Message Queuing Telemetry Transport (MQTT).

In summary, the components in MEx support the development of message-oriented middleware systems (Goel; Sharda; Taniar, 2003) structured using the publish/subscribe pattern (Tarkoma, 2012). Its benefits include asynchronous communication, lightweight implementation, low resource consumption, scalability, decoupling, heterogeneity and distribution (Sheltami; Al-Roubaiey; Mahmoud, 2016; Razzaque et al., 2016). Additionally, MEx supports adaptability in IoT environments.

3.2.2 MEx Transparencies

MEx middleware provides transparencies (see Section 2.3) for IoT application developers. These transparencies release developers from dealing with distribution concerns, allowing them to focus on application-specific requirements. Challenges such as concurrent processing, cooperation between devices in different locations, network delays, failures, and device heterogeneity (e.g., varying programming languages, operating systems, and communication protocols) are addressed by transparencies offered by MEx.

In this sense, one of the critical transparencies offered by MEx is *location transparency* allows applications or developers to access resources without knowing their physical location on the network. MEx ensures that the location of resources does not affect how developers or applications interact with them. For example, in a water level monitoring system that sends alerts about issues such as water shortage or excessive consumption, MEx ensures messages and notifications are delivered to subsystems (e.g., Web applications) regardless of the physical location.

Another important transparency is *access transparency*, which abstracts the difference between accessing local and remote resources. MEx provides a simplified interface to applications. Then, applications exchange messages without specifying recipients or knowing how messages will be processed and received.

MEx also supports *technology transparency*, enabling applications to be developed in different programming languages, as it has been developed in Python, JavaScript, and C. Consequently, applications can execute on different platforms, such as Linux-based systems, embedded devices running FreeRTOS, and micro-controller environments, such as MicroPython. MEx also accommodates different hardware devices, including Personal Computer (PC), ESP8266, and Raspberry Pi, simplifying the development process by enabling interoperability among diverse computing and communication environments.

Finally, *concurrency transparency* ensures that resources can be accessed simultaneously without compromising the correctness of the processing results. While this mechanism ensures the integrity of concurrent operations, it may introduce overheads that affect QoS, such as increased latency or reduced throughput. Nonetheless, it facilitates effective resource management by balancing the needs of different applications and environments, helping to achieve performance and maintain acceptable QoS.

3.2.3 IoT-Driven Development

As mentioned in Section 2.1, the development of IoT applications has several challenges. Characteristics of IoT involve diverse hardware and software components, each with distinct development processes and constrained resources, and having highly dynamic and distributed interactions among multiple parties (Fahmideh et al., 2022).

Most research in IoT has focused on the technical and empirical aspects of IoT implementation (Fahmideh et al., 2022). MEx follows a technical approach, i.e., an implementation-oriented method, addressing characteristics of IoT environments, especially the resource constraints. The proposed approach involves developing, testing, iterating and adapting solutions based on researched concepts, refining them for improvement.

MEx implements a communication protocol between applications and includes a component for communication with MQTT, which has become a *de facto* standard for IoT applications (Al-Fuqaha et al., 2015).

Resource limitations are another critical challenge in IoT. While some IoT applications may eventually require more powerful computing resources for tasks like routing, switching, and data processing, many IoT devices are small, low-cost, and embedded, with significant constraints on processing power, memory, and communication. These limitations must be carefully considered during development to ensure that applications remain efficient and functional even on the most resource-constrained devices. For this reason, MEx prioritizes low-cost, low-power, and low-CPU operations, for example, by avoiding loops and list comprehensions, using local variables, and optimizing the code whenever possible.

3.2.4 Manage Multiple Uncertainties

To face the uncertainties inherent in IoT operations (see Section 2.1.2), MEx utilizes self-adaptation mechanisms based on adaptation goals to enhance resilience and adaptability in dynamic and resource-constrained environments.

For uncertainty related to application evolution, specifically component updates and compatibility, MEx uses a strategy of continuous updates and component replacement. The middleware is regularly updated to the latest component versions, improving compatibility and performance. When new versions are available, outdated components are automatically replaced, allowing the system to evolve without manual intervention.

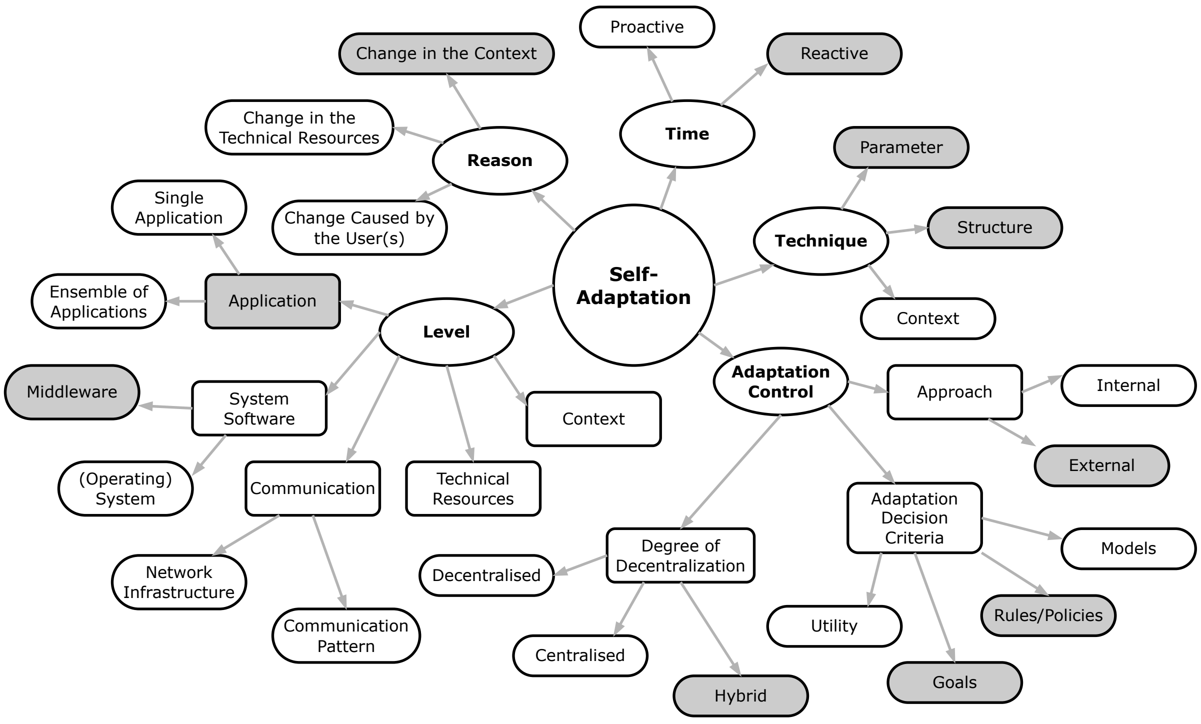
To face uncertainty related to device lifetime, MEx incorporates energy-saving mechanisms. The application reduces power consumption during idle periods or low battery levels, extending device operation. It also monitors the battery level, allowing dynamic adjustments to energy usage based on environmental conditions.

For uncertainty associated with environmental changes, MEx has an adaptive approach that adjusts operating parameters based on environmental changes. The parametric mechanisms modify settings (e.g., data collection intervals) to balance resource efficiency and data accuracy, ensuring reliable operation under diverse conditions.

3.2.5 Self-Adaptation Capability

The final design decisions concern the self-adaptation capabilities. Common issues associated with the design and development of self-adaptive systems include: *When to adapt*? *Why to adapt*? *Where to adapt*? *What to adapt*? and *How to adapt*?. The design decisions to address these questions are systematically organized using the taxonomy proposed by Krupitzer et al. (2015) shown in Figure 3.2.

Figure 3.2 – Taxonomy of Self-Adaptation



Source: (Krupitzer et al., 2015)

The issue *Why to adapt* concerns reasons for adaptation, which can be classified into three main categories: *Change in Context*, *Change in Technical Resources*, and *Change Caused by the User(s)*. *Change in Context* refers to environmental modifications, such as network instability or changes in application behavior. *Change in Technical Resources* involves adaptations due to resource limitations or failures, such as power loss. *Change Caused by the User(s)* refers to evolving user preferences, such as adding new features or replacing components.

In MEx, adaptation strategies were primarily developed for *Change in Context* and *Change Caused by the User(s)*. For example, MEx updates the middleware by replacing components to fix bugs or enhance performance and security. It also adjusts parameters, such as increasing the frequency of sensor readings in response to environmental changes. Although primarily focused on these aspects, MEx allows customization to address further adaptation reasons, such as *Change in Technical Resources*.

The issue *When to adapt* refers to the time the adaptation occurs. Adaptation can be *proactive* when the system anticipates and acts before undesirable events (e.g., performance issues) or *reactive* when the system responds after the event. MEx supports *reactive* adaptation and can be customized for proactive one. The choice between *proactive* and *reactive* adaptation depends on how the adaptation logic is implemented, and both approaches can coexist in MEx.

MEx provides three adaptation mechanisms (see Section 3.4.5). Two mechanisms adjust application parameters in response to context changes, and the third one adapts the middleware when a new version of a given middleware component becomes available. Moreover, MEx follows the quiescence principle (Kramer; Magee, 2007), ensuring that adaptations are only triggered under stable conditions, such as when components are inactive or have no pending tasks.

The issue *Where to adapt* refers to the location or level of adaptation in the system. According to the taxonomy, adaptation can occur at the *Application Level*, *System Software Level* (Middleware and Operating System), and *Communication Level*. In MEx, adaptations happen at the *application* and *middleware levels*. At the *application level*, adaptations are needed due to the dynamic nature of IoT applications, which often face context changes or evolving user preferences. For example, the application may adjust sensor reading frequencies based on environmental changes.

Adaptation is a natural choice at the *middleware level*, as MEx is a middleware framework. Here, adaptations ensure the middleware remains stable by replacing outdated components or integrating new versions to improve performance, security, or functionality. In both levels, MEx responds to changes while maintaining system stability and performance.

The issue *What to adapt* refers to the attributes or artifacts modified during adaptation. According to the taxonomy, adaptation techniques fall into three categories: *parameter*, *structure*, and *context*.

At the *application level*, adaptation involves parameter changes, such as adjusting sensor reading frequencies. At the *middleware level*, MEx allows structure adaptation, allowing dynamic replacement or reconfiguration of components at runtime. Components can be updated, replaced, or integrated to improve performance, security, or functionality. Additionally, MEx supports simultaneous replacement of multiple components.

Finally, *How to adapt* addresses how adaptation actions are implemented and the criteria that guide the decision-making process for adaptation. According to the taxonomy, two common approaches for implementing adaptation logic are *internal* (logic is integrated with the system) and *external* (logic is separated from the system).

The *external* approach is used in MEx, as proposed by Weyns, Schmerl et al. (2013) (see Section 2.2). This decision considered the context of IoT applications, where resource constraints demand a flexible and easily modifiable architecture. By separating the adaptation logic from the system, MEx can more easily adapt to changes and be customized without compromising system stability or performance.

Additionally, a metric is needed to determine how the system should adapt. These metrics include models, rules and policies, goals, and utility functions. In MEx, the adaptation logic follows the MAPE-K loop, where the Analyzer evaluates the system's state to identify adaptation needs.

MEx uses goals, rules, and policies as the primary metrics for making adaptation decisions. For example, adaptation goals are set to maintain system performance and stability, while rules and policies guide how specific adaptations are executed based on context or user requirements. However, MEx is designed to be flexible, allowing developers to customize the self-adaptation mechanisms and integrate additional metrics as needed to suit specific use cases.

Finally, the degree of decentralization of adaptation logic is also a concern. Centralized logic is often suitable for systems with few resources to manage. A decentralized approach can improve adaptation performance by distributing responsibilities across multiple components; each component has its adaptation logic, enabling various communication patterns. Hybrid approaches combine centralized components with decentralized elements, distributing the adaptation logic across subsystems.

A hybrid decentralized adaptation logic was chosen. This approach operates both on the device and externally, addressing the specific challenges of the IoT environment (Razzaque et al., 2016), such as the resource constraints of devices (e.g., to save computational and power resources of the devices) and fluctuating environmental conditions (Razzaque et al., 2016). This hybrid approach is based on decentralized control patterns for self-adaptive systems (Weyns; Schmerl et al., 2013).

In summary, MEx implements its adaptive capabilities by offering solutions to common design issues related to when, why, where, what, and how to adapt. The system provides mechanisms for adaptation at the application and middleware levels. The primary reasons for adaptation focus on changes in the application context, changes in technical resources and changes caused by the users. MEx supports reactive adaptation but can also be customized for proactive approaches. It uses external adaptation logic to ensure flexibility and ease of modification, which is particularly important for IoT applications with resource constraints. Finally, MEx has a hybrid decentralized approach to enhance adaptation performance by distributing adaptation logic across the system. This approach allows better handling of the dynamic and resource-limited conditions typical in IoT environments.

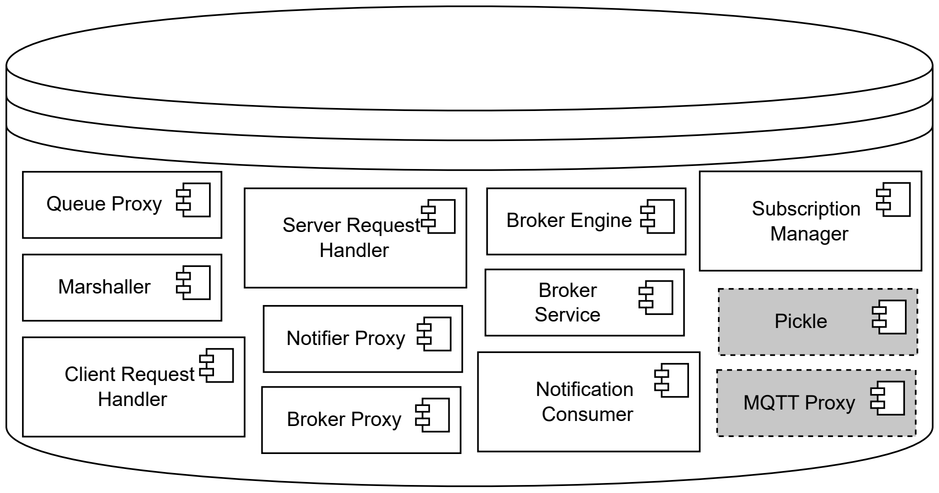
3.3 DEVELOPMENT TIME

As mentioned in Section 3.1, during the development time, MEx facilitates the implementation of self-adaptive middleware systems by providing a library of middleware components and an ADL. These elements will be detailed in the following sections.

3.3.1 Library of Middleware Components

The components library is a collection of pre-implemented, well-defined, and loosely coupled components that developers can (re)use to build adaptive middleware systems. The library contains twelve components, as shown in Figure 3.3: Queue Proxy, Marshaller, Client Request Handler, Server Request Handler, Broker Proxy, Broker Engine, Subscription Manager, Notifier Proxy, Notification Consumer, Broker Service, MQTT Proxy, and Pickle.

Figure 3.3 – Library of Middleware Components



Source: Author

These components are software blocks that provide essential functionality within a middleware system. For example, they offer interfaces for applications that use middleware services, handle data serialization and implement communication mechanisms. They were designed and implemented according to the remoting patterns (Voelter; Kircher; Zdun, 2004) and generic publish/subscribe components (Tarkoma, 2012), as shown in Table 3.1.

Table 3.1 – Middleware Components and Functions

Component	**Function/Purpose**
Queue Proxy	It is used for remote invocations and serves as an entry point to the middleware. This proxy provides publish/subscribe operations invoked by applications running on devices and executed remotely in the messaging service.
Marshaller	It marshalls local objects into a byte stream before sending them through the network. Similarly, it unmarshalls byte streams into objects after receiving them from the network.
Client Request Handler	It encapsulates communication issues on the client side, including opening and closing the network connections, sending requests, and receiving replies to/from the MEx Broker.
Broker Service	It is the MEx messaging service.
Broker Proxy	It implements a listening operation that dispatches the start request from the Broker Service to initiate the Server Request Handler.
Server Request Handler	It manages communication issues on the MEx Broker, e.g., accepting connections to receive requests from the MEx Clients.
Notifier Proxy	It implements the operation for the message service to notify subscribers used by the notification consumer.
Broker Engine	It implements the traditional operations of a messaging service, namely publish, subscribe, and unsubscribe.
Subscription Manager	It manages subscriptions and stores them in a database.
Notification Consumer	This component notifies subscribers when new messages arrive in the Broker.
MQTT Proxy	Similarly to the Queue Proxy, it acts as a proxy for the MQTT middleware, providing MQTT operations and transparently abstracting the interoperability between MEx and MQTT.
Pickle	This component (de)serializes messages using the Python Pickle library. It has the same role as the Marshaller.

Source: Author

It is essential to observe that developers can customize and build different adaptive middleware systems by reusing the components provided in the library. Constructing self-configuring middleware using the MQTT to implement IoT applications is possible. Thus, MEx incorporates a dedicated component, the *MQTT Proxy*, specifically designed for this purpose.

3.3.2 *p*ADL

In addition to its library of middleware components, MEx provides *p*ADL, a declarative architecture description language that allows developers to describe adaptive middleware software architectures. It establishes rules governing the composition of components, which determine and ensure the execution order of elements in the middleware architectures.

*p*ADL follows principles introduced by Medvidovic (Medvidovic; Taylor, 2000) and further extended by Rosa (Rosa et al., 2020) to have simple syntax and semantics. In practice, developers only need to specify the middleware architecture using the *p*ADL and then deploy it in the execution environment. For this purpose, a *p*ADL architecture has four explicit sections for specifying middleware architectures: components, attachments, adaptability, and configuration. The Backus-Naur Form (BNF) of *p*ADL can be found in Appendix A.

Section *Components* declares the components that make up the architecture. Each component has a name and an associated type. The architecture shown in Source Code 1 declares three components (Lines 2-6).

The *Attachments* defines the interactions between two components. Semantically, the execution of the components happens sequentially, based on the sequence of interactions defined in the attachments. For example, *queueproxy* interacts with *marshaller*, which in turn interacts with *crh* (Lines 7-10).

The *Adaptability* section defines the parameters of the adaptation strategy, namely the type and adaptation interval. In the example, the defined adaptation mechanism is *evolutive* (Line 12), aiming to keep the middleware updated. As mentioned in Section 3.4.5.1, the architecture is updated whenever a new version of any component used in middleware architecture is available. The *interval* (Line 13) defines an interval between checks of the need for adaptation. In this example, every 1200 seconds, the adaptation mechanism checks if a new version of the architecture components becomes available.

Source Code 1 – Python-based Architecture Description Language

```
 1 padl = {
 2     "components" = {
 3         "QueueProxy": "queueproxy",
 4         "Marshaller": "marshaller",
 5         "ClientRequestHandler": "crh"
 6     },
 7     "attachments" = {
 8         "queueproxy": 'marshaller',
 9         "marshaller": "crh"
10     },
11     "adaptability" = {
12         "type": "evolutive",
13         "interval": 1200
14     },
15     "configuration" = {
16         "starter" = "queueproxy",
17         "other_configs": {}
18     }
19 }
```

Source: Author

Finally, the *Configuration* section defines properties and rules governing architecture execution. For example, the *starter* field (Line 16) indicates the first component to be executed. This component is typically associated with the application running on top of MEx. Each component and application is an independent software module which executes sequentially several times without entering an infinite control loop. In practice, the execution unit starts the application and manages the sequential execution of each component based on the *p*ADL attachments. Therefore, it is essential to define which component initiates the execution, ensuring the proper order of operations.

3.4 EXECUTION TIME

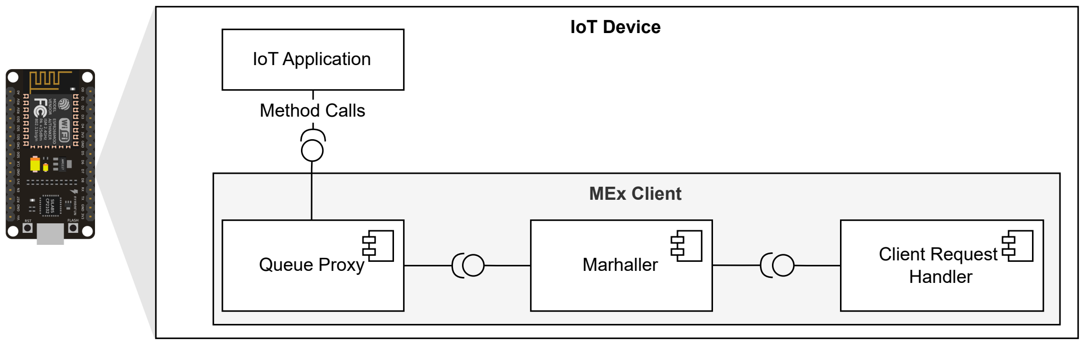
At execution time, MEx comprises five elements: MEx*Client*, MEx *Broker*, *Execution Unit*, *Managing System*, and *Knowledge Database*. These elements are responsible for executing the software architecture defined at development time and deployed in the execution environment.

3.4.1 MEx Client

MEx *Client* executes directly on the IoT devices. It isolates IoT applications from communication, distribution, and interoperability issues. Publishers use the MEx *Client* to publish messages without explicitly specifying recipients, selecting only the topics where the messages are published. Similarly, subscribers use the MEx *Client* to subscribe to some topic and receive messages from topics.

Figure 3.4 shows the application and the components that make up a standard architecture of the MEx *Client* based on the *p*ADL defined in Source Code 1. An *IoT application* is a software system that implements business logic to interact with IoT devices, enabling the collection, processing, and analysis of data from the environment.

Figure 3.4 – MEx Client



Source: Author

The *Queue Proxy* serves as an intermediary between the IoT application and the messaging service. It simplifies application development by providing a local interface for remote communication, abstracting the complexities of interacting with remote systems. The application only interacts with the *Queue Proxy* locally for remote invocations, making the process of remote communication simple and transparent.

Queue Proxy provides operations to publish, subscribe, check, and notify messages. These operations are used by applications to interact with the MEx *Broker*. Applications can publish messages without specifying recipients or knowing the intended recipients. They can also subscribe to one or more topics to receive messages or be notified when new messages arrive in the MEx Broker. These operations enable asynchronous and decoupled communication. The *Queue Proxy* exposes the same publish/subscribe interface to local IoT applications that the MEx *Broker* implements remotely, allowing them to invoke operations without concern for the underlying details of remote execution, which are managed by the MEx *Broker*.

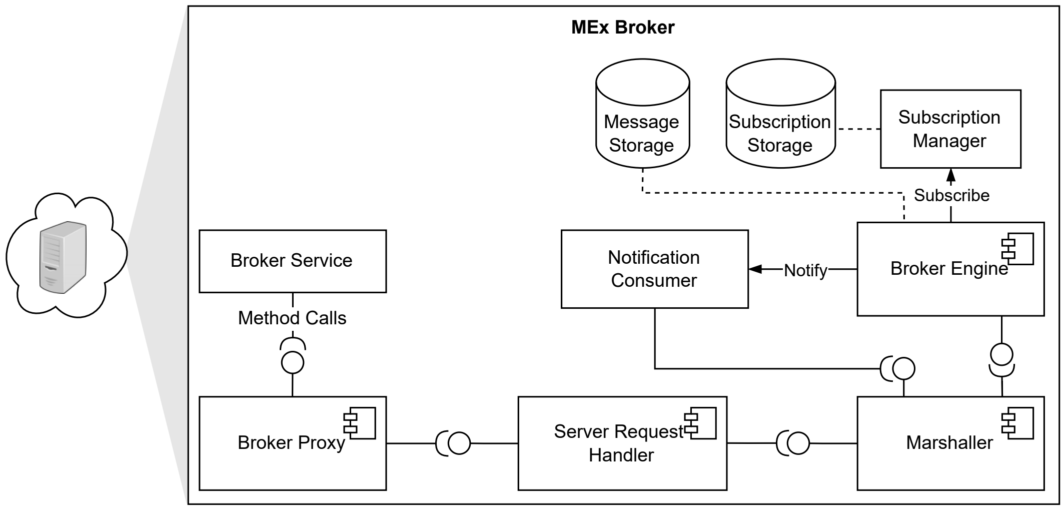
The *Marshaller* is the middleware component that serializes and deserializes messages, ensuring they are correctly converted into a format that can be transmitted over the network. Serialization involves converting the messages into a byte stream, while deserialization is the reverse process, transforming the byte stream back into messages. This capability is crucial for supporting interoperability in heterogeneous environments. The *Marshaller* ensures that applications do not need to manage low-level serialization logic, allowing them to focus on higher-level functionality.

Finally, the *Client Request Handler* deals with communication activities in the MEx. It is responsible for opening and closing network connections using the operating system's socket API, sending requests, and receiving responses to/from the network. In MEx, the *Client Request Handler* handles asynchronous invocations and manages timeouts and error detection.

3.4.2 MEx Broker

The MEx *Broker* is the messaging service of MEx. It runs in the cloud, stores messages published by IoT applications, manages topics and subscribers, and delivers/notifies messages to subscribers. Figure 3.5 shows the architecture of the MEx *Broker*, including components of a messaging service as proposed by Tarkoma (Tarkoma, 2012).

Figure 3.5 – MEx Broker.



Source: Author

The *Broker Service* starts the messaging service by invoking the *Broker Proxy*, which implements the operations responsible for initializing the service. Next, it calls the *Server Request*

Handler that manages the communication on the MEx *Broker* side and handles interactions with the *Client Request Handler*, such as accepting connections and receiving/sending data from/to the MEx *Client*.

The *Broker Engine* is the main component of MEx *Broker*. It manages the back-end processing of the messaging service, e.g., publish, subscribe, and unsubscribe. When messages are published, the *Broker Engine* receives and stores them in the *Message Storage*. Upon receiving subscription requests, the *Broker Engine* forwards them to the *Subscription Manager*, which stores the subscriptions in the *Subscription Storage*. This storage maintains an updated list of subscribers and the topics they are interested in, enabling it to identify which subscribers should be notified quickly.

The *Broker Engine* also sends stored messages to subscribers. It manages the relationship between topics in the *Message Storage* and the subscribers. When a notification is needed, the *Broker Engine* retrieves messages from the *Message Storage* and passes them to the *Notification Consumer*.

The *Notification Consumer* is an intermediary in the notification process. It receives messages from the *Broker Engine*, filters them to identify which subscribers are registered for each notification, and then notifies them through the notify operation.

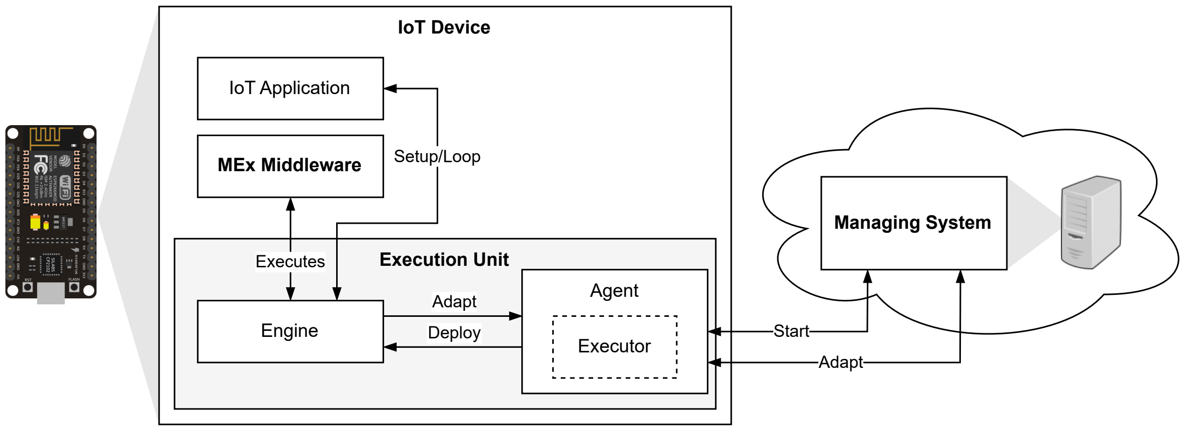
However, some applications work slightly differently. Instead of being notified directly, these applications may retrieve messages using a *checkMessage* operation. In this case, the *Broker Engine* delivers messages to subscribers through the *checkMessage* operation, referencing the topics in which the subscribers have expressed interest. This dual functionality accommodates push-based and pull-based message delivery models, providing flexibility depending on the application's needs.

3.4.3 Execution Unit

The *Execution Unit* loads, starts and manages the life-cycle of each component of the software architecture. It also executes the adaptation process. Figure 3.6 shows the architecture of the *Execution Unit*, which consists of two modules: the *Engine* and the *Agent*.

The *Agent* deploys the components of the middleware and performs adaptations locally on the IoT device, where it works as the executor within the MAPE-K adaptation loop. The *Agent* initially sends a *start* message to the *Managing System*. Next, it receives the middleware architecture, components, and configurations from the *Managing System*. Then, it passes these

Figure 3.6 – Execution Unit



Source: Author

elements to the *Engine*.

The *Engine* instantiates, loads, and executes the software architecture. Initially, the *Engine* processes the software architecture to obtain information about components, their interactions, and adaptation configurations. Next, the components are loaded, and the *Engine* starts an infinite loop that instantiates and executes all components, including the IoT application.

At the end of each execution cycle, the *Engine* checks the time elapsed since the last adaptation. When the elapsed time exceeds the adaptation interval, the *Engine* prompts the *Agent* to assess if an adaptation is necessary. The *Agent* gathers data about running components, applications, and the environment, depending on the adaptation mechanism in place. Then, it sends an adapt message to the *Managing System*, which handles the remote execution of adaptations.

Next, the *Agent* receives an adaptation plan along with any required elements (e.g., new components or configuration parameters) from the *Managing System* to adapt the MEx middleware or the application. At this stage, the *Agent* assumes the role of the *Executor* within the MAPE-K. In practice, executing the adaptation plan may involve removing existing components from the IoT device, loading new components, updating component records or adjusting application parameters. It is important to note that all these tasks occur at runtime without stopping the device or recompiling the software. This adaptation process is also transparent to the device and application, as it only occurs during application and middleware quiescence.

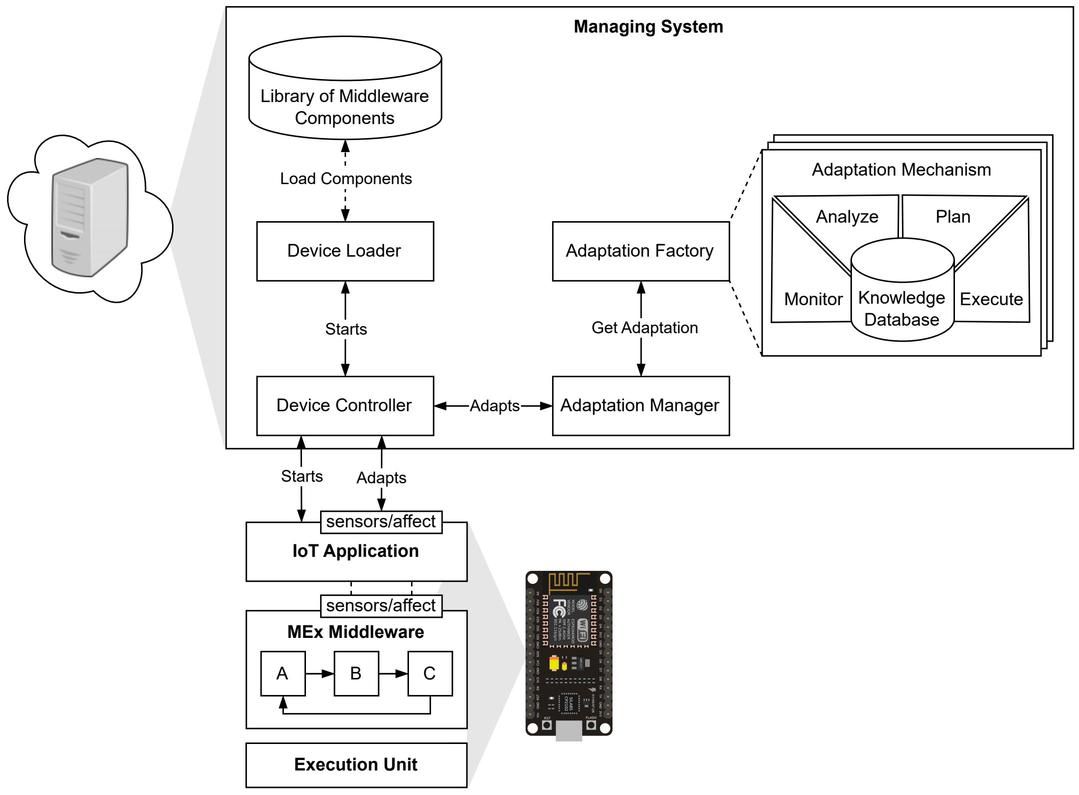
3.4.4 Managing System

The *Managing System* is responsible for the initialization and adaptation in the MEx. For the initialization, it performs a uniform, automatic, and transparent deployment of the IoT applications (Vögler et al., 2016) and the middleware. It is worth observing that manual deployment can be costly, especially for applications with several IoT devices (Razzaque et al., 2016; Vögler et al., 2016).

For the adaptation, the *Managing System* operates in a distributed manner, delegating some tasks to the *Agent* running on the device (see Section 3.4.3). It also implements the MAPE-K.

Figure 3.7 shows the main components of the *Managing System*. The *Device Controller* works as the interface for the *Managing System*, handling requests received from the MEx *Client* or MEx *Broker* to performing deployment or adaptation tasks. When an IoT device starts, the *Device Controller* receives a request from the device and forwards it to the *Device Loader*. The *Device Loader* automatically deploys the necessary middleware components on the device.

Figure 3.7 – Managing System



Source: Author

For adaptations, the *Managing System* triggers and executes them. It works as a hub of adaptation mechanisms, extensible and customizable, enabling quick adjustments to these mechanisms according to the needs of the IoT environment.

MEx's adaptive mechanisms support dynamic adjustments at both application and middleware levels. At the application level, adaptations reconfigure execution parameters without stopping or recompiling the application, e.g., modifying sensing frequencies to better respond to environmental changes. At the middleware level, adaptations allow fixing bugs, adding new functionality, or updating it.

MEx supports reactive adaptations, i.e., adaptations occur in reaction to a given event. MEx implements a decentralized MAPE-K in which the *Monitor*, *Analyzer* and *Planners* executes on the cloud and the *Executor* runs on the IoT devices. The *Monitor* may operate on both devices and the cloud, collecting data for the adaptation process, such as battery level and new component versions. The *Analyzer* analyses monitored data and decides whether an adaptation is necessary. The *Planner* builds an adaptation plan if an adaptation is needed. The *Executor* runs directly on the devices. Finally, the *Knowledge Base* stores information about things and applications, including their IDs, current component versions, and adaptation types and parameters.

For adaptations, when the *Managing System* receives an adaptation request, the *Device Controller* forwards it to the *Adaptation Manager*, who orchestrates the whole adaptation process. The *Adaptation Manager* interprets this request by including the type of adaptation required and the required adaptation elements, such as components to be replaced and parameter values. Then, it forwards this information to the *Adaptation Factory* responsible for creating instances of the requested adaptation mechanisms. Next, these adaptive mechanisms trigger and execute the adaptation process.

The decentralization of the MAPE-K is essential in IoT deployment. It helps to save devices' computational and energy resources by offloading adaptation processing. Moreover, the adaptation logic in MEx differs slightly from traditional MAPE-K implementations. The *Executor*, which typically has a passive role, is active in MEx.

3.4.5 Adaptation Mechanisms

The adaptation mechanisms in MEx address the uncertainties inherent to IoT applications and environments. They implement strategies based on specific goals, using computational resources such as rule engines and upload and download operations. MEx implements two adaptation mechanisms: *Evolutive Mechanism*, and DCAM.

3.4.5.1 *Composite Adaptation*

In the composite adaptation, the software behavior is altered by adding, removing, replacing or reconnecting components (McKinley et al., 2004). The composite adaptation in MEx is implemented through an *Evolutive Mechanism*.

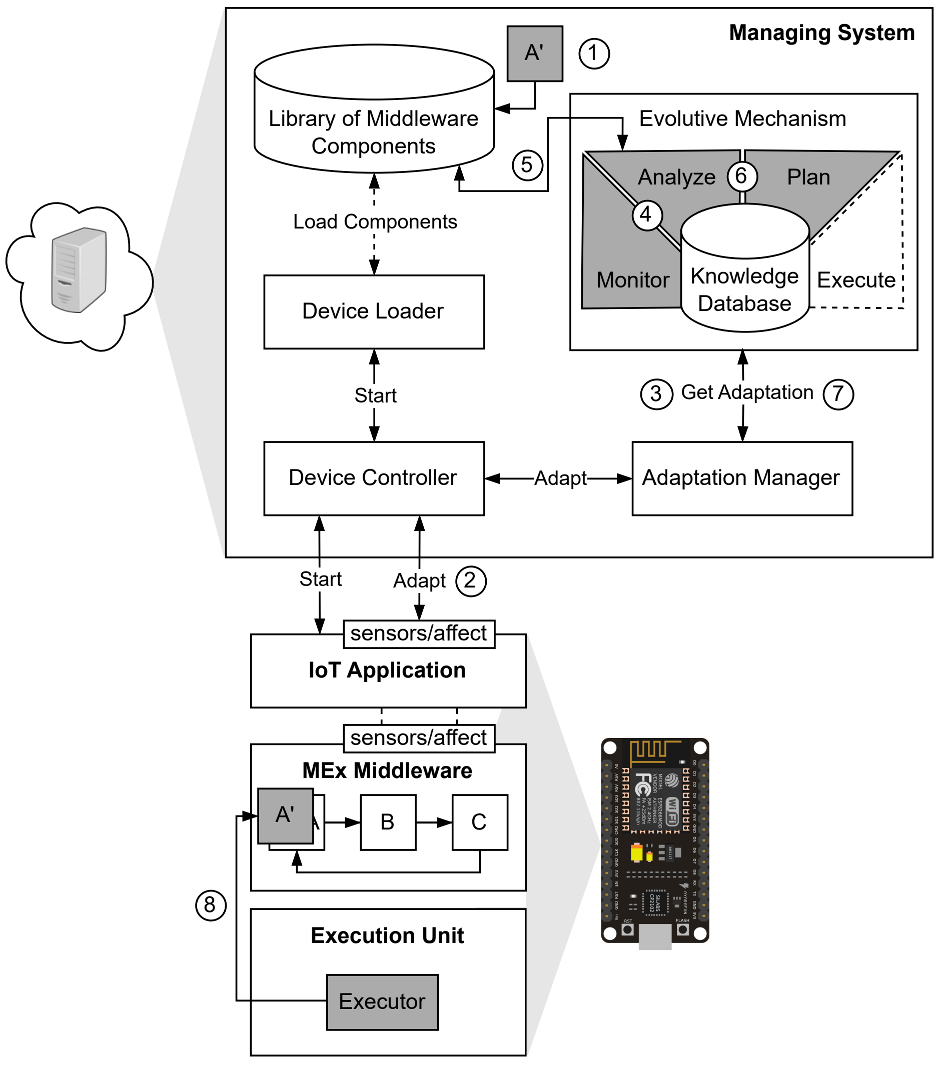
The *Evolutive Mechanism* (Cavalcanti; Carvalho; Rosa, 2021) continuously updates the middleware to incorporate new features, fix bugs, or enhance performance. To achieve this, it implements a strategy that replaces middleware components running on the device whenever a new component becomes available in the component library. Figure 3.8 shows the steps of *Evolutive Mechanism* in action.

When a new version of a given component (e.g., A') appears in the library (1), this new version is perceived by the *Monitor*. Next, when the *Device Controller* receives an adaptation request from the IoT device (2), it forwards the request to the *Evolutive Mechanism*, which dispatches this information to the appropriate *Monitor* (3). The *Analyzer* then receives the information of the request. The *Monitor* continuously monitors and gathers data about the current versions of the specified components installed on the device from the *Knowledge Database* and forwards this information to the *Analyser* (4). The *Analyzer* then compares these component versions with those available in the library (5) and decides the necessity for adaptation. This decision verifies if a newer version of a specific middleware component is available. Upon detecting a new version, the *Analyzer* prompts the *Planner* (6) to build an adaptation plan for replacing the component's versions running on the device (the old version) with the newly available one. Next, the new component is transmitted to the *Executor* operating on the device (7). Finally, the *Executor* executes the adaptation by reloading the new components and discarding the old ones (8).

3.4.5.2 *Parametric Adaptation*

Parametric adaptations alter parameters to adjust the application's behavior at runtime. Generally, IoT applications consist of IoT devices performing sensing and actuation tasks coupled with software modules responsible for data processing and visualization. Dynamic configurations are necessary given the inherent uncertainties in IoT environments. For example,

Figure 3.8 – Steps of Evolutive Adaptation



Source: Author

changing communication parameters (e.g., thing mobility, dynamic broker provisioning, and broker load balancing) without stopping the IoT device operation helps to solve communication uncertainties.

DCAM (*Duty Cycle Adaptive Mechanism*) is a parametric adaptive mechanism to save battery of IoT devices. DCAM automatically adjusts the device operating mode to shorten or extend its operation according to the application context, which adjusts the power consumption. It uses the *duty cycle* to save energy and maximize the battery lifetime(Abdul-Qawy; Almurisi; Tadisetty, 2020). The *duty cycle* allows devices periodically to switch between active (e.g., sampling and radio transmission) and non-active (e.g., sleeping) periods in the operating mode to use minimal energy. *Duty Cycle* is defined as shown in Equation 3.1.

$$Duty\ Cycle = \frac{Tw}{T} 100\%, \tag{3.1}$$

where *Tw* is the active application time, and *T* is the total operation time. For example, a *duty cycle* of 30% means that the application works for 30% and sleeps for 70% of the time.

Although periods of activity/sleeping always happen when the *duty cycle* strategy is adopted, there is no typical pattern in how they are appropriately defined for IoT applications. Hence, it is possible to simplify this issue by dividing these periods into three stages: *sampling*, *transmission*, and *sleeping* (Abdul-Qawy; Almurisi; Tadisetty, 2020). The device is active in the sampling and transmission stages, and its components execute tasks and consume energy. In the sleeping stage, less energy is consumed because the device suspends the energy supply of most components, does not realize any processing task, and the communication is switched off. Therefore, it is easy to notice that power consumption is more significant in the first two stages than in the sleeping stage.

Those stages are usually statically configured regarding the *duty cycle*. After deployment, it cannot be updated without stopping the application. In this sense, the device can lead to application unreliability/uncertainties because it may be sleeping when it should be running or consuming energy when it should be sleeping. For this reason, the *duty cycle* adaptive mechanism adjusts the *duty cycle* ratio based on the application's sleep time.

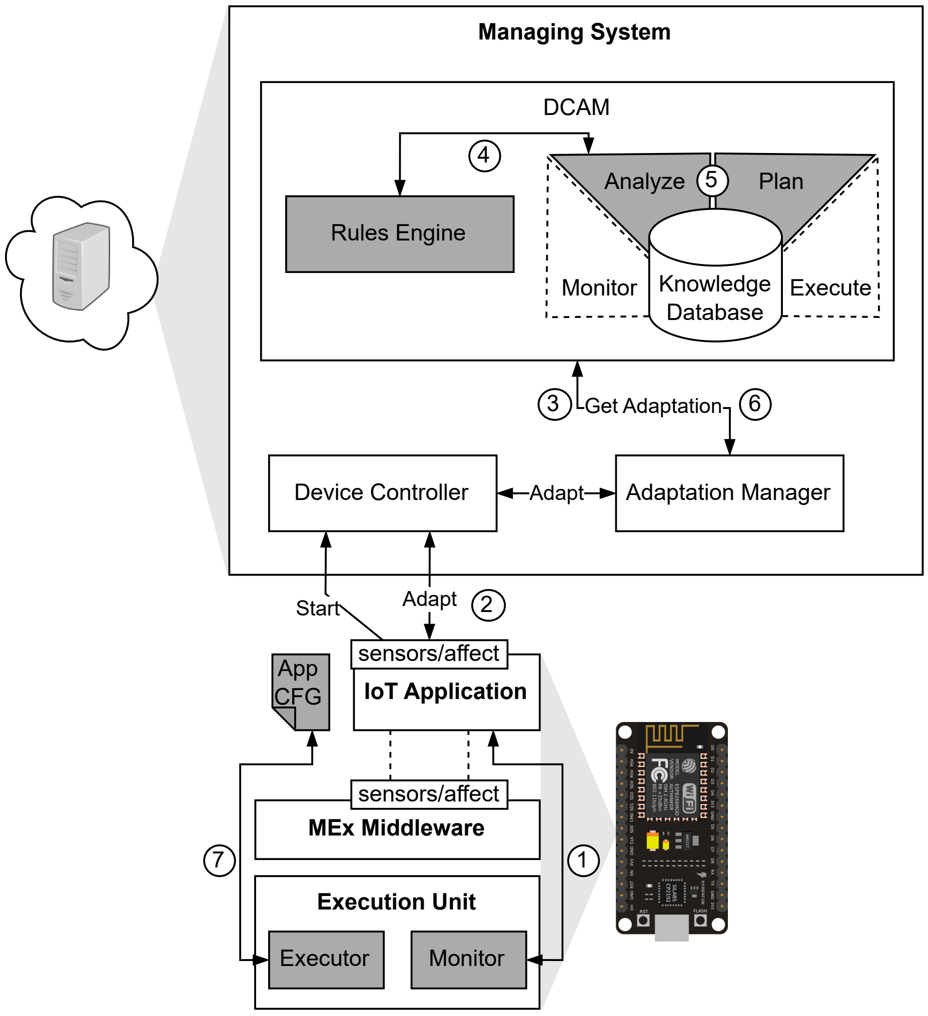
DCAM is customizable and employs a declarative, rule-based approach. It uses an engine[1] that allows the creation of conditional business rules simply and flexibly, with an easily understandable and configurable syntax. The engine evaluates a set of rules, where each rule consists of conditions (predicates) and actions to be triggered when those conditions are met. When provided with data (referred to as *facts*), the engine checks whether the conditions are true, and if so, executes the corresponding actions. The engine also supports the creation of complex rules with boolean logic, comparison operators, and custom functions. This approach makes it easier to implement dynamic business logic in systems that need to respond to changing situations based on varying data.

Figure 3.9 shows a general overview of the *parametric adaptation*.

The adaptation process starts with the *Monitor* receiving parameters for adaptation decisions (1). These parameters are stored in the device configuration, generated in the application context or environment, or are directly generated by the device. For example, the *Monitor* run-

[1] https://www.npmjs.com/package/json-rules-engine

Figure 3.9 – Steps of the DCAM Adaptation



Source: Author

ning on the device may collect data such as temperature, humidity, or water levels from the device's sensors. It may also gather the device's battery level, a parameter generated internally by the device. When the *Device Controller* receives an adaptation request from the IoT device (2), it forwards the request to the DCAM, which dispatches this information to the *Analyzer* (3).

The *Analyzer* determines whether an adaptation is necessary based on the defined adaptation goals. It uses a goal declarative based on the rule-based approach. A rule-based system is an expert system that uses rules to store and manipulate knowledge to interpret information and make decisions. The *Analyzer* passes the adaptation parameters to the rules engine[2], which processes these parameters according to defined rules to determine the need for adaptation (4). If adaptation is necessary, the *Analyzer* asks the *Planner* (5) to build an adaptation plan

2 https://www.npmjs.com/package/json-rules-engine

to adjust the application's parameters. The updated configuration is then transmitted to the *Executor* operating on the device (6).

Finally, the *Executor* executes the adaptation by replacing the old configuration with a new one (7). It is worth observing that the adaptation process occurs without stopping the application or the device entirely.

3.5 IMPLEMENTATION

MEx[3] modules were implemented in MicroPython[4], JavaScript, and C. Components running on IoT devices are implemented in MicroPython or C, while cloud components utilize JavaScript.

To have MEx running on any device, it is initially necessary to manually deploy the *Engine*, *Agent*, and application on the device. The *Engine* executes the middleware's software architecture defined in the *p*ADL, while the *Agent* is responsible for the automatic deployment of other required elements and for adaptation actions.

When the device starts, the *Agent* automatically deploys the architecture, configuration file, and middleware components. Next, it passes them to the *Engine* for instantiation, initialization, and execution.

In addition, MEx was implemented using concepts of object-oriented programming. Each middleware component is a Python class with a particular generic method for communicating among the middleware components within the *Engine*. The *Engine* interprets the middleware architecture and a configuration file to execute the middleware. It receives the data from the attached component as arguments, according to *p*ADL attachments (see Section 3.3.2).

Once MEx is primarily based on Python, the language's resources enrich the *p*ADL. In this sense, the identifier corresponds to the Python file name encapsulating the component's functionality, while the type represents the component class. Also, the entire *p*ADL is built as a Python collection structure. This Python-oriented approach adds dynamism to the *p*ADL in the MEx.

Similar to *p*ADL, the *configuration file* is Python-based and structured as a dictionary containing three main settings sections: application, device, and environment, as shown in Source Code 2. Each section has specific settings with different parameters to define how the

[3] <https://github.com/davidjmc/middleware-extendify>
[4] <https://micropython.org/>

IoT application and MEx should behave. These configurations allow MEx to customize and adapt the software's functionality without changing the source code.

Source Code 2 – Configuration File of the MEx

```
 1  configurations = {
 2      "application" = {
 3          "loop_interval": 30,
 4          "publish_in": "1b:e2:51:97:31:42",
 5          "subscribe_to": None
 6      },
 7      "device" = {
 8          "id": "1b:e2:51:97:31:42",
 9          "location": None
10      },
11      "environment" = {
12          "broker_host": "172.22.64.223",
13          "broker_port": 60000,
14          "await_broker_response": 1
15      }
16  }
```

Source: Author

The *application* section (Lines 2-6) defines settings for the IoT application. For example, loop_interval (Line 3) sets the interval in seconds for executing a loop or cycle of operations of the applications. The publish_in (Line 4) identifies the topic to which the application should send (publish) data to the MEx *Broker*. The subscribe_to (Line 5) identifies the topics to which the application should subscribe to receive data. This example is set to *None*, indicating that the application is a publisher. Otherwise, it would be a subscriber. If both *publish_in* and *subscribe_to* were configured, the application would be both a publisher and subscriber.

The *device* section (Lines 7-10) contains settings about the IoT device. For example, id (Line 8) represents the device's unique identifier, usually using the device's MAC address as the identifier. The location parameter (Line 9) indicates the device's location. Although this parameter is currently not used, it shows that various parameters can be included and customized to the application's context.

The *environment* section (Lines 11-15) specifies settings for the environment. For example, *broker_host* and *broker_port* (Lines 12-13) are the IP address or hostname of the Broker and the network port used by the Broker, respectively. These parameters are used for communication. The *await_broker_response* parameter (Line 14) determines if the system should wait for a response from the Broker. It is set to 1, indicating it should wait. This setting is set to 0 when the client is both publishing and subscribing, meaning it should not wait for a server response after publishing to stay ready to receive messages.

It is essential to highlight that this configuration file is used for the parametric adaptation (see Section 3.4.5.2). It becomes possible to change the software's behavior without modifying the code. This action includes changing loop intervals, configuring devices for publishing and subscribing, defining device details, and specifying broker settings. By changing this file at runtime, MEx can adjust the software's settings for different environments and requirements, maintaining flexibility and ease of maintenance.

The *Engine* is also responsible for executing the IoT application. In practice, applications run several times without a continuous control loop, as it is part of the main loop provided by the *Engine*. The steps of the engine are summarized in Source Code 3.

Initially, the *Engine* extracts information from the software architecture (*p*ADL), such as middleware components (Line 2), attachments (Line 3), adaptation settings like the type of adaptation and the adaptation interval (Line 4), and MEx configurations, e.g., the component that starts the architecture execution (Lines 5). It also reads a separate configuration file specific to the application, environment, and device, which contains details such as the Broker's IP and adaptation parameters (Line 6).

Next, it dynamically loads the component instances (Lines 11-13) and the application setup (Line 17). Once everything is loaded, the *Engine* enters an infinite loop, running the application (Line 22-40) and middleware components.

Source Code 3 – Engine of the MEx

```
1  class Engine:
2      self.components = adl.adl['components']
3      self.attachments = adl.adl['attachments']
4      self.adaptability = adl.adl['adaptability']
5      self.configuration = adl.adl['configuration']
6      self.configurations = configurations
7  
8      def __init__(self):
9  
10         # load components
11         for component in self.components:
12             component_file = self.components.get(component)
13             current[component] = __import__('components.' + component_file)
14 
15     def run(self, app):
16         self.app = app
17           self.app.setup()
18 
19         if self.last_adaptation == 0:
20             self.last_adaptation = time.time()
21 
22         while True:
23             try:
24                 # application execution loop
25                 self.app.loop()
26 
27                 # components execution starting with the starter
28                  for component in self.configuration['starter']:
29                     component_instance = self.current[component]
30                     component_instance.run()
31 
32                 if (self.adaptability['type'] not in ['', None]
33                 and (time.time() - self.last_adaptation) >
34                 self.adaptability['interval']):
35                     # it will adapt
36                     updated = self.agent.adapt()
37                     if updated:
38                         self.reload_components()
39                         self.configurations = configurations
40                     self.last_adaptation = time.time()
```

Source: Author

At the end of each execution cycle, if the adaptation interval is exceeded (Lines 32-34), the *Engine* invokes an adaptation mechanism using the *Engine* (Line 39). This mechanism runs remotely, monitoring the system and making decisions about the need for adaptation. If adaptation is required, it is executed (Lines 37-39).

Another critical component is the *Agent*, responsible for deployment and adaptations on the IoT device and whose partial implementation is shown in Source Code 4.

Source Code 4 – Agent of the MEx

```
1  class Agent:
2      conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3      conn.connect(socket.getaddrinfo(managing_sytem, port)
4  
5      def start():
6          msg = b'START\nThing:' + bytes(THING_ID)
7          data = self.send_receive(conn, msg)
8          data = str(data, 'ascii')
9          try:
10             [headers, data] = data.split('\x1e')
11         except:
12             pass
13         AmotAgent.update_files(data)
14 
15     def adapt():
16         msg = b'ADAPT\nThing:' + bytes(THING_ID)
17         for info in self.app_context:
18             msg += b'\n' + bytes(info, 'ascii') + b':' + self.app_context[info]
19         for info in self.env_context:
20             msg += b'\n' + bytes(info, 'ascii') + b':' + self.env_context[info]
21         data = self.send_receive(conn, msg)
22         data = str(data, 'ascii')
23         if data == None:
24             return
25 
26         if len(data) > 0:
27             self.update_files(data, False)
28             print('Adapted')
29             return True
30         return False
```

Source: Author

The *Agent* is directly triggered by the *Engine* and implements two functions, *start()* and *adapt()*. When the device starts, the *Engine* calls *Agent.start()*, which connects to the *Managing System* (Lines 2-3) and initiates the startup sequence (Lines 5-13). It sends a *START* request along with the device ID (Lines 6-7), receives the middleware software architecture and components (Line 8), and deploys them (Line 13).

When the adaptation interval is exceeded, the *Engine* calls *Agent.adapt()*. In this case, the *Agent* acts as an *Executor* and can also work as a *Monitor*. It collects data from the application and environment, such as component versions or sensor readings (e.g., water and battery levels) necessary for the adaptation process. The *Agent* then aggregates this information into an adaptation message (Lines 16-20) and sends an *ADAPT* request (Line 21). If adaptation is required, the *Agent* executes the adaptation plan on the device (Lines 26-29).

The *Managing System* and *Adaptation Mechanism* were implemented using JavaScript within Node.js[5]. The *Managing System* plays a crucial role in executing adaptation processes, applying the appropriate mechanisms for each situation. To achieve this, it includes an *Adaptation Manager* module, which orchestrates the entire adaptation process. A partial imple-

[5] <https://nodejs.org/en/about>

mentation of the *Adaptation Manager* is presented in Source Code 5.

Source Code 5 – Adaptation Manager

```
1  export class AdaptationManager {
2      constructor(thing) {
3          this.thing = thing
4          this.adapters = this.loadAdapters()
5      }
6
7      loadAdapters() {
8          let adapters = this.thing.adaptability.type.map(type => {
9              return AdapterFactory.for(this.thing, type)
10         }).filter(adapter => !!adapter)
11         // return a list of Adapter objects
12         return adapters
13     }
14
15     with(request) {
16         if (!this.adapters.length) {
17             return null
18         }
19         let adapter = this.adapters.pop()
20         let adaptation = adapter.adaptFor(request)
21         this.adapters.map(adapter => {
22             adaptation.merge(adapter.adaptFor(request))
23         })
24         return adaptation
25     }
26 }
```

Source: Author

This code includes functions to load existing adaptation mechanisms (Lines 7-13) and map and return the appropriate mechanism for the requesting device (Lines 15-25), respectively.

When an adaptation request is received, the *Managing System* directs it to the *Adaptation Manager*, which processes the request by gathering device information (Line 3) and loading the necessary adaptation mechanisms directly within its *constructor* (Lines 2-5). Next, these mechanisms are passed to the *Adaptation Factory* (Lines 6-7), which instantiates and returns the requested adaptation mechanism. The *Adaptation Factory* acts as the central hub for adaptation mechanisms, maintaining a list of available mechanisms, as shown in Source Code 6 (Lines 5-19).

The *Adaptation Manager* receives the desired adaptation mechanism. Finally, the mechanism used triggers and executes adaptations.

Source Code 6 – Adaptation Factory

```
import EvolutiveAdapter from './EvolutiveAdapter.js';
import ParametricAdapter from './ParametricAdapter.js';
import TDCAM from './TDCAM.js';

export default class AdapterFactory {
    static for(thing, type) {
        if (type == 'evolutive') {
            return new EvolutiveAdapter(thing)
        }
        if (type == 'dcam') {
            return new ParametricAdapter(thing)
        }
        if (type == 'tdcam') {
            return new TDCAM(thing)
        }
        //...

        return null
    }
}
```

Source: Author

Finally, as much as possible, the MEx implementation uses low-cost, low-CPU, and low-power-consuming operations. For example, avoid using loops and list comprehensions and use local variables to optimize the code whenever possible. In addition, using MicroPython is crucial because it is compact enough to fit and run within just 256k of code space and 16k of RAM. It was also necessary to avoid external libraries and focus on adopting native MicroPython objects (built-ins) to optimize the code with faster execution operations.

3.6 CONCLUDING REMARKS

This chapter presented MEx. Initially, an overview of MEx was introduced. Next, the design decisions of MEx were presented, followed by its development time, where the framework and its elements were outlined. Then, the execution environment of MEx was described, including its adaptation mechanisms. Finally, implementation aspects were discussed.

4 MEX BASED SOLUTION FOR WATER MONITORING

"If we had computers that knew everything there was to know about things—using data they gathered without any help from us—we would be able to track and count everything and greatly reduce waste, loss and cost. We would know when things needed replacing, repairing or recalling and whether they were fresh or past their best. The Internet of Things can change the world, just as the Internet did. Maybe even more so."

—Kevin Ashton

This chapter presents AquaMOM, an adaptive IoT system for water monitoring in challenging environments, which utilizes the capabilities of MEx. This chapter begins by introducing the context and motivation behind the AquaMOM. Next, it presents an overview of the system and details the critical modules of AquaMOM. Finally, it concludes with aspects of AquaMOM's implementation.

4.1 CONTEXT AND MOTIVATION

A primary goal of IoT is to distribute functionality to physical and virtual devices, improving system infrastructure and efficiency. In this context, the World Economic Forum recognizes global water scarcity as a significant risk (Forum, 2023; Forum, 2018), emphasizing the urgent need for efficient water management solutions.

The growing concerns over water scarcity are driven by rapid population growth and increasing water demand. According to theUnited Nations (UN), water consumption is expected to outstrip supply by 40% by 2030[1]. This water crisis is complex, involving ecological, economic, and social factors. Urban growth and agricultural expansion have increased competition for water resources, leading to conflicts between sectors and communities. Additionally, climate change is intensifying this issue by altering rainfall patterns and increasing the frequency and severity of droughts and floods[2].

[1] <https://www.theguardian.com/environment/2023/mar/17/global-fresh-water-demand-outstrip-supply-by-2030>
[2] <https://www.un.org/pt/climatechange/science/causes-effects-climate-change>

Searching for effective water management and conservation solutions is crucial in this context. These solutions include monitoring and alert systems, public conservation policies, and wastewater reuse to ensure the sustainability of water resources (Getirana; Libonati; Cataldi, 2021; Brito; Lopes; Neta, 2019). An alternative to face this challenge is developing smart water management solutions, which can measure water consumption, detect leaks, and predict waste (Han; Mehrotra; Venkatasubramanian, 2019). Smart water solutions, a subdomain within smart city initiatives, utilize IoT technology to enhance water infrastructure, making urban environments more sustainable and efficient. These solutions optimize water monitoring, sourcing, treatment, and delivery. The evolution of IoT has spurred the development of many such systems, which lead with sourcing, treatment, and delivery (Singh; Ahmed, 2021; Han; Mehrotra; Venkatasubramanian, 2019). For example, Liu and Mukheibir (2018) shows that simply implementing an automatic feedback system for customers can reduce water consumption by 4.2% to 8.5% without additional interventions.

In Brazil, particularly in the Northeast region, water scarcity is intensified by rainfall variability and extensive semi-arid areas. These facts make water conservation critical, especially in remote, economically disadvantaged regions with limited resources (Gondim et al., 2017). Water cisterns are widely used to collect rainwater for dry periods, but optimizing water usage demands advanced solutions. Meanwhile, the critical need for rational water use and management can be achieved through IoT systems, which can improve water infrastructure and conservation.

Implementing IoT systems in semi-arid regions is challenging due to harsh conditions that introduce uncertainties in sensor accuracy, battery life, and maintenance. Addressing these uncertainties is essential for effective system operation. IoT systems require a robust design to face dynamic conditions, including aging infrastructure, limited capacity, and climate changes that affect sensor reliability, power management, and system durability.

This thesis also proposes AquaMOM, an adaptive IoT system designed for monitoring water consumption and built on the MEx middleware. The choice of a smart water management application is driven by the urgent need for water conservation strategies in the Northeast region of Brazil, where environmental and socioeconomic factors prioritize efficient water use. Beyond its technical contributions, AquaMOM also addresses a critical social challenge by promoting more sustainable water management in resource-constrained environments.

AquaMOM leverages MEx's capabilities to manage uncertainties, respond to dynamic changes and meet application demands. The MOM architecture of MEx plays a key role in handling dynamic network conditions, supporting adaptive mechanisms, and ensuring reliable message exchange, even in challenging environments. By integrating this approach, AquaMOM shows how an IoT system built atop a MEx middleware can manage uncertainties and adapt to real-world deployments.

4.1.1 Semi-arid Regions of Brazil

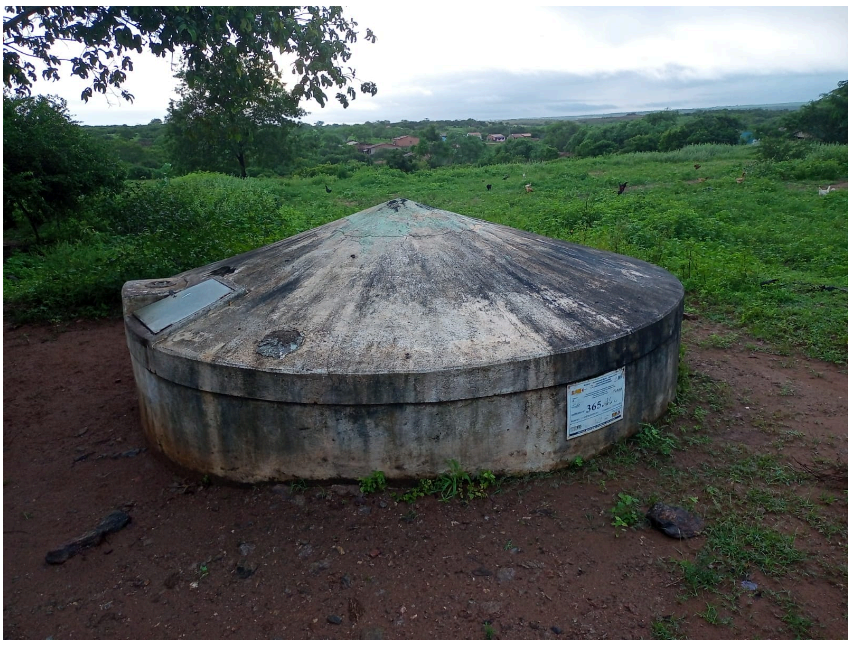
Brazil's semi-arid region covers approximately 1,322,680 km^2, which is 15% of the national territory, and comprises 1,477 municipalities across eleven states. This region faces critical water scarcity, affecting around 53.1 million people, many of whom live in poverty (Barbosa, 2024). Irregular rainfall and prolonged droughts exacerbate water shortages, resulting in insufficient water supplies that directly affect quality of life, health, and community resilience. These facts have provoked a pressing need for effective water conservation and management solutions (Rodriguez; Pruski; Singh, 2016).

Historically, the Brazilian government has employed various policies to mitigate the effects of drought (Rodriguez; Pruski; Singh, 2016), like building cisterns for rainwater storage to support families during dry periods. Each cistern (see Figure 4.1) is 1.8 meters tall and 3.4 meters in diameter, holding 16 m^3 of water—sufficient to sustain a family of four for up to five months at a rate of 25 liters per person per day (Rodriguez; Pruski; Singh, 2016), aligning with the World Health Organization's recommended minimum daily water requirement (França et al., 2010).

Despite these government efforts, water shortages persist, signaling the need for improved solutions. Implementing new actions and measures to ensure population access to water is necessary. For example, there is a critical need for rational water use and better customer management. In this context, technological solutions, such as smart water systems, which incorporate IoT devices (e.g., sensors and actuators), can play a pivotal role in enhancing water infrastructure and conservation efforts (Singh; Ahmed, 2021; Han; Mehrotra; Venkatasubramanian, 2019).

The implementation of IoT solutions in these regions presents significant challenges. Harsh environmental conditions, such as high temperatures and low humidity, can reduce the lifespan of sensors, batteries, and other electronic components and affect measurement accuracy and system durability. The distance from urban centers makes maintenance costly and complex, and aging infrastructure and unreliable Internet and mobile networks further complicate data transmission and remote monitoring. These factors demand the development of robust and

Figure 4.1 – Cistern of Water in Semi-arid Region



Source: Author

resilient IoT solutions that can withstand harsh climates and operate reliably, even in the face of connectivity and maintenance challenges.

4.1.2 IoT Systems for Challenging Environments

IoT systems have gained widespread use in diverse domains, including smart cities, healthcare, and industrial applications, each with specific requirements. These systems have recently been deployed in challenging environments, such as extreme areas, such as chemical facilities or nuclear plants, and emergency management systems for disaster response scenarios (Kant; Jolfaei; Moessner, 2024).

However, such IoT systems must be specifically designed to operate in these harsh environments. They must be robust and capable of functioning under extreme conditions. Remote and underserved areas, such as small towns far from urban centers or semi-arid regions, can present unexpected challenges. They are due to aging infrastructure, demand exceeding capacity, resource limitations, and increasingly harsh operating conditions caused by climate change, e.g., dry heat climate (Kant; Jolfaei; Moessner, 2024).

Designing and implementing IoT systems for such environments is difficult. The main challenge is to assess potential damage with minimal manual effort and adjust or improve the system before, during, and after an event according to its needs. This approach aims to maintain service quality, maximize coverage, and optimize the resources offered (Kant; Jolfaei; Moessner, 2024).

4.2 AQUAMOM

AquaMOM is a holistic and adaptive IoT system designed specifically for monitoring water consumption. It provides customers with digital feedback and assists them in improving water conservation efforts in demanding settings. It informs consumers about volume, usage control, and water depletion forecasts (Cavalcanti et al., 2024).

While the AquaMOM can be customised for whatever environment, it is primarily tailored for water cisterns in Brazil's semi-arid regions, as shown in Figure 4.1. AquaMOM combines a distributed software stack and a low-cost IoT device. The stack includes a Web application for digital feedback on water usage and an IoT application running on IoT devices to collect water data. Both applications are built on a middleware developed with MEx.

AquaMOM instantiates the components of the MEx middleware and validate its concepts. During the development of AquaMOM, a dedicated IoT device was required to collect and transmit water consumption data.

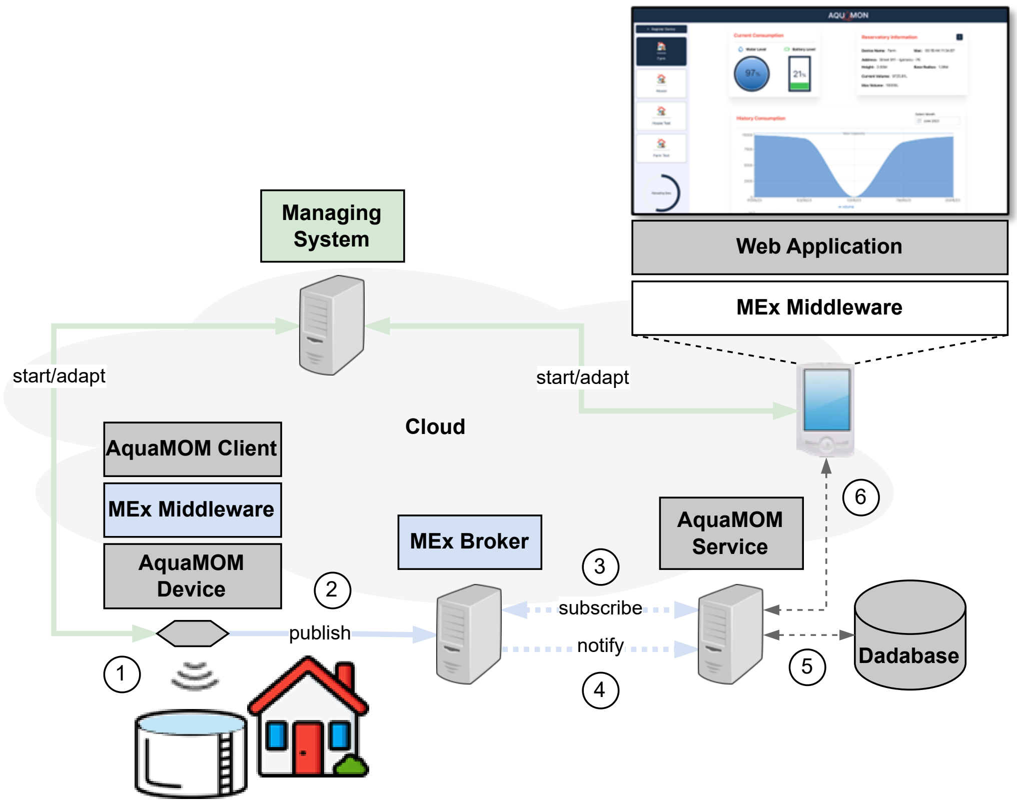
The device is equipped with sensors to measure water usage. The choice of micro-controllers and sensors was driven by the need for an affordable solution that supports MEx's requirements and ensures low-cost production. This included selecting a micro-controller that supports a programming language capable of handling socket implementations and off-the-shelf components to ensure functionality and cost-effectiveness.

The essential difference of AquaMOM from existing solutions is the use of adaptive concepts, which ensure proper functioning in challenging environments while minimizing deployment and maintenance costs. Figure 4.2 presents a general overview of AquaMOM, including the AquaMOM Client, AquaMOM Service, AquaMOM device, and the MEx middlware distributed over IoT devices and the cloud.

The AquaMOM Client is an application running on the AquaMOM device, whose primary function is periodically measuring the water level in a home cistern using a sensor (1). After collecting data, the AquaMOM Client publishes it to a specific topic associated with the device in the MEx Broker in the cloud through MEx middleware (2).

In the cloud, the AquaMOM Service, subscribed to this topic on the MEx Broker, listens for the sensed data (3). When new data is published, the MEx Broker, functioning as

Figure 4.2 – Overview of AQUAMOM



Source: Author

a messaging service, notifies the AQUAMOM Service (4). The AQUAMOM Service processes the received data based on predefined business rules, stores it in a relational database (5), and makes it available to a Web application (6). This web application displays digital feedback on water usage, helping to engage customers and promote behavior change through dynamic water consumption updates.

In Figure 4.2, the green arrows represent communication between the AQUAMOM Client (on the device) and the Managing System of MEX. These arrows depict messages related to the automatic deployment of the application and its middleware components (start message) and adaptation messages (adapt message) that trigger changes in the middleware or the application itself.

The blue arrows represent the middleware operation messages, specifically the publish, subscribe, and notify operations the application uses to communicate with the MEX Broker.

Finally, the grey arrows illustrate the internal communication between the AQUAMOM Service, the database, where the data is stored, and the dashboard, where the water information is displayed. This communication is essential for providing users with feedback on water consumption.

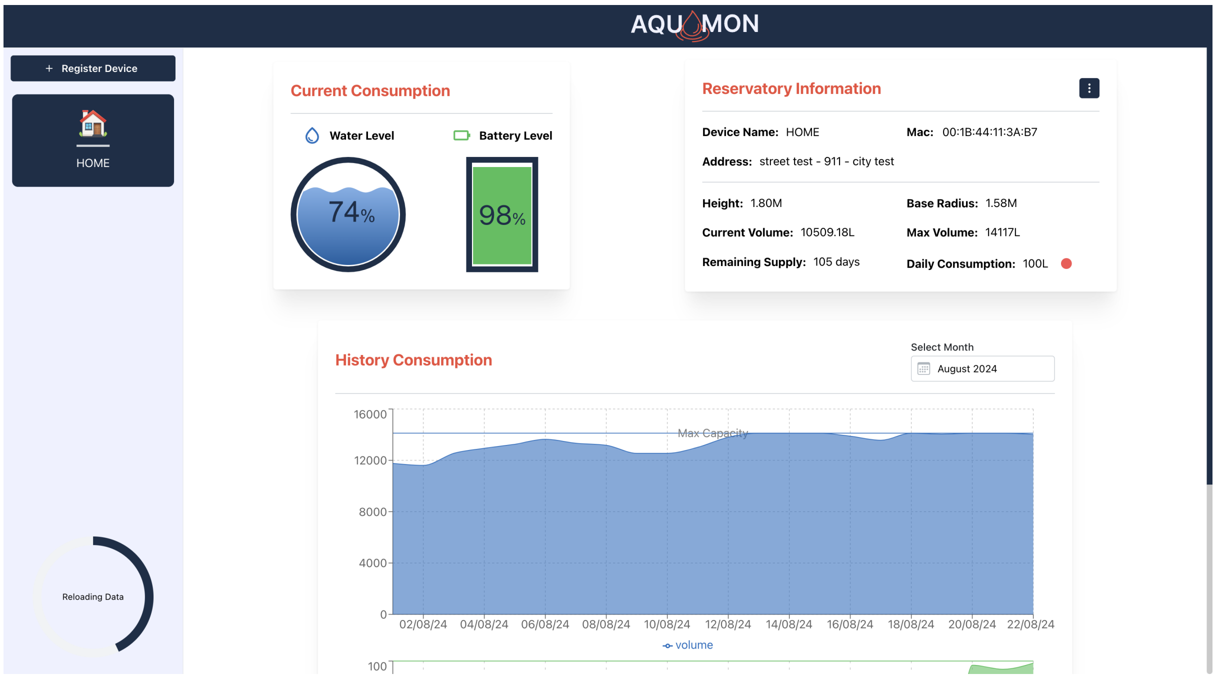
The AquaMOM uses MEx concepts for managing communication, data distribution and execution. Additionally, these concepts allow AquaMOM to adapt dynamically. AquaMOM uses parametric adaptation (see Section 4.2.3) to save energy by (re)configuring measurement frequency of sensed data; close a smart water valve in the house may also be necessary. More details on the adaptation process are provided in Sections 3.4.4, 3.4.5 and 5.

4.2.1 AquaMOM Service

The AquaMOM Service is a Web application responsible for processing data received from each device. This MEx middleware abstracts the subscription to a topic in the MEx Broker (see Section 3.5) and receives the sensed data from AquaMOM devices. The AquaMOM Service is notified whenever new data is published in the MEx Broker, establishing communication with each AquaMOM device via a topic.

Once the data is received, the AquaMOM Service analyses, processes, and stores it in a database, making it accessible to the customers through a Web graphical user interface, as shown in Figure 4.3.

Figure 4.3 – Graphical User Interface of AquaMOM



Device Name: HOME
Mac: 00:1B:44:11:3A:B7
Address: street test - 911 - city test
Height: 1.80M
Base Radius: 1.58M
Current Volume: 10509.18L
Max Volume: 14117L
Remaining Supply: 105 days
Daily Consumption: 100L ●

Source: Author

The Graphical User Interface (GUI) was designed to be simple and user-friendly, allowing customers to register new cisterns by providing details, such as name, ID, and location. The main area features a panel displaying *Current Consumption*, showing water and battery consumption levels as percentages, and a *History Consumption* panel, presenting the history of monthly usage. A user-friendly feedback panel called *Reservatory Information* provides digital insights into the cistern's water supply and characteristics, such as dimensions, capacity, current water volume, daily consumption, remaining days of water availability, and battery level. As water is used, the circle's color changes from green to yellow to red, helping users easily track their consumption. This digital feedback is expected to be part of a customer engagement strategy to encourage users to adjust their water usage behaviors.

Regarding adaptation, AquaMOM uses the Managing System (see Section 3.7) to orchestrate adaptive changes in the MEx execution environment. When an adaptation is required, the MEx middleware sends an adaptation request to the Managing System. This approach ensures that AquaMOM can dynamically adjust to changing conditions without recompiling or stopping altogether.

Since AquaMOM devices are battery-powered and battery life is one of the most critical uncertainties, preserving it in challenging environments is essential. Therefore, customized parametric adaptation mechanisms developed in MEx to save energy in IoT devices have been used, as previously presented in (Cavalcanti; Carvalho; Rosa, 2021; Cavalcanti; Hughes; Rosa, 2023).

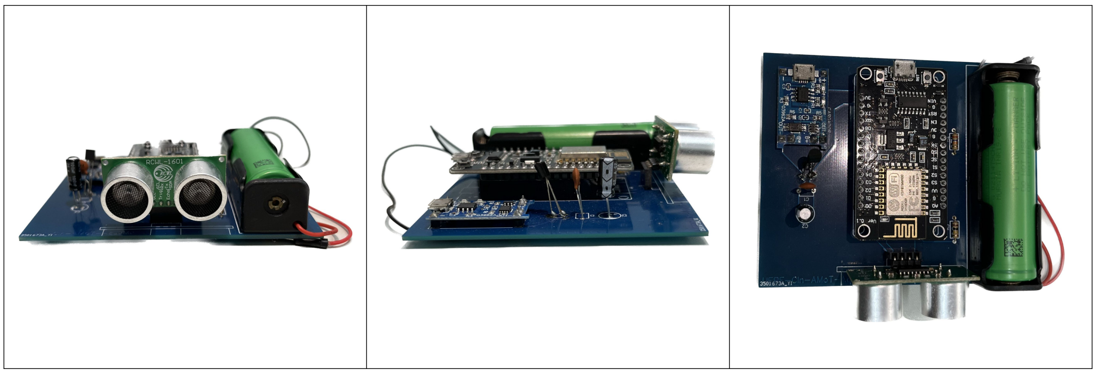
4.2.2 AquaMOM Device

Figure 4.4 shows the prototype of the proposed AquaMOM device, where the AquaMOM Client and MEx middleware are deployed and executed. This device was developed using available electronic components and could support the implementation of the concepts developed in the MEx framework.

The AquaMOM device is a hardware platform with off-the-shelf sensors to measure cistern water levels and a micro-controller. It uses an HC-SR04[3] ultrasonic sensor and a low-cost, open-source IoT controller named NodeMCU ESP8266 12E[4]. The controller features a single-core 32-bit processor operating at 160 MHz, with 160 KBytes (SRAM), integrated Wi-Fi (IEEE 802.11 b/g/n, 2.4 GHz), and supports executing software. Additionally, it has a TP4056[5] module for recharging the battery and preventing overcharging. The platform was selected for

3 <https://www.osepp.com/electronic-modules/sensor-modules/62-osepp-ultrasonic-sensormodule>
4 <https://www.espressif.com/en/products/modules/esp8266>
5 <http://www.tp4056.com/d/tp4056.html>

Figure 4.4 – Prototype of the AquaMOM Device



Source: Author

its cost and limited hardware configuration that can give insight into the performance of MEx.

Although the device is primarily designed for water level monitoring, its application is not limited and can be adapted for other fluid monitoring applications. The HC-SR04 sensor emits ultrasonic signals that bounce off the water's surface and return to the sensor. By measuring the time taken for the signal to return, the sensor calculates the distance to the water. Using this distance, combined with information about the cistern's dimensions and location, the application calculates the current volume of water in the cistern.

The schematic diagram of the AquaMOM device can be found in Appendix C, and the printed circuit board of AquaMOM device Appendix D.

4.2.3 Software Stack on AquaMOM Device

The software stack on the AquaMOM device comprises the AquaMOM Client and the MEx middleware. The AquaMOM Client is an IoT application responsible for measuring the water level in cisterns and monitoring the device's current power supply. After collecting this data, the AquaMOM Client uses the MEx middleware to publish a message containing this information to the messaging service (MEx Broker). The broker then forwards these messages to the AquaMOM Service. The MEx middleware is implemented using the MEx framework described in Section 3).

AquaMOM uses the MEx middleware for communication and parametric adaptation of the application. The adaptation improves the application's operation and adjusts it to environmental conditions or context-specific behaviors. For example, in AquaMOM, DCAM (see Section 3.4.5.2) was customized to change the frequency of measurements in response to environmental changes, such as variations in water levels (Cavalcanti; Hughes; Rosa, 2023), decreasing battery life, or even time of day.

Table 4.1 shows the rules applied for adjusting the monitoring frequency of the AquaMOM in response to water level variations. These rules are inspired by the river water level monitoring for flood early detection as proposed in Sulistyowati, Sujono and Musthofa (2017). It represents the water level information by the status of *Peace, Safe, Not Safe, and Danger*. It adjusts the deep-sleep time parameter according to the current water level.

Table 4.1 – Examples of rules implemented in the DCAM's Analyzer

Water Level	**Condition (%)**	**Event (min)**
Peace	Volume ≥ 75	*deep-sleep_time* $= 30$
Safe	$50 \leq$ Volume < 75	*deep-sleep_time* $= 15$
Not Safe	$25 <$ Volume < 50	*deep-sleep_time* $= 10$
Danger	Volume ≤ 25	*deep-sleep_time* $= 5$

Source: Author

These rules change the monitoring frequency by adjusting the *deep-sleep_time* parameter based on the water volume, which serves as the adaptation condition. Each condition has a fact, for example, *Volume* of the cistern, a conditional operator, for example, *lessThanInclusive*, and a reference value, for example, 25%. The rules engine runs a check to raise an event of its rules and determine if some condition was satisfied (3). This event should reconfigure the parameters associated with the rule, for example, *deep-sleep_time*. In practice, this rule expresses that an application monitors the water level of a cistern every 30 minutes when it is at a *Peace* level. In contrast, it may need to intensify monitoring (e.g., monitor every 1 minute) when the cistern reaches a *Danger* level, for example, below 25%.

Evaluating these rules makes it possible to decide whether an adaptation is necessary. If unnecessary, a new loop starts at the *Monitor* and waits for new adaptations. Otherwise, the *Analyzer* passes the adaptation decision to the *Planner* (4). The *Planner* receives the parameters that must change and creates a plan containing the sequence of actions that must be performed; for example, replace the parameter's value running on the IoT device with a new one obtained from the rule engine and update it. In this case, it creates a new configuration with the new values. It forwards it to the Executor running in MEx on the IoT device (5).

Finally, the Executor adapts the parameters by replacing the old configuration with a new one from the Planner (6). It is essential to mention that the entire adaptation process occurs without stopping the application or the IoT device entirely. Furthermore, the adaptation only occurs at the moment of application and middleware quiescence (Kramer; Magee, 2007).

Another adaptation strategy is the TDCAM (*Time-Based Duty Cycle Adaptive Mechanism*), which is a customized DCAM (see Section 3.4.5.2). TDCAM automatically adapts the duty cycle based on the time of day and usage patterns, reducing monitoring frequency during low-demand periods and increasing it during peak times. This dynamic adaptation helps to prolong battery life and ensures efficient energy consumption.

Table 4.2 shows the customized rules.

Table 4.2 – Examples of Rules implemented in the TDCAM's Analyzer

Time of Day (h)	**Event (min)**
0 AM $\leq$ Time of Day $<$ 8 AM	*deep_sleep* $= 60$
8 AM $\leq$ Time of Day $<$ 1 PM	*deep_sleep* $= 30$
1 PM $\leq$ Time of Day $\leq$ 11 PM	*deep_sleep* $= 1$

Source: Author

For example, there is no water consumption during the dawn, so it monitors the cistern's water level once an hour. In contrast, from $1PM$ to $11PM$, monitoring frequency intensifies to once every minute.

The adaptation process starts with the *Monitor* within the IoT device sensing information used in the adaption process, e.g., the timestamp and duty cycle. Next, the *Monitor* forwards these parameters to the *Analyzer* (2). The Analyzer examines the sensed data, checks with the historical in the *Knowledge Database*, and decides whether an adaptation is needed (3). If an adaptation is required, the *Analyzer* forwards this decision to the *Planner* (4). The *Planner* creates an adaptation plan, which consists of a sequence of actions (e.g., adjust the duty cycle parameter and upload it) that must be sent and executed on the IoT device (5). Finally, the *Executor* adjusts the parameter in the IoT device (6).

4.3 IMPLEMENTATION

The AquaMOM implementation[6] uses different programming languages and technologies, depending on the software component and device where it runs. The AquaMOM Client was implemented focusing on low-cost, low-power and low-CPU consuming operations. It was

[6] <https://github.com/davidjmc/middleware-extendify/tree/main/aquamom>

developed using MicroPython[7], compact enough to fit and run within micro-controllers. It was also necessary to avoid external libraries and focus on adopting native MicroPython objects (built-ins) to optimize the code with faster execution operations.

The AquaMOM Service is a cloud-based Web application with a GUI that displays information to customers. In addition to processing data according to business rules, the AquaMOM Service interacts with a database. It has two components responsible for creating and updating records and making data available to users. Whenever a new message is received, it processes and stores the data in the database, allowing user interaction through an API.

The AquaMOM Service was built using JavaScript. Data processing and business logic are implemented in Nest.js[8], a framework for building back-end applications. The Web interface was built using React.js[9], a library to create dynamic user interfaces. Finally, the database used in AquaMOM is a PostgreSQL[10] instance, a relational database system. The database schema can be found in Appendix B.

4.4 CONCLUDING REMARKS

This chapter presented AquaMOM, a holistic and adaptive IoT system designed to monitor water consumption in semi-arid regions. Initially, it presented a context and motivation, followed by some base concepts. Next, an overview of AquaMOM was introduced, followed by its software and hardware components. The chapter concluded with a summary of the implementation aspects.

7 <https://micropython.org/>
8 <https://nestjs.com/>
9 <https://react.dev/>
10 <https://www.postgresql.org/>

5 EVALUATION

"It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change."

—Charles Darwin

This chapter presents multiple experiments to evaluate MEx, its adaptation mechanisms, and AquaMOM. It begins by defining the objectives of the evaluation. Next, it divides the experiments into two scenarios and describes metrics, factors, and workload parameters used in the scenarios. Finally, it presents and discusses the obtained results.

5.1 OBJECTIVES

The evaluation presented in this chapter follows the steps proposed in Jain (1991) and has four main objectives:

- **Objective 1**: To compare the performance of AquaMOM built atop different MEx middleware flavors with an existing widely adopted middleware based on MQTT.
- **Objective 2**: To measure the impact of MEx adaptation on the performance of AquaMOM.
- **Objective 3**: To show the adaptive mechanisms in action, evaluating their impact on reducing uncertainties and improving AquaMOM's performance to ensure efficient operation.
- **Objective 4**: To estimate the potential change in water consumption when using the AquaMOM.

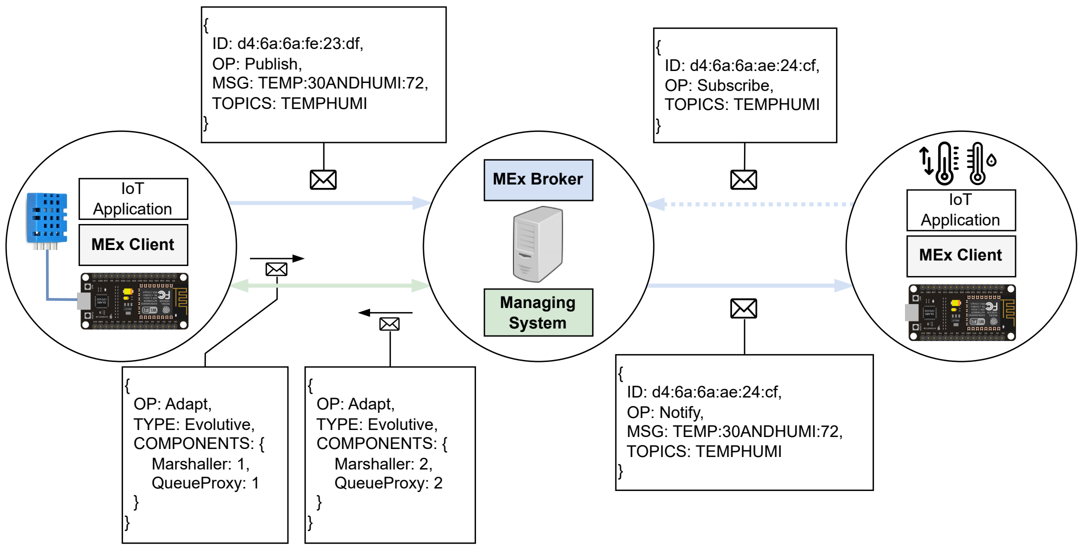
Two scenarios were defined to achieve these objectives. In the first scenario (Scenario 1), experiments focus on the implementation aspects of MEx. In the second scenario (Scenario 2), the experiments involve running AquaMOM on top of MEx middleware systems.

5.2 SCENARIO 1

Figure 5.1 shows the elements of Scenario 1. This scenario has a publish/subscribe application that monitors the temperature and humidity. The *Publisher* continually measures the temperature and humidity using a digital DHT11[1] sensor attached to the IoT device and publishes them (e.g., *TEMP:30ANDHUMI:72*) to a topic called *TEMPHUMI* on the MEx Broker. A *Subscribe* interested in these values subscribes to the same topic and is notified by the MEx Broker when a new message arrives. This application was also implemented atop Mosquitto (Light, 2017), a widely adopted MQTT-based middleware.

The *Publisher* and *Subscriber* were executed on two separate devices having a similar configuration: NodeMU ESP8266 12E[2], with a Tensilica Single-Core 32-bit L106 processor, 160 MHz (*megahertz*) clock speed, 36 kBytes (*kilobytes*) of SRAM, integrated Wi-Fi, low-power consumption and running MicroPython[3] Version 1.12. This device was selected due to its limited hardware configuration and low cost, providing real insight into the performance of MEx and making it accessible for cheap IoT projets. The MEx Broker and the *Managing System* were executed on a PC with an Intel Core i7 processor, 2.70 GHz (*gigahertz*), 16 GB (*gigabytes*) of RAM, running Linux Mint 19.1 Cinnamon[4] 64-bit and Python 3.6.9.

Figure 5.1 – Scenario 1



Source: Author

[1] http://www.aosong.com/en/products-21.html
[2] https://components101.com/sites/default/files/2021-09/ESP12E-Datasheet.pdf
[3] https://micropython.org/
[4] https://www.linuxmint.com/rel_tessa_cinnamon.php

5.2.1 Metrics, Parameters and Factors

The performance metric adopted in all experiments was the *publishing time*, which is measured on the publisher side. This is the time elapsed between the IoT application on the IoT device publishing a message and receiving the confirmation that the MEx Broker received it. The sent message goes through all components of the MEx Client.

Network time (time in the transport layer) was not considered due to its high variability. It is worth noting that IoT devices use WiFi for Internet access, and network conditions can fluctuate, causing congestion or intermittent issues. However, experiments were scheduled during times of lower network usage, primarily at night, to mitigate these issues.

Two parameters were kept fixed in all experiments: publication interval and adaptation mechanism. The interval between successive message publications was fixed at 5 seconds, and the publisher sent out 1000 messages. The adaptation mechanism used was the *Evolutive Mechanism* (see Section 3.4.5.1). Then, every time a new version of one or more MEx components becomes available, the adaptation is triggered to replace the old component with the new one. In practice, a new version means that a file (the new component version) needs to be uploaded from the component library to the IoT device.

Several parameters (factors) changed during experiments, as shown in Table 5.1. These factors exploited different situations when executing the application.

Table 5.1 – Factors of Scenario 1

Factor	**Level**
Adaptation	Disabled, Enabled
Replaceable Component	Marshaller/Pickle, Queue Proxy, Marshaller+Queue Proxy
Adaptation Interval	5 min, 10 min, 15 min, 20 min
Middleware Flavor	Std-MEx, Thin-MEx, MQTT

Source: Author

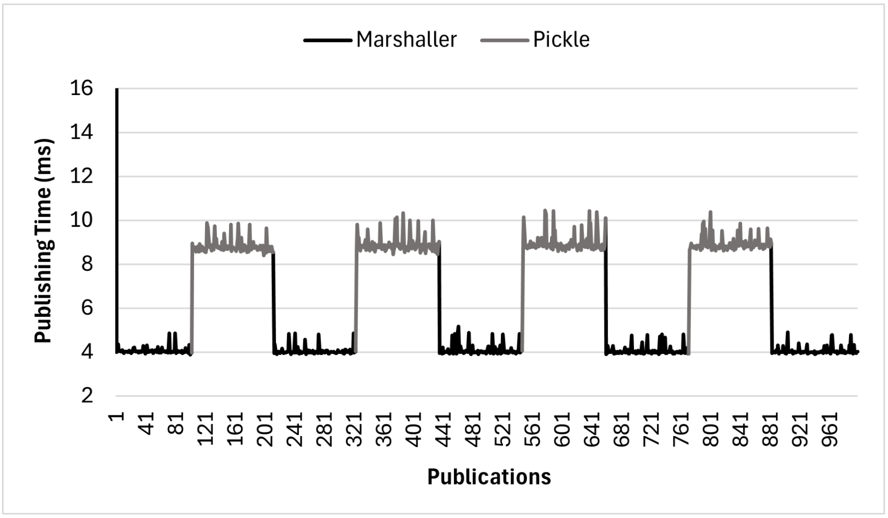
The first factor (*Adaptation*) defines whether the adaptation is enabled or not. When enabled, MEx triggers an adaptation if necessary. Otherwise, MEx remains static and does not change at runtime. Factor (*Replaceable Component*) exploits the possibility of adapting (replacing) one or more components simultaneously: replace *Marshaller*(*649 bytes*) by *Pickle*, replacement of *QueueProxy*(*992 bytes*) and simultaneous replacement of *Marshaller* and *QueueProxy*. The *Marshaller* uses a native serialization specifically implemented for MEx, and *Pickle*[5] is an object serialization of Python.

In relation to the *Adaptation Interval*, four values were configured: 5, 10, 15, and 20 minutes. These intervals were used to evaluate different adaptation frequencies, as no standard values exist, assessing their impact on system performance while balancing responsiveness and resource usage. Finally, three different middleware flavors were used: *Std*-$\mathrm{ME}\textsc{x}$, *Thin*-$\mathrm{ME}\textsc{x}$ and MQTT. The Std-$\mathrm{ME}\textsc{x}$ is a basic implementation of an adaptive middleware using $\mathrm{ME}\textsc{x}$ framework. Thin-$\mathrm{ME}\textsc{x}$ has the $\mathrm{ME}\textsc{x}$ components necessary to provide the same functionality as the MQTT without a marshaller, i.e., the application needs to serialize messages before sending them. Finally, the MQTT flavor is the Mosquitto (Light, 2017), a widely adopted implementation of MQTT. It is worth observing that MQTT has been used as a standard protocol for implementing publish/subscribe middleware systems in IoT environments (Al-Fuqaha et al., 2015).

5.2.2 Adaptation in Action

The first experiment shows the *Evolutive Mechanism* in action. In this case, the serialization component of the middleware is alternated at each adaptation interval, i.e., every 10 minutes in this experiment. The component *Marshaller*(faster) is replaced by component *Pickle* (slower). Figure 5.2 shows the behavior of the *Publishing Time* during the experiment.

Figure 5.2 – Alternation of the serialization component ($Marshaller$ and $Pickle$)



Source: Author

The results indicate that the *Publishing Time* varies depending on the serialization com-

[5] https://docs.python.org/3/library/pickle.html

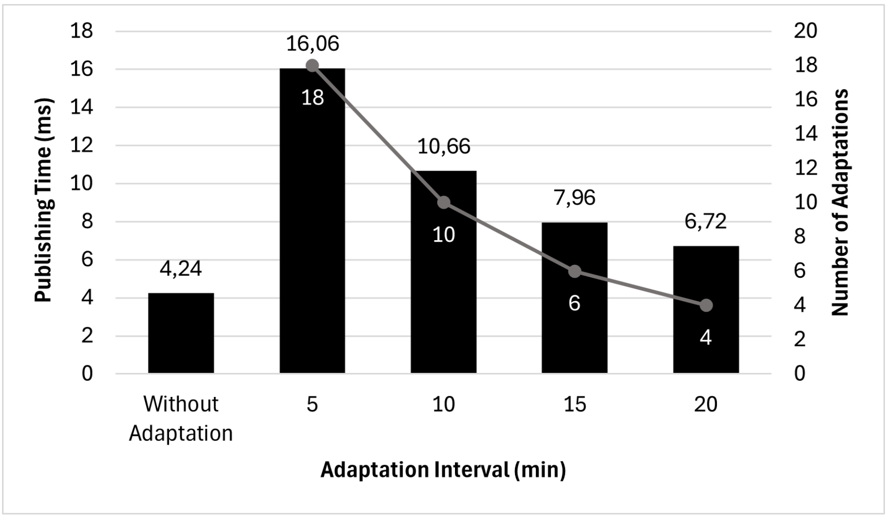
ponent used. The *Pickle* serialization results in higher publication times of 8 ms, while the *Marshaller* achieves lower times of 4 ms. Additionally, the experiment shows that $\mathrm{ME}\textsc{x}$ serialization is 50% more efficient than *Pickle*, leading to improved application performance. This increase in efficiency is likely a result of optimizations made during the development of $\mathrm{ME}\textsc{x}$ serialization, which were specifically designed for the IoT environment.

5.2.3 Impact of Adaptation

Previous experiments demonstrated the effectiveness of the *Evolutive Mechanism*. To investigate its impact on IoT applications further, new experiments were conducted in which the mechanism was either enabled or disabled. When the mechanism was enabled, either one component ($Marshaller$) or two components ($Marshaller$ and $QueueProxy$) were replaced simultaneously. Similarly to the previous experiments, the assessed metric was *Publishing Time*, and the intervals between adaptations were set according to the specifications in Table 5.1.

Figure 5.3 shows the *publishing time* and *number of adaptations* when the adaptation mechanism is enabled/disabled and a single component is replaced.

Figure 5.3 – Impact of Adaptation on the Application (one component)

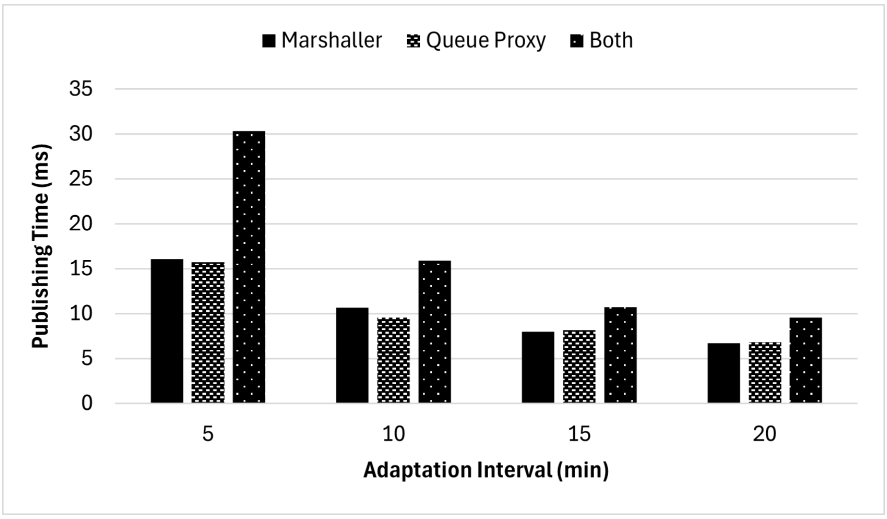


Source: Author

As expected, the effect on *Publishing Time* is more significant when the adaptation mechanism is active. Without adaptation, the average *Publishing Time* was 4.24 ms. However, with adaptation, the impact increased as the frequency of adaptations rose. For example, with the adaptation interval set to 5 minutes, the *Number of Adaptations* performed was 18, and the average *Publishing Time* increased to 16.06 ms. As the adaptation interval increased, for example, with the adaptation interval set to 20 minutes, only 4 adaptations occurred, and the average *Publishing Time* was reduced to 6.72 ms, representing a 58.2% reduction. As this adaptation interval becomes longer, the impact of adaptation is reduced, approaching the publishing time without adaptation. This point is important because the frequency of adaptations in this experiment is extrapolated to assess its impact. In a real-world scenario, adaptations can occur in hours, days, or even months.

Figure 5.4 shows the impact on the publishing time when one ($Marhsaller$) or two components ($Marshaller$ and $QueueProxy$) are replaced. It is worth noting that the code size of *Queue Proxy* (992 bytes) is 52.9% larger than *Marshaller* (649 bytes). Furthermore, in this experiment, both the *Marshaller* and *Queue Proxy* components are replaced by alternative versions of themselves to assess the impact of component replacement, taking into consideration whether the size of the component influences the impact.

Figure 5.4 – Impact of Adaptation on the Application (Two components)



Source: Author

Observing the results, the impact of replacing individual components on the publishing time is very similar, regardless of their sizes and the configured adaptation interval. However, the impact is greater when both components are adapted simultaneously, causing an increase in publish time. This behavior was expected because adaptation involves uploading files from the *Managing System* to the IoT device (see Figure 5.1). Meanwhile, similarly to the previous experiments, this impact is reduced as the adaptation interval becomes longer, suggesting that adaptations need to have a longer interval to occur.

The higher impact observed when replacing two components simultaneously compared to one indicates that the process of updating multiple components simultaneously can introduce additional overhead. The experiment also revealed a limitation in adaptation when updating multiple components simultaneously. Specifically, adapting three components simultaneously caused a memory overflow on the ESP8266 devices, which interrupted the application's operation and rendered the adaptation infeasible. This issue highlights the overhead introduced by simultaneous updates and underscores the need for strategic adaptation scheduling to minimize performance degradation. Despite attempts to address the problem through code optimizations, garbage collection removal, and increased adaptation intervals, no solutions were successful. Given that ESP8266 devices are low-cost and have extremely limited computational resources, these adaptation strategies might perform better on devices with higher capabilities, such as a Raspberry Pi[6].

5.2.4 Comparative Evaluation

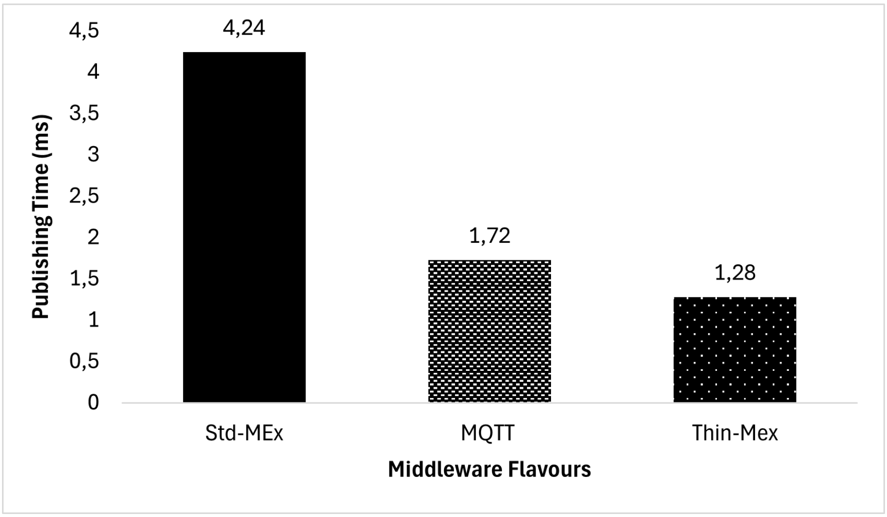
The final experiments in Scenario 1 compared the performance of the Std-MEx and MQTT, i.e., they exploit the implementation of the same publish/subscribe application atop different middleware flavors (see Section 5.2.1).

Figure 5.5 shows the average *Publishing Time* of the application. As expected, the adaptation mechanism impacts the application's performance, which cannot be overlooked. However, the ability to dynamically adjust the middleware behavior according to changes in IoT environments and the possibility of creating specific MEx configurations tuned to each device's characteristics can compensate for the performance overhead. MQTT has both a monolithic code and the same configuration, whatever the device. Finally, it is worth noting that the customized version of MEx (Thin-MEx) with an average *Publishing Time* of 1.28 ms performs better than MQTT with 1.72 ms.

The comparison indicates that while Std-MEx introduces performance overhead due to its adaptation mechanism, it offers significant benefits in customization for specific devices. The improved performance of Thin-MEx demonstrates that a tailored approach can achieve better efficiency compared to standard protocols like MQTT.

6 <https://www.raspberrypi.com/documentation/computers/getting-started.html>

Figure 5.5 – MEx versus MQTT



Source: Author

5.3 SCENARIO 2

Scenario 2 includes AquaMOM (see Section 4.2) and uses an empirical approach that combines the AquaMOM device (see Section 4.2.2) with simulated water consumption behavior in a controlled environment. It follows a traditional IoT application centralized architecture, incorporating a controller, sensors, and an application for data display (Singh; Ahmed, 2021; Muccini et al., 2018).

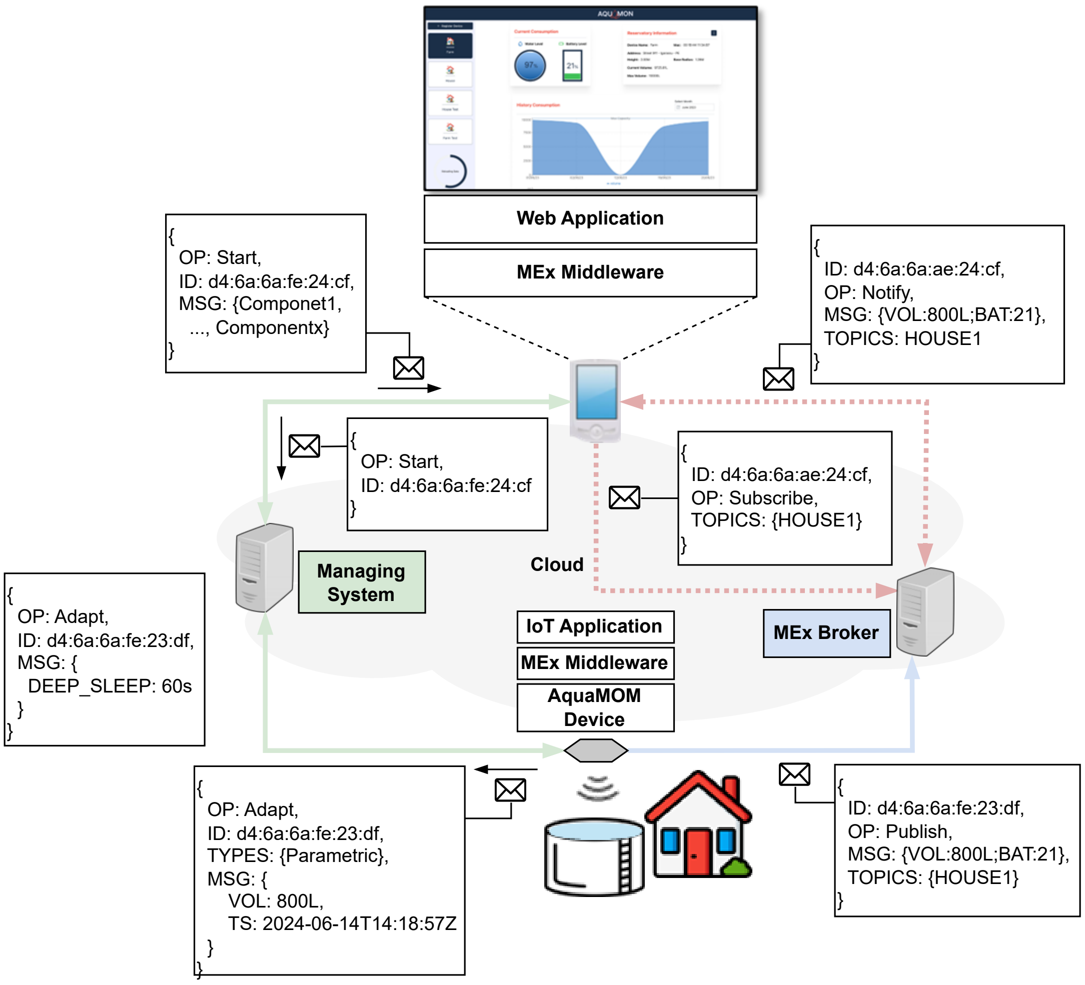
Figure 5.6 shows the elements of Scenario 2. The AquaMOM device hosts a publisher that periodically monitors a home cistern water level. A distributed Web application (subscriber) runs on the cloud and processes and displays the monitored data. The publisher and the subscriber were implemented atop MEx middleware systems. It is important to mention that as the subscriber is a Web application, it was implemented on MEx middleware built with JavaScript components.

5.3.1 Metrics, Parameters and Factors

Two metrics were adopted in all experiments of Scenario 2: *publishing time* and *power consumption*. The *publishing time* is similar to one used in Scenario 1 (see Section 5.2.1). The *power consumption* is the amount of energy consumed by a thing during its execution, specifically while realizing the duty cycle (Aslanpour; Gill; Toosi, 2020) (see Section 3.4.5).

Table 5.2 presents the factors adopted in the experiments of Scenario 2. In this table, the

Figure 5.6 – Scenario 2


Web Application
MEx Middleware
{ OP: Start, ID: d4:6a:6a:fe:24:cf, MSG: {Componet1, ..., Componentx} }
{ ID: d4:6a:6a:ae:24:cf, OP: Notify, MSG: {VOL:800L;BAT:21}, TOPICS: HOUSE1 }
{ OP: Start, ID: d4:6a:6a:fe:24:cf }
{ ID: d4:6a:6a:ae:24:cf, OP: Subscribe, TOPICS: {HOUSE1} }
Managing System
Cloud
{ OP: Adapt, ID: d4:6a:6a:fe:23:df, MSG: { DEEP_SLEEP: 60s } }
IoT Application
MEx Middleware
AquaMOM Device
MEx Broker
{ OP: Adapt, ID: d4:6a:6a:fe:23:df, TYPES: {Parametric}, MSG: { VOL: 800L, TS: 2024-06-14T14:18:57Z } }
{ ID: d4:6a:6a:fe:23:df, OP: Publish, MSG: {VOL:800L;BAT:21}, TOPICS: {HOUSE1} }

Source: Author

MEx *Configuration* and *Adaptation Interval* are similar to the ones adopted in Scenario 1 (see Table 5.1). Two adaptation mechanisms are being considered, namely DCAM and TDCAM (see Section 3.4.5). Three middleware flavors have been utilized: Std-MEx, MEx-MQTT, and MQTT. Std-MEx is a basic middleware built using the MEx framework. MEx-MQTT is a middleware incorporating an MQTT-proxy component from the MEx framework's library, and MQTT is similar to Scenario 1. Two different intervals between publications (*Inter-publication time*) are being adopted: one for DCAM and another one for TDCAM. The difference in intervals is due to the distinct adaptation strategies used by each mechanism. The interval for DCAM is in seconds, as it monitors the continuous behavior of water usage in the cistern, where consumption is constant and requires frequent updates to adjust the publication frequency. In contrast, TDCAM uses discrete intervals based on the time of day, such as morning, afternoon, or night, which reduces the monitoring frequency, e.g., by decreasing the frequency during the night. Finally, the results consider the publication of 50 messages. This sample size was chosen to balance data collection with battery consumption. In the case of TDCAM, the battery is often allowed to deplete, limiting the number of messages published.

Table 5.2 – Factors of Scenario 2

Factor	**Levels**
Adaptation	Enabled, Disabled
Adaptation Mechanism	DCAM, TDCAM
Adaptation Interval	1 min, 5 min, 10 min, 15 min, 20 min
Middleware Flavor	Std-MEx, MEx-MQTT, MQTT
Inter-publication time (DCAM)	5 s, 15 s , 30 s, 45 s, 60 s
Inter-publication time (TDCAM)	1 min, 30 min, 60 min

Source: Author

5.3.2 Evaluating the MEx's Performance

In the first set of experiments of Scenario 2, the performance of the middleware flavors was compared, considering different deep-sleep times. Figure 5.7 shows the results[7] to compare the performance of Std-MEx, MEx-MQTT and MQTT against different intervals between publications. Although the results visibly suggest that the mean *Publishing Time* of MEx middleware systems appears to be better, statistical t-tests indicate no significant difference between Std-MEx, MEx-MQTT, and MQTT. However, MEx middleware systems achieved a mean *Publishing Time* approximately 13% faster (70 milliseconds) than MQTT when the interval between publications was set to 30 seconds.

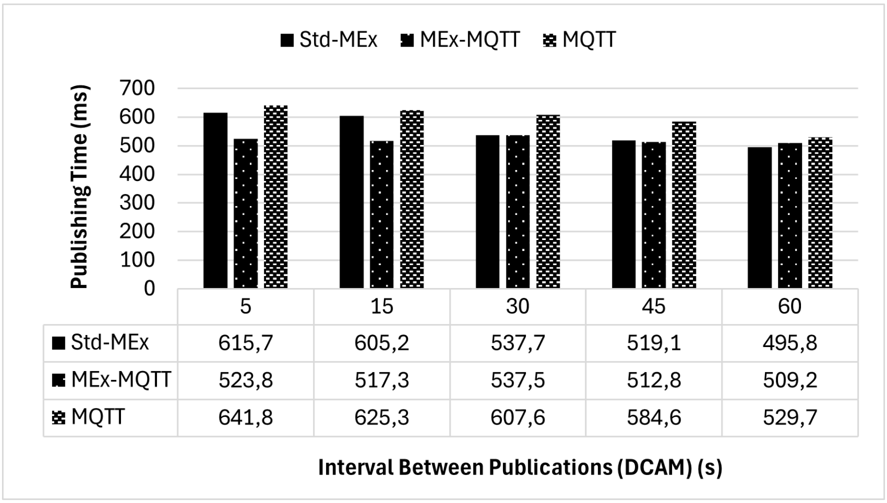
As the interval between publications increases, the middleware flavors improve the publishing time, with MEx middleware systems maintaining a consistent advantage in all cases. In the end, MEx has similar performance while supporting different middleware implementations and offering dynamic adaptation capabilities.

The similar performance observed in both implementations, i.e., the Std-MEx and MEx-MQTT, indicates the effectiveness of the MEx framework. This results suggests that MEx exhibits versatility in implementing different middleware systems while maintains good performance, even when incorporating external libraries, such as MQTT.

From Figure 5.7, it is also possible to compare the performance between the MEx-MQTT and MQTT implementation, both utilising Mosquitto Broker. Results show that the perfor-

[7] Evaluation results available at http://tinyurl.com/ijcs-evaluation-results

Figure 5.7 – Comparison of Publishing Time Between Std-MEx, MEx-MQTT, and MQTT



	5	15	30	45	60
■ Std-MEx	615,7	605,2	537,7	519,1	495,8
■ MEx-MQTT	523,8	517,3	537,5	512,8	509,2
▨ MQTT	641,8	625,3	607,6	584,6	529,7

Source: Author

mance of MEx-MQTT also slightly improves over the MQTT. The MEx-MQTT presented a mean *publishing time* of 12.67% lower than that of MQTT.

As observed, when the interval between publications is set to 5 seconds, MEx-MQTT achieves a mean *publishing time* of 523.8 ms, whereas MQTT recorded 641.8 ms. Similarly, with a interval between publications of 60 seconds, the mean *publishing time* of MEx-MQTT was 509.2 ms, compared to MQTT's 529.7 ms. These results relate to how the components are designed and implemented in MEx, i.e., they naturally have an awareness of the resource constraints of IoT environments. For instance, the component managing socket connections in the MEx Broker operates slightly differently from MQTT. It retains the MEx client's connection status, and a new connection is only initiated if one does not already exist.

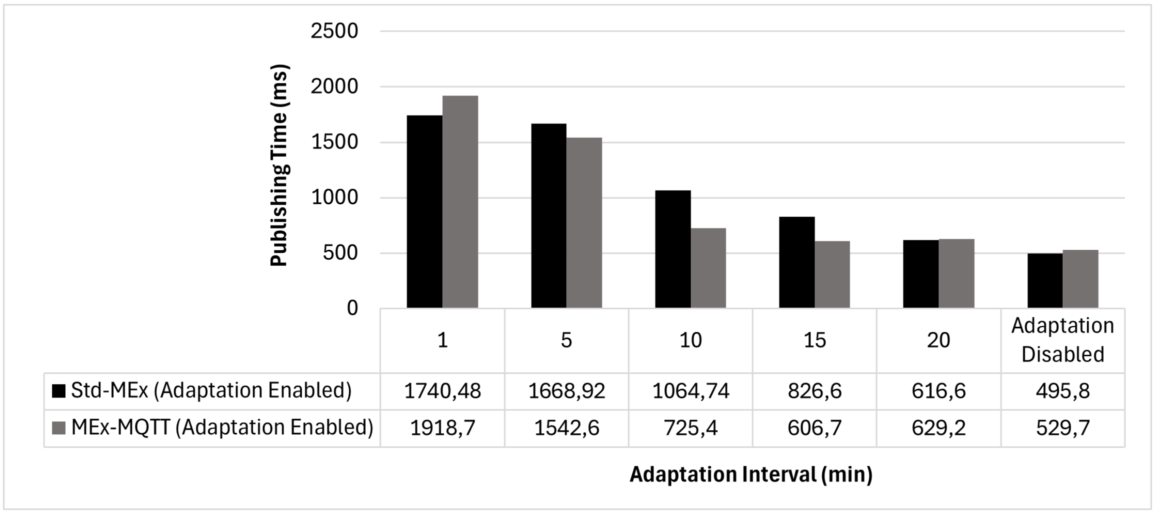
Overall, the comparable performance of the two middleware implementations indicates the effectiveness of the MEx framework. The results further suggest the potential of MEx as a solution that delivers customizable adaptive middleware systems for IoT applications, offering flexibility and efficient communication while adapting to changing environmental conditions. This adaptability enables developers to optimize communication protocols based on specific operational needs, enhancing overall system responsiveness and reliability.

5.3.3 Impact of Adaptation on Performance and Power Consumption

In this second set of experiments in Scenario 2, the impact of adaptation on the application was measured. It is worth observing that a configurable interval adaptation time is crucial to help define the optimal adaptation timing. Frequent adaptations may lead to increased computational resource consumption and potential impacts on application reliability, while excessively long intervals may hinder responsiveness. In this evaluation, DCAM was selected as the adaptation mechanism due to its ability to dynamically adjust the duty cycle by modifying the *deep-sleep_time* parameter based on the application context, specifically water levels. This mechanism was chosen for its potential to balance performance and power consumption, which are the metrics evaluated in this assessment. DCAM adjusts the monitoring frequency to match environmental conditions, reducing energy consumption while ensuring reliable water level monitoring, particularly in resource-constrained and remote environments.

Figure 5.8 shows how the adaptation affects the *publishing time* across various adaptation intervals.

Figure 5.8 – Impact of Adaptation on Performance



	1	5	10	15	20	Adaptation Disabled
■ Std-MEx (Adaptation Enabled)	1740,48	1668,92	1064,74	826,6	616,6	495,8
▨ MEx-MQTT (Adaptation Enabled)	1918,7	1542,6	725,4	606,7	629,2	529,7

Source: Author

As expected, the impact of adaptation on *publishing time* is elevated compared with the situation in which the adaptation is disabled, especially at shorter adaptation intervals. However, as *adaptation interval* increases, this impact decreases. For example, with an *adaptation interval* of 1 minute, the *publishing time* stands at 1740.4 ms. When the *adaptation interval* becomes longer (20 minutes), there is a significant reduction of the *publishing time* to 616.6 ms. These results show that while frequent adaptations can degrade performance, longer adaptation intervals have a lower impact on the application performance.

Similar behavior is observed when the focus is on the results of the MEx implementations only (adaptation enabled). They suggest that varying middleware architectures (Std-MEx and MEx-MQTT) have minimal impact on performance. When the *adaptation interval* is 20

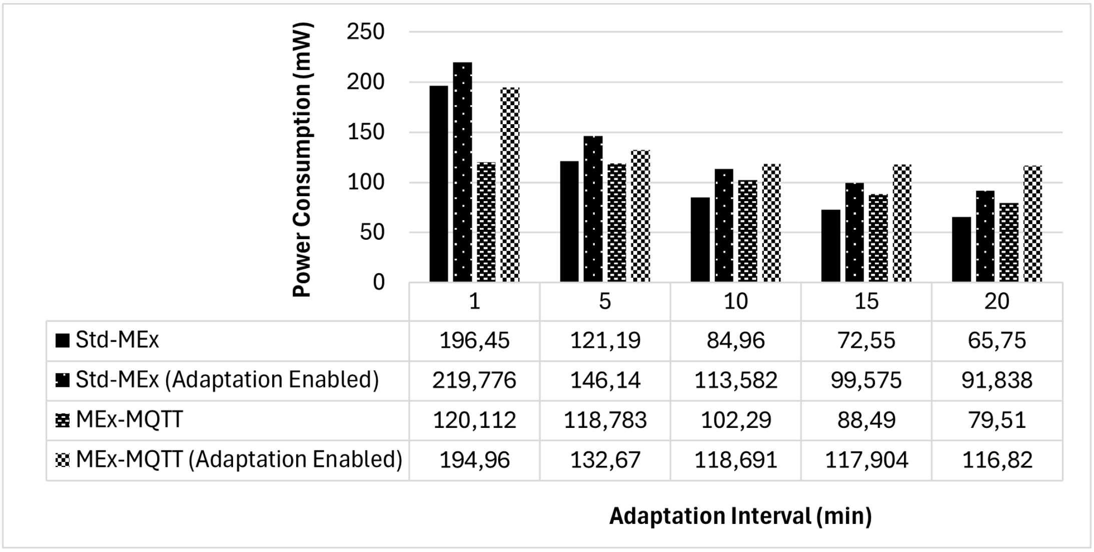
minutes, the *publishing time* for Std-MEx is 616.6 ms, while for MEx-MQTT is slightly higher at 629.2 ms. Finally, it is worth observing that there is an extrapolation in the frequency of adaptations in these experiments to show its impact. In real-world scenarios, *adaptation intervals* may extend to hours, days, or even months.

Concerning the *power consumption*, Figure 5.9 presents the impacts of adaptation on *power consumption* across different *middleware flavors*, adaptation *disabled* or *enabled* and *adaptation intervals*. As observed, incorporating adaptation introduces a slight increase in *power consumption* due to the additional computational effort required by the adaptive mechanism.

Firstly, the average *power consumption* increases by approximately 16 mW when comparing the standard MEx (Std-MEx) to the MEx integrated with MQTT (MEx-MQTT) in a scenario where the adaptation is disabled. Enabling adaptation in the MEx-MQTT setup, referred to as MEx-MQTT (with Adaptation Enabled), results in a further average increase of about 34 mW. Secondly, when comparing Std-MEx to Std-MEx (Adaptation Enabled), there is an average increase of approximately 26 mW in *power consumption*.

However, the difference in *power consumption* decreases as the *adaptation interval* extends beyond 20 minutes. The overall consumption can decrease, especially as the application remains in a sleep state for longer. This behavior suggests that longer sleep durations and extended adaptation intervals can lead to more efficient power usage in adaptive systems.

Figure 5.9 – Impact of Adaptation on Power Consumption



	1	5	10	15	20
■ Std-MEx	196,45	121,19	84,96	72,55	65,75
▪ Std-MEx (Adaptation Enabled)	219,776	146,14	113,582	99,575	91,838
▦ MEx-MQTT	120,112	118,783	102,29	88,49	79,51
▨ MEx-MQTT (Adaptation Enabled)	194,96	132,67	118,691	117,904	116,82

Source: Author

In summary, the results show that the adaptation has a low impact on both *publishing time* and *power consumption* while allowing for necessary adjustments in the application.

This balance enables responsiveness without compromising resource efficiency. The minimal performance trade-offs observed with longer adaptation intervals suggest that MEx is well-suited for applications operating in dynamic environments where efficient resource management is crucial.

5.3.4 Evaluating the Impact of the Adaptation on Battery Lifetime

Section 3.4.5.2 mentions that power consumption is essential since IoT devices are battery-powered, and applications operate in unpredictable environments. In this context, after evaluating the impact of adaptation on performance and power consumption, it is necessary to understand how the adaptation mechanisms, DCAM and TDCAM, help extend the life batteries of IoT devices.

Initially, some experiments were carried out with each middleware flavor to understand the behavior of *duty cycle* when the deep sleep changes. It is worth noting that the *deep-sleep* duration is equivalent to the *inter-publication time* (DCAM) shown in Table 5.2. In addition, the *duty cycle* is calculated using Equation 3.1. Table 5.3 shows that the duty cycle decreases by increasing the deep sleep time. For example, the highest (59.2%) and lowest (14.3%) duty cycles occur with a deep sleep time of 5 seconds and 60 seconds, respectively. However, the duty cycle was slightly different according to the middleware flavor. For example, when the deep sleep time was set to 5 seconds, the duty cycle values were 59.2% (Std-MEx), 61.2% (MEx-MQTT), and 58.3% (MQTT). This behavior is expected since the duty cycle represents the percentage of a device's working time that periodically switches between periods of inactivity (e.g., sleeping) and activity (e.g., connection, sampling, and transmission) periods. Within these active periods, some activities can present unpredictable behavior.

Table 5.3 – Duty Cycle Behavior of Middleware Flavors

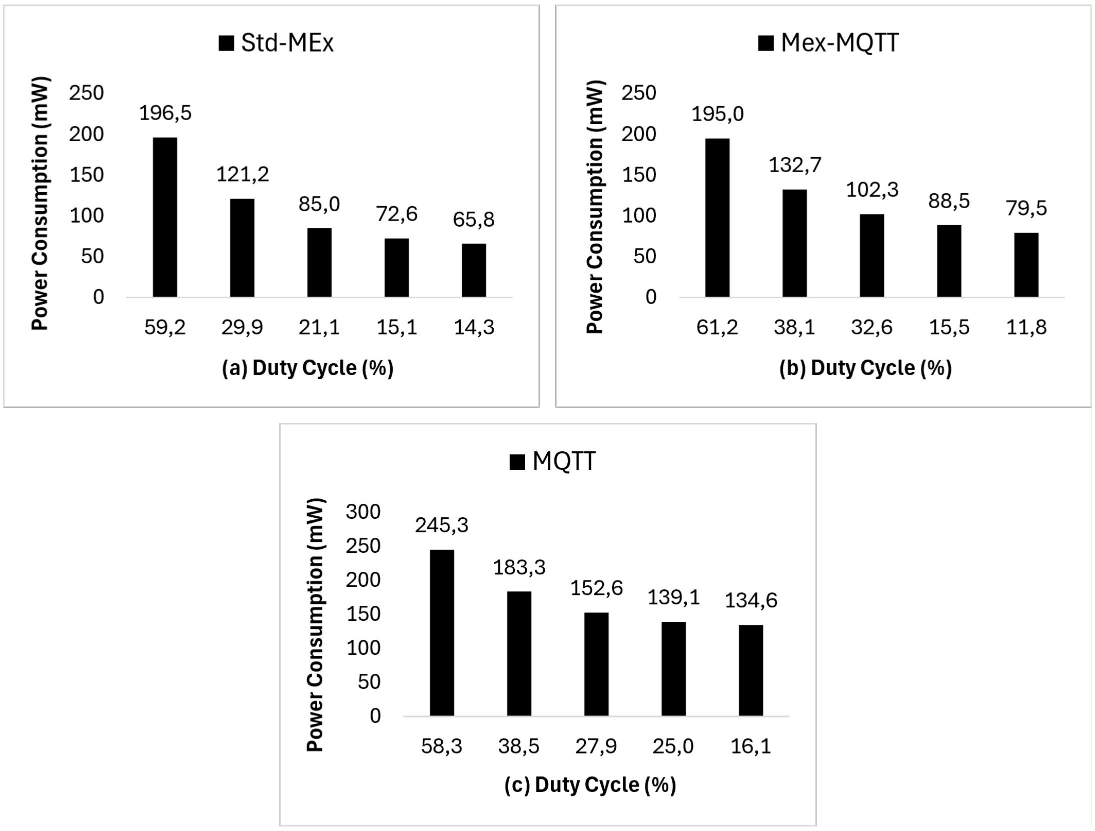
Deep-sleep (s)	**Duty Cycle (%)**		
	Std-MEx	MQTT-MEx	MQTT
5	59.2	61.2	58.3
15	29.9	38.1	38.5
30	21.1	32.6	27.9
45	15.1	15.5	25.0
60	14.3	11.8	16.1

Source: Author

The duty cycle's alternated between decreasing and increasing the electrical current consumed in the activity/inactivity periods during the execution. As expected, the electrical current consumed during activity periods is high, averaging around 90 *miliampères* (mA). In contrast, the electrical current consumption significantly decreased during deep sleep, averaging about 19 mA.

Figure 5.10 presents the power consumption of each middleware flavor. The *power consumption* depends on the duty cycle's current operating condition, which can be in active/inactive periods. Figure 5.10(a), 5.10(b) and 5.10(c) shows the *power consumption* of the Std-MEx, MQTT-MEx and MQTT, respectively.

Figure 5.10 – Power Consumption (varying duty cycle)

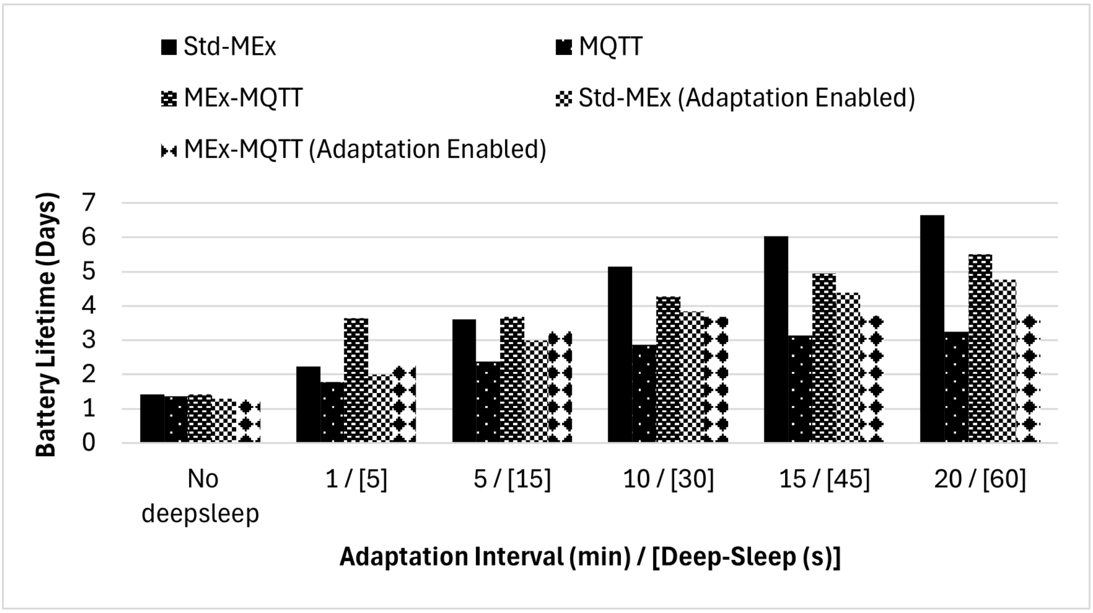


Source: Author

The *power consumption* of Std-MEx and MEx-MQTT is lower than MQTT. This result is important because it shows that MEx reduced the power consumption while allows adaptability, which is absent in MQTT. The results also indicate that the duty cycle significantly impacts the device's *power consumption*. A lower duty cycle corresponds to reduced *power consumption*. The possibility of adjusting the duty cycle based on application requirements can reduce or increase The *power consumption*, e.g., if the water cistern is complete, the monitoring (duty cycle period) can be adjusted to more extended periods (e.g., 30 minutes), thus reducing *power consumption* in non-critical scenarios.

In the following experiments, the AquaMOM device was equipped with a 3000 mAh (*milliampere-hour*) battery. Figure 5.11 shows the estimated lifetime of the AquaMOM device with DCAM, considering *deep-sleep_times* ranging from 5 to 60 seconds and *adaptation intervals* from 1 minute to 20 minutes. The results indicate that adjusting the deep-sleep duration can extend the device's lifetime from 1.4 to 6.6 days.

Figure 5.11 – Estimation of Battery Lifetime (DCAM)

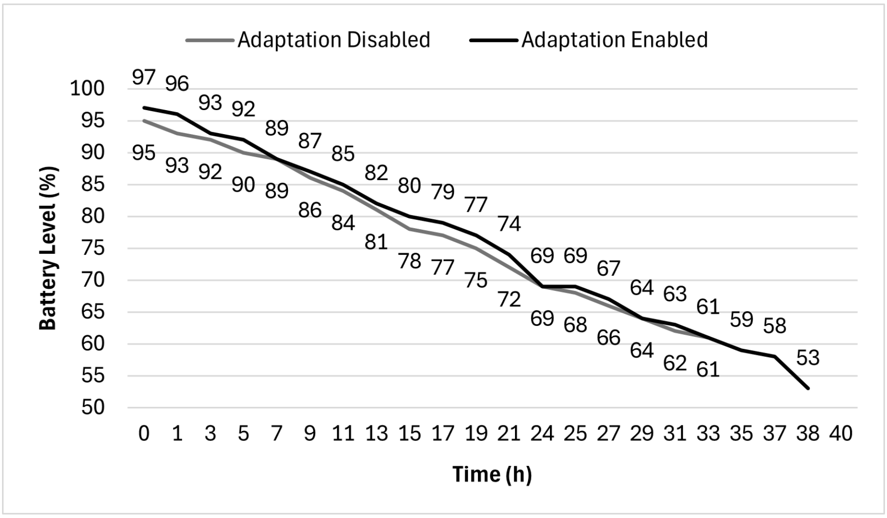


Source: Author

It is worth noting that these deep sleep values, utilized in seconds, were chosen to extrapolate the experimental scenario, as applications usually consider deep-sleep in minutes. Moreover, there is also extrapolation in the frequency of adaptations as mentioned before. The battery lifetime may increase as this deep sleep becomes longer. However, long deep sleep times imply that the application may not run during some critical events, potentially affecting application reliability. At the same time, as the adaptation interval becomes longer, the impact of adaptations on power consumption is also reduced, improving and increasing the thing's lifetime.

To evaluate TDCAM, the AquaMOM device was again equipped with a 3000 mAh battery and run until the battery was depleted. Without adaptation, it monitored the water level every minute. With adaptation, TDCAM adjusted the monitoring frequency according to the customized rules outlined in Table 4.2. Figure 5.12 shows the results[8] of this experiment.

Figure 5.12 – Estimated lifetime with variable deep-sleep using TDCAM



Source: Author

With adaptation, the AquaMOM device consumed 45.36% of its battery over 38 hours, while without adaptation, it consumed 35.79% over 33 hours. Despite the higher battery consumption, the adapting device operated for 5 hours longer than the non-adapted one, demonstrating its advantage in scenarios where battery life is crucial. Furthermore, the adaptation capacity provides customization and flexibility, improving aspects beyond battery consumption, such as overall device performance and application reliability. These facts are possible by allowing adjustments to the middleware and application either before (development) or after (runtime) deployment based on specific user needs or environmental conditions, making adaptation beneficial for various usage contexts.

In summary, introducing adaptive capabilities is very interesting once an application can adjust its duty cycle to align with the application's requirements without heavily impacting the application's *power consumption*. Spending some energy adjusting a duty cycle that offers the shortest or longer deep sleep regarding the application's context is better than randomly selecting one static deep-sleep configuration that can influence the application´s availability and reliability.

[8] Evaluation results available at https://tinyurl.com/isc22024-results

5.3.5 Evaluating the impact of using AquaMOM

For the final set of experiments, AquaMOM is combined with a domestic water consumption simulation. In this simulation, the human consumption behaviors of residents from the semi-arid region are being considered as presented in Marzall and Nascimento (2023). They show the hourly evolution of water consumption inside homes, with peak water usage times around noon and between 6 PM and 8 PM, with significantly lower consumption between midnight and 5 AM.

The metric analyzed in this experiment is *daily water consumption* (measured in liters per day). The factors being used are the *feedback mechanism* and *time of day*, with their corresponding levels summarized in Table 5.4. This metric aims to quantify the water consumed in each simulation scenario, serving as the key performance indicator to evaluate the efficiency of the different feedback mechanisms.

Table 5.4 – Factors Impacting Water Consumption with AquaMOM

Factor	**Levels**
Feedback Mechanism	Without Feedback, With Feedback, Autonomous Regulation
Time of Day	Peak Hours (noon, 6 PM–8 PM), Normal Hours, Dawn Hours (midnight–5 AM)

Source: Author

To compare the water consumption with and without the digital feedback provided by AquaMOM, three simulation scenarios were considered: consumption without feedback, i.e., he/she consumes water throughout the day without restrictions; consumption with digital feedback, i.e., he/she consumes water considering the feedback provided by AquaMOM; and autonomous consumption, i.e., a smart valve automatically regulates the water flow without the participation of the consumer.

In all experiments, the daily water consumption was set to 100 liters (L), as specified in (França et al., 2010). In addition, it was assumed that the consumption of 6 and 10 liters at peak and normal hours, respectively. Meanwhile, the consumption of 0 liters/hour was assumed for 6 hours at dawn. In the experiments without feedback, it is assumed 3 liters over water consumption in peaks, non-peaks, and dawn hours. In the experiments with feedback, the customer receives feedback at three levels: *Green*, *Yellow*, and *Red*, as shown in Equation 5.1.

$$Red_{Fb}(c) = \begin{cases} \text{Green}, & c \leq 25L \\ \text{Yellow}, & 25L < c \leq 50L \\ \text{Red}, & c > 50L \end{cases} \tag{5.1}$$

where, c is the customer's water consumption per hour, and $Red_{Fb}(c)$ (*Reduction by Feedback*) represents the percentage reduction in customer water consumption each hour based on the feedback received. If the digital feedback is *Green*, the cumulative water consumption is below 25 L, and the customer has no concerns yet, resulting in a 0% reduction. For *Yellow* feedback, where cumulative water consumption is between 25 L and 50 L, a reduction is randomly calculated between 0 and 1. This value is normalized to a range of 4.2% to 8.5%, as found in (Liu; Mukheibir, 2018). For *Red* feedback, with consumption above 50 L, the same calculation is used but with a higher weight, multiplying by 2, and capped at a maximum of 8.5%, reflecting more conservative behavior.

A policy of flow rate regulation is defined to simulate the water consumption reduction in the scenario with a smart valve, which adjusts water flow for more efficient usage. This policy must vary according to cumulative consumption and the time of day, as shown in Equation 5.2.

$$Flow_{limit}(h) = \frac{100 - \sum_{i=0}^{h} \textit{Consumption}(i)}{24 - h}, \tag{5.2}$$

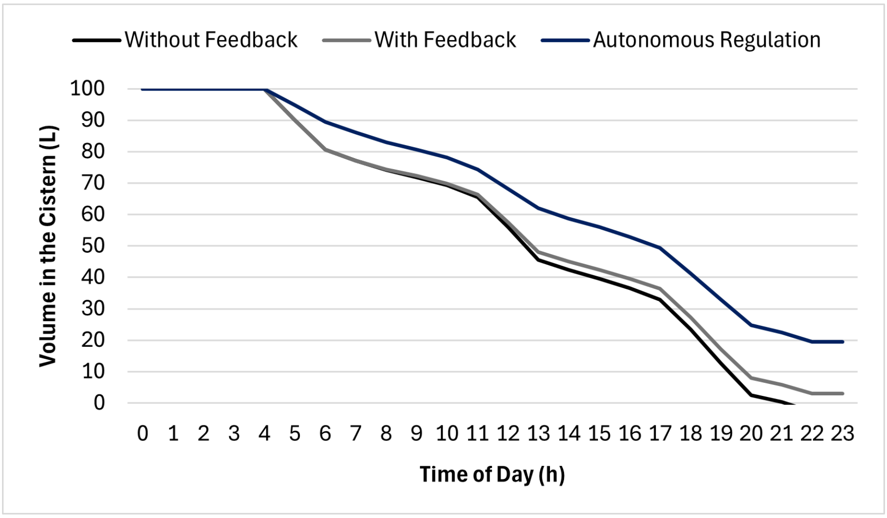
where, h represents the current hour of the day (ranging from 0 to 24), and $Flow_{limit}$ is the new flow rate limit of the smart valve. The numerator represents the volume of water available for consumption at that time, which equals the anticipated 100 L per day minus the volume already consumed up to h. The denominator is the number of hours remaining in the day, calculated as 24 minus the current time of day.

Figure 5.13 presents the results[9] of the *daily consumption behavior* for each simulation experiment. For customers without feedback, the simulation showed an overuse of 102.8 L/day, ceasing around 21:00 hrs. With consumption feedback, customers adjusted their behavior, reducing their consumption from 102.8 L/day to 96.9 L/day by 23:00 hrs. Finally, the autonomous regulation that uses the smart valve enhances water efficiency, reducing consumption to 80.6 L/day, nearly 20 L less than without feedback.

Based on these results, an estimate for 16.000 L indicates that without feedback, this volume lasts about 5 months; with feedback, the duration increases by 10 days, extending to

[9] Evaluation results available at https://tinyurl.com/isc22024-results

Figure 5.13 – Comparison of simulation results for different behaviors



Source: Author

about 5 months and 10 days, representing a 6.67% increase. With the smart valve, there is an increase of 1 month and 13 days, representing a 28.67% increase.

During the simulations, some sensor readings differ significantly from the expected values. These outliers may stem from sensor inaccuracies or uncontrolled variables such as Internet connections or power supply. Hence, these values were removed from the results and analysis to ensure the relevance of the simulation results.

5.4 CONCLUDING REMARKS

This chapter presented the experimental evaluations conducted with MEx and AquaMOM. First, the evaluation objectives were presented. Then, Scenario 1 was introduced, including its metric, parameters and factors, followed by the experiments and their results. Next, Scenario 2 and its metrics, parameters and factors were presented. The results of the experiments, focusing on the performance of MEx and its impact on AquaMOM's operation, were then discussed. Finally, the social impact of reduced water consumption through the use of AquaMOM was presented.

6 RELATED WORK

"If I have seen further it is by standing on the shoulders of Giants."

—Isaac Newton

This chapter presents and analyses existing works related to what is being proposed in this thesis. It discusses how the works were categorized. Next, each category is explained in detail. Finally, a comparative analysis highlights how the existing solutions differ from MEx.

6.1 OVERVIEW

Existing approaches related to what is proposed in this thesis can be organized into four groups: IoT middleware frameworks, IoT adaptive middleware systems, IoT non-adaptive middleware systems and other works.

The first group surveys existing solutions for implementing IoT middleware systems instead of a specific middleware instance, i.e., typically middleware frameworks. These solutions provide services and models that enable communication and interaction between devices and applications, offering a foundation for IoT system integration. The second group focuses on IoT middleware systems having adaptive capabilities, while the third group discusses IoT middleware systems without adaptation. The last group includes solutions not directly related to the middleware domain but central mechanisms adopted in the design of MEx, such as energy-saving and smart water applications.

Finally, before analyzing the related works, it is worth noting that middleware systems built using MEx are referred to as MEx middleware

6.2 MIDDLEWARE FRAMEWORKS

Among the frameworks, Park and Park(Soojin; Sungyong, 2019) proposed an adaptive cloud-based middleware framework for developing IoT applications through collaboration services. This framework enables developers to collaboratively utilize IoT services among multiple applications and devices collaboratively, facilitating the implementation and adaptation of IoT applications when necessary. This solution adopts MAPE-K control loop as a cloud service.

Park et al.(Park; Song, 2015) presented a self-adaptive IoT middleware framework for mobile applications. This framework identifies user interface usability issues and dynamically reconfigures the application's graphical interface accordingly. It monitors and analyses user behavior to identify usability issues, such as frequent screen scrolling, and determines the underlying causes. Based on these findings, the solution selects an appropriate adaptation strategy and creates an adaptation plan to change the user interface. However, the adaptation only takes effect upon restarting the application.

Hassan et al. (Hassan et al., 2023) proposed PlanIoT, a middleware framework for enabling adaptive data flow management in IoT-enhanced environments such as smart buildings. PlanIoT uses automated planning methodologies and a generic QoS model to evaluate data flow performance in edge infrastructures (exchanged between applications and devices) and QoS configurations (network resource allocation, priority policies). PlanIoT evaluates this data flow performance under various configurations and generates a performance metrics dataset. This dataset serves as input to automated planning representations to satisfy the QoS requirements of deployed applications, including response times and resource allocation. Additionally, PlanIoT organizes IoT applications into categories based on their requirements, allowing tailored management of data flows and resource prioritization.

6.3 ADAPTIVE MIDDLEWARE

Several IoT adaptive middleware systems share similarities with MEx middleware systems, but they generally concentrate on adaptations at application or network levels rather than the middleware itself.

Achilleos et al. (Achilleos et al., 2017) proposed ARM (Adaptive Runtime Middleware) as a service-oriented middleware platform to address IoT device heterogeneity and interoperability. ARM enables distributed IoT application development using Open Services Gateway Initiative (OSGi)[1] (OSGi Working Group, 2024) framework and RESTful architectural pattern to manage application heterogeneity, scalability, and adaptability. It dynamically generates code at runtime from annotations, facilitating the injection of new service interfaces into IoT devices. It provides two core services implemented as OSGi components: one for detecting devices and

[1] <https://www.osgi.org/resources/where-to-start/>

their resources through OSGi annotations, while the other for generating RESTful service interfaces exposed by a REST service for device usage. The dynamic nature of ARM is primarily enabled by OSGi, which permits the installation, starting, and changing of services used by devices at runtime.

Rausch, Nastic, and Dustdar (Rausch; Nastic; Dustdar, 2018) proposed EMMA, an edge-enabled publish/subscribe middleware designed for client mobility and QoS optimization (e.g., message delivery guarantees) in edge computing applications. EMMA implements a protocol to monitor the network conditions and dynamically reconfigures network parameters based on monitored real-time metrics, such as network latency and broker load balancing. EMMA is based on MQTT, provides low-latency communication, and facilitates the distribution of messages to geographically dispersed locations.

Portocarrero et al. (Portocarrero et al., 2016) proposed SAMSON (Self-Adaptive Middleware for Wireless Sensor Networks), a middleware for managing wireless sensor networks and adapting sensor behavior based on application context and requirements. SAMSON is developed utilizing an ADL-specified software architecture, where each architectural element becomes a software component deployed and executed on the Contiki sensor network platform(Dunkels; Gronvall; Voigt, 2004). SAMSON's adaptation mechanism follows the MAPE-K loop, distinguishing between three types of sensor nodes: base nodes, manager nodes, and managed nodes. SAMSON offers two adaptation strategies: 1) adjusting sensor node parameters, such as reconfiguring sensor tasks, topologies, and clusters by selecting active nodes, and 2) reprogramming sensor nodes using dynamic loading to extend node lifetime, e.g., changing sensor node configuration due to low battery charge or converting a managed sensor node into a manager node.

Uribarren et al. (Uribarren et al., 2008) presented CAHIM (Configurability, Adaptability, Heterogeneity, and Interoperability Middleware), a middleware that facilitates interoperability between heterogeneous pervasive applications and IoT devices. This interoperability occurs using a standard syntax for describing service interfaces and adopting a standard communication protocol. CAHIM coordinates application cooperation and adaptation to environmental changes, such as adding new devices and available services, failures, and new user preferences. Adaptations are automatically triggered by applications or based on user-defined preferences.

Rahman et al. (Rahman et al., 2018) proposed a multi-sensor adaptive IoT platform for environmental pollution measurement in smart cities. This platform comprises a Raspberry Pi for control and detection tasks, an Arduino Nano for sensor control and detection, and a set of sensors collecting temperature, air quality, and multimedia data, e.g., local images and sounds. Given potential Wi-Fi unavailability, the platform utilizes 3G data connections, which are limited and expensive for uploading data to the cloud. To address this limitation, this platform includes two adaptation strategies. Firstly, an infrastructure-level adaptation approach dynamically adjusts sensor reading intervals based on the bandwidth of the contracted 3G plan to keep data volume within budget. Secondly, an information-level adaptive strategy utilizes a middleware that enables containers with specific sensors to process data locally or in the cloud. These containers determine whether to process raw data to reduce the volume of processed data or to send it to the cloud based on user-specified criteria, such as avoiding overuse of the contracted 3G data plan.

Mohalik et al. (Mohalik et al., 2016) proposed InteropAdapt, an adaptive middleware that facilitates control interoperability between applications and IoT devices. InteropAdapt dynamically orchestrates application workflows based on detected dynamic events, ensuring alignment between application functionalities and device operations. These dynamic events refer to changes in application or device resource sets or contexts, such as alterations in application requirements, device capabilities, or device mobility. Inspired by the MAPE-K, InteropAdapt continuously monitors applications and devices for dynamic events. Upon detecting such dynamic events, it analyzes whether there have been any changes to the application's command set or device capabilities. It plans new workflows or modifies existing ones if necessary, orchestrating their execution corresponding to the application's functionality. To ensure that application functionality and device capabilities match, InteropAdapt adopts an ontology. Furthermore, InteropAdapt stores static knowledge about the context and knowledge gained during the execution of dynamic events for forecasting and initial workflow generation and optimization.

Han, Mehrotra, and Venkatasubramanian (Han; Mehrotra; Venkatasubramanian, 2019) presented AquaEIS (Aqua Event Identification System), a middleware for monitoring and identifying events in distributed environments, particularly water infrastructure networks. AquaEIS integrates data from various sources, including IoT devices, geophysical and user-collected data, and simulation and modeling tools. Its operation comprises two phases. In the offline phase, AquaEIS generates profiles of anomalous events and optimizes the placement of IoT devices to enable the detection of failures, such as breaks and leaks in pipes. In the online phase, AquaEIS combines models of water distribution networks with data from different sources and employs an adaptation mechanism, such as valve control, to maintain continuous system operation and predict or mitigate impacts in real time. AquaEIS is implemented with a logic loop that monitors, analyzes, and adapts, transforming incoming data streams from various sources into actionable information to adapt water infrastructure at the application level. This adaptation serves multiple purposes, including identifying vulnerable points in the water infrastructure, swiftly identifying and locating problems, preventing further failures, and predicting future events.

Peros, Joosen, and Hughes (Peros; Joosen; Hughes, 2021) proposed Ermis, a middleware solution that automatically adapts sensor sampling periods at runtime to match application requirements in IoT environments. Ermis addresses the challenge of ensuring that IoT sensors' data generation rate aligns with applications' data processing rate despite the inherent dynamism of IoT infrastructures, such as sensor mobility, failure, changing application data requirements, and varying message transmission delays. Ermis configures sensor sampling periods by continuously monitoring both the state of the IoT infrastructure and the application's data requirements to match the data generation rate with the data processing rate of applications. It automatically adjusts sensor sampling to ensure data processing efficiency and energy conservation without relying on prior knowledge of the underlying infrastructure.

Pradeep, Krishnamoorthy, and Vasilakos (Pradeep; Krishnamoorthy; Vasilakos, 2021) proposed AUM-IoT, an Adaptive Ubiquitous Middleware, which considers the situational context of the applications, devices, or people and the contexts of the network formed and accordingly adapts the behavior of the IoT ecosystem. AUM-IoT is a multi-agent, multi-communication middleware that acts as an integration point for applications to access context, share it with other applications and have services made available via a multi-communication protocol bridge. AUM-IoT includes an agent manager, which coordinates communication and manages entities, and an agent registry that stores agent profiles. Additionally, AUM-IoT incorporates a service allocation model to optimize resource utilization and minimize average response time, enhancing users' Quality of Experience (QoE). The authors present AUM-IoT's capabilities through a use case involving a smart classroom and smart healthcare scenario. This use case highlights the middleware's ability to adapt dynamically to context changes and provide appropriate services in different domains. AUM-IoT emphasizes context-aware adaptation IoT by enabling interaction between devices and services through multi-agent and multi-protocol communication mechanisms in response to changing contexts.

Ahmed (Ahmed, 2022) proposed MPaS (Micro-services based Publish and Subscribe), a scalable microservices-based publish/subscribe middleware for IoT that dynamically adapts to varying workloads. MPaS combines concepts from software-defined networking and fractal theory to decompose the middleware into control and data planes. Controllers manage client subscriptions and their filtering criteria (predicates). Brokers handle the routing of messages to subscribers based on these predicates. Using fractal theory, brokers also scale dynamically by self-replicating on demand. When the number of predicates registered for a given broker exceeds its capacity (a preconfigured number), a new broker is created to distribute the dynamic load balancing and scaling. Similarly, new controllers are activated to manage the increased demand when the number of connected clients exceeds a given maximum. This self-replication mechanism ensures scalability without relying on resource-intensive tools, making it lightweight and efficient. Inspired by fractal principles, MPaS scales vertically in the same layer (new brokers) and horizontally across devices (new controllers).

Jung et al. (Jung et al., 2024) proposed ImmunoPlane. This middleware enables adaptivity in IoT applications, allowing them to handle failures and network congestion in different runtime infrastructures while meeting their requirements with minimal user effort. It provides a Domain-Specific Language (DSL) that allows users to declaratively state application-specific requirements, such as high availability or minimum throughput. Then, it produces an adaptive deployment plan based on the given infrastructure and those user-provided application requirements. This deployment plan determines where components should run (e.g., on edge or in the cloud) on the available resources and the location of faults that can occur in that infrastructure.

Hassan et al. (Hassan et al., 2024) proposed PlanEMQX, a publish/subscribe message broker architecture for adaptive data exchange in IoT environments. This architecture improves traditional brokers by incorporating an Automated Configuration Planner that generates configuration and adaptation plans and an Adaptive Data Flow Broker responsible for managing IoT data flows based on these plans. These components work together to automate configuration and adaptation. PlanEMQX enables dynamic data flow adjustment, modifying drop rates or priorities in response to changes in IoT environments or evolving application requirements, ensuring efficient data exchange under varying conditions. PlanEMQX focuses on automating configuration and managing data flows based on application QoS requirements through configuration plans at the broker level.

6.4 NON-ADAPTIVE MIDDLEWARE

IoT Middleware systems without adaptation capabilities have been divided into ones executing outside the things and ones running on the devices.

6.4.1 Off-device Middleware

Off-device IoT middleware solutions neither run directly on the IoT device nor the edge. Instead, they typically operate on cloud platforms. Consequently, these solutions require communication gateways for integration with IoT devices.

SmartComm (Agostinho et al., 2018) is an IoT middleware that adopts a microservices architecture to facilitate messaging and the exchange of services running on devices between gateways and various heterogeneous devices in smart homes. The middleware has a module for facilitating communication between gateways and IoT devices and a module for exchanging information between devices and services. Each device has an associated microservice for interaction with the gateway and other devices. This architecture enables IoT devices to send messages to others or collect sensor data through microservices using an intermediary gateway.

Joseph et al. (2017) proposed an IoT middleware for smart cities, enabling communication among several heterogeneous public utility systems within the city, such as smart transport, smart water management, and security systems. This middleware allows authorities to effectively manage the data infrastructure of these public utility services operating within the urban environment. The middleware implements an IoT gateway that supports MQTT, Hypertext Transfer Protocol (HTTP), HTTPS, and Constrained Application Protocol (CoAP) protocols, ensuring concurrent connections, managing data traffic, and featuring a software layer as the entry point for all systems' data. Moreover, it facilitates the communication between these utility systems and their data sources, including data collection services, sensors, and controllers. Additionally, it implements messaging and routing systems for forwarding data to designated systems and offers security services, such as authentication and encryption of queued information.

Elkhodr, Shahrestani and Cheung (2016) proposed IoT-MP (IoT Management Platform), focusing on device privacy and location management. Its distributed architecture includes managed devices (MTs), manager systems, and a manager of managers. These elements work together to ensure devices are lightweight, mobile across heterogeneous environments and communicate transparently with other devices or applications. MTs are IoT devices equipped with agent systems for communicating with the manager while the manager orchestrates operational roles, such as requests, responses, and actions on the managed devices. The manager of managers facilitates access to MT data for IoT applications. Finally, IoT-MP contains security and privacy modules that provide authentication, authorization, network confidentiality (integrity), and user privacy protection functionalities.

6.4.2 Device-based IoT Middleware

Mosquitto (Light, 2017) is an open-source lightweight messaging solution designed for IoT devices with limited resources. It allows clients (publishers and subscribers) to operate directly on the IoT device. Additionally, it provides a messaging broker that executes outside the thing. When a publisher sends a message, the Mosquitto broker receives it and then delivers it to all subscribers who have subscribed to a particular topic. This middleware follows the publish/subscribe pattern, where devices interested in specific information subscribe to topics, and those publishing data to those topics are known as publishers. The Mosquitto broker acts as an intermediary, ensuring the seamless exchange of messages between publishers and subscribers, enhancing the efficiency and scalability of the communication process.

HiveMQ[2] simplifies the development of IoT applications through an MQTT broker that provides efficient messaging and communication. HiveMQ enables the integration of IoT devices and systems running MQTT clients through the publish/subscribe pattern.

6.5 OTHER WORKS

As mentioned in Section 6.1, some related works have not focused on the middleware design but on mechanisms that have been implemented in MEx.

6.5.1 Energy-saving Approaches

Minimizing power consumption has led to the emergence of conscious energy-efficiency approaches. These approaches have used different energy-saving strategies, such as energy-aware data routing and acquisition, duty cycling, and energy harvesting (Abdul-Qawy; Almurisi;

[2] Available in: https://www.hivemq.com/

Tadisetty, 2020). In addition, similar to the proposed mechanisms, some approaches consider adaptive energy-saving aspects, such as dynamically reconfiguring energy-aware parameters at runtime.

Ramachandran et al.(Ramachandran et al., 2016b) propose Dawn, a middleware that dynamically adjusts the bandwidth of IoT networks based on applications' demand executing on the IoT nodes. Dawn monitors and detects changes in the application's bandwidth requirements and runs an algorithm to estimate the new needed bandwidth for adjustments. The adjustments are made by altering the time slots available for node communication. Reducing the number of time slots means the node stays longer sleeping (inactive) and reduces power consumption. In contrast, the adaptation mechanisms proposed in MEx saves power consumption by dynamically adjusting the duty cycle of the IoT device based on their application context, offering a more context-aware energy-saving approach.

Ramachandran et al.(Ramachandran et al., 2016a) also proposed a middleware able to dynamically modify how IoT data are aggregated dynamically at the nodes. This proposed middleware allows application developers to classify their network traffic into high-priority and low-priority categories and uses application knowledge to perform data aggregation. The aggregation strategy is combined with the priority scheme so that unused payloads of high-priority communications transport aggregated data of low-priority ones. This approach reduces power consumption by minimizing the data transmitted by each node. However, the adaptive energy-saving mechanisms implemented in MExreduce power consumption through runtime adjustments to the duty cycle of IoT devices, specifically tailored to the application context.

Venanzi et al.(Venanzi et al., 2019) propose PEND (Power Efficient Node Discovery), a solution designed explicitly for IoT-Fog environments to improve Bluetooth Low Energy (BLE) node discovery. It leverages the location awareness of the nodes given by the fog paradigm and effectively triggers the discovery process based on the advertisers' arrival frequency and only when it is strictly needed. In this way, it saves power by introducing an adaptive strategy to dynamically adjust the BLE interface, deciding when to switch BLE interfaces on/off based on the expected frequency of the node approximation. Unlike this related work, our power consumption reduction strategy adjusts the duty cycle dynamically and does not switch the node interfaces on/off.

Kim et al.(Kim; Kang; Rim, 2019) propose the DDC-MAC (Dynamic Duty-Cycle-Medium Access Control) protocol to reduce transmission delays and power consumption in IoT environments. It implements an algorithm that adjusts the duty cycle ratio to increase the number of wakeups of the receiving nodes. This algorithm uses Early Acknowledgment at peak and off-peak times to transmit data to reduce the delay time and minimize power consumption. In addition, DDC-MAC changes the window size of transmitting nodes according to the traffic congestion for various IoT devices. Instead of proposing a communication protocol and an algorithm to adjust the duty cycle ratio, MEx works at the middleware level. It uses a rules engine and application context for those adjustments.

Finally, Munir et al.(Munir et al., 2018) propose a method to optimize the duty cycle period in IoT devices equipped with energy harvesting systems. Considering that a duty cycle has an active and non-active period, this method changes the thing's non-active time to adjust the power consumption during the period of activity. The objective is to achieve an Energy-Neutral Operation (ENO). ENO is the mode of operation where the thing's power consumption is always less or equal to the energy harvested. It uses an exponentially weighted moving-average filter to predict how much energy is harvested during the active period. Next, it estimates the minimum duration of the non-active period to harvest enough energy to adjust the next duty cycle. While Munir changes the duty cycle based on the harvested energy, MEx uses a rule-based approach and application context to adjust the duty cycle.

6.5.2 Smart Water Management

The use of IoT applications for smart water management has gained significant attention due to their potential to improve water resource management and sustainability. Researchers have applied these applications for various purposes, such as monitoring water consumption, quality detection and prevention issues, and alerts to accidents and disasters (Singh; Ahmed, 2021). These applications are implemented in different areas, such as smart cities, smart agriculture, and disaster prediction.

Regarding water quality, IoT applications monitor aspects such as pH, conductivity, oxygen levels, and total dissolved solids in real-time. These solutions have gained prominence in smart cities, where they help determine whether water is safe for consumption and identify potential sources of contamination (Mukta et al., 2019; Pujar et al., 2020; Lakshmikantha et al., 2021; Tubio et al., 2023; Kumar et al., 2024). Unlike these approaches, AquaMOM focuses on monitoring water consumption, prioritizing its conservation instead of water quality.

Researchers have also proposed solutions for monitoring water levels. Generally, current water level monitoring solutions are designed to analyze water consumption, detect leaks, and monitor real-time water levels in remote locations (Malche; Maheshwary, 2017; Singh; Ahmed, 2021; Saritha et al., 2023; Kumar et al., 2024; Essamlali; Nhaila; El Khaili, 2024). Although these solutions share the same idea of water level monitoring with AquaMOM, they do not account for the challenges of harsh environments, such as aging and inadequate infrastructure, limited resources and remote or difficult-to-access locations. These solutions assume ideal environmental conditions and do not seek solutions to address uncertainty issues, which can compromise their proper functioning.

Other solutions have been developed with a similar focus on monitoring water levels in challenging environments (Kumar et al., 2019; Sulistyowati; Sujono; Musthofa, 2017; Hassan et al., 2020; Thirumarai et al., 2024; Ranieri et al., 2024; Lee et al., 2024; Han; Mehrotra; Venkatasubramanian, 2019). Respectively, these solutions include continuous water level monitoring to control motors automatically, remote monitoring to protect personnel in dangerous situations, monitoring river water levels to detection systems to provide early flood warnings, and combining artificial intelligence to determine damaged infrastructure, contamination, and water consumption. However, unlike AquaMOM, none of these solutions incorporate adaptive concepts to address the operational uncertainties in such challenging environments.

6.6 COMPARATIVE ANALYSIS

Table 6.1 provides a comparative summary of MEx and related works. The table is organized into dimensions that reflect the core characteristics of MEx, aligning with the design decisions discussed in Section3.2. It is important to note that this table focuses specifically on adaptive IoT middleware solutions, which are the central focus of this thesis.

The comparison is organized into three main dimensions: *Execution Environment*, *Framework*, and *Adaptability*. The *Execution Environment* dimension refers to the operational environment where the solution operates: on IoT devices, cloud environment (the Cloud), or both. The *Framework* dimension evaluates whether the solution works as a framework or as an IoT middleware system. The *Adaptability* dimension distinguishes between static solutions (not allowing runtime adaptation) and dynamic solutions (adaptable at runtime).

Adaptive works were further analyzed using five sub-dimensions derived from the taxonomy of adaptive systems proposed by Krupitzer et al.(Krupitzer et al., 2015) (see Section 3.2.5). These sub-dimensions allow a more granular basis for comparing MEx with related works. Columns without responses are marked as NA (Not Applicable), indicating either the absence of adaptation or that the dimension is irrelevant to the related work or was not identified during the analysis.

Table 6.1 – Summary of Related Works

Work	**Execution Environment**	**Framework**	**Adaptability**					
			Static	**Dynamic**				
				Why	**When**	**Where**	**What**	**How**
Agostinho et al. (2018)	Cloud	No	Yes	NA	NA	NA	NA	NA
Elkhodr, Shahrestani and Cheung (2016)	Cloud	No	Yes	NA	NA	NA	NA	NA
HiveMQ[3]	Cloud and IoT Thing	No	Yes	NA	NA	NA	NA	NA
Joseph et al. (2017)	Cloud	No	Yes	NA	NA	NA	NA	NA
Light (2017)	Cloud and IoT Device	No	Yes	NA	NA	NA	NA	NA
Achilleos et al. (2017)	Cloud	No	No	Changes in Context and Changes in User Requirements	Reactive	Application	Structure	External, Goals and Centralized
Han, Mehrotra and Venkatasubramanian (2019)	Cloud	No	No	Changes in Context	Reactive	Application	Parameters	External, Models and Centralized
Mohalik et al. (2016)	Cloud	No	No	Changes in Context	Reactive	Application	Structure	External, Goals and Centralized
Soojin and Sungyong (2019)	Cloud	Yes	No	Changes in Context	Reactive	Application	Structure	External, Goals and Decentralized
Park and Song (2015)	IoT Device	Yes	No	Changes in Context	Reactive	Application	Parameters	External, Goals and Centralized
Portocarrero et al. (2016)	Cloud and IoT Device	No	No	Changes in Context and Changes in Technical Resources	Reactive and Proactive	Application	Parameters and Structure	External, Rules/Policies and Decentralized
Rahman et al. (2018)	Cloud and IoT Device	No	No	Changes in Technical Resources	Reactive	Application	Parameters and Structure	External, Utility and Centralized
Rausch, Nastic and Dustdar (2018)	Cloud	No	No	Changes in Technical Resources	Reactive and Proactive	Network	Parameters	External, Rules/Policies and Centralized
Uribarren et al. (2008)	Cloud	No	No	Changes in Context and Changes in User Requirements	Reactive	Application	Parameters and Structure	External, Rules/Policies and Centralized
Peros, Joosen and Hughes (2021)	Cloud	No	No	Changes in Technical Resources	Reactive	Application	Parameters	Internal, Utility and Centralized
Pradeep, Krishnamoorthy and Vasilakos (2021)	Cloud	No	No	Changes in Context	Reactive	Application	Structure	External, Goals and Centralized
Ahmed (2022)	Cloud	No	No	Changes in Technical Resources	Reactive	Middleware	Structure	Internal, Goals and Centralized
Hassan et al. (2023)	Cloud	Yes	No	Changes in Technical Resources	Reactive	Application	Parameters	External, Models and Goals, and Centralized
Jung et al. (2024)	Cloud	No	No	Changes in Context and Changes in Technical Resources	Reactive	Application	Structure	External, Goals, Utility, and Centralized
Hassan et al. (2024)	Cloud	No	No	Changes in Context and Changes in Technical Resources	Reactive	Middleware	Parameters	Internal, Models and Goals and Centralized
MEx	**Cloud and IoT Device**	**Yes**	**No**	**Changes in Context and Changes in Technical Resources**	**Reactive**	**Application and Middleware**	**Parameters and Structure**	**External, Goals, Rules/Policies and Hybrid**

Source: Author

The *Why* refers to the reasons behind the adaptation, categorized into *Changes in Context*, *Changes in Technical Resources*, and *Changes in User Requirements*. *Changes in Context* refer to adaptations made due to environmental changes, such as intermittent network connectivity affecting data throughput. *Changes in Technical Resources* refer to adaptations in response to resource availability of resources, such as software component failure or hardware issues, such as power depletion or low storage memory. *Changes in user requirements* refer to adaptations motivated by changing goals or user preferences, such as adding new functionality or replacing a component. Moving to the *When*, it offers two possible approaches: *Reactive* and *Proactive*. Adaptation occurs after an undesired behavior occurs in the *Reactive* approach, while it anticipates undesired system behavior in the *Proactive* approach.

The *Where* indicates where the adaptation is implemented in the solution, and the possible answers are *Application*, *Middleware*, and *Network Infrastructure*. It is more common that the adaptation occurs in the business logic (*Application*), e.g., adding new functionality or correcting an error. However, it is possible to make changes to the *Network Infrastructure* that refer to changes in communication, e.g., network reconfiguration or even *Middleware* changes to improve itself or the surrounding environment. The *What* indicates what has changed in the solution: Application Operating *Parameters*, which refers to changes in application settings or configurations, e.g., frequency of collection of a physical quantity and size of the communication buffer. Another answer is that Software *Structure* refers to changes in the solution's components, e.g., replacing a software component with a more current one, adding a new element with new functionality, and so on.

The *How* examines how the adaptation logic is implemented. It is essential to mention that each solution has its adaptation logic, and this involves mechanisms for monitoring, analyzing, planning, and executing the adaptations. This sub-dimension has three subdivisions: where the adaptation logic is placed, the adaptation criterion, and the degree of decentralization. Adaptation logic can be integrated with the business logic (*Internal*) or separately (*External*). The adaptation criterion defines the elements used in decision-making on whether or not adaptation is necessary: *Models*, *Rules/Policies*, *Goals*, and *Utility*. *Models* make decisions based on analyzing models that describe the system's real and desired situations, which may be related to objectives, architecture, and environment. *Rules/Policies*-based approaches follow predefined rules or policies that dictate how the system should react in different situations. At the same time, *Goals* define how the system should behave and what goals it should meet, e.g., respond in a maximum of 100 milliseconds. *Utility*-based approaches use a function that assigns a system value to the user based on attribute costs involved in executing the solution, and the objective is to maximize the overall utility of the system. In this case, the adaptation is based on evaluating these values, and the selected adaptation strategy is the one with the most significant utility. Finally, regarding the degree of decentralization of the adaptation logic, it can be implemented in a single component (*Centralized*) for the entire adaptive system, or each subsystem of the adaptive system has its adaptation logic (*Decentralized*) to orchestrate its adaptations, and these can communicate for global purposes. It is also possible that a single adaptation logic has its functionalities implemented and distributed among several subsystems (*Hybrid*).

Table 6.1 shows that most middleware solutions operate in the cloud rather than on the IoT device and are not frameworks. Only three solutions are frameworks (Hassan et al., 2023; Soojin; Sungyong, 2019; Park; Song, 2015), and among them, only one operates on the device. In contrast, MEx is the only solution that operates both in the cloud and on the IoT device. Compared to MEx, these frameworks include adaptation capabilities. However, they execute in the cloud or require application restarts for the adaptation to be applied on the device. In contrast, MEx runs directly on IoT devices, dynamically allowing seamless runtime adaptations for middleware and applications. Additionally, while these frameworks focus on adaptive services or data flow management to meet QoS requirements, MEx performs runtime self-adaptation, adjusting middleware components and application parameters to handle uncertainties and improve system efficiency.

Additionally, Table 6.1 highlights that static middleware solutions (Agostinho et al. (2018), Elkhodr, Shahrestani and Cheung (2016), Joseph et al. (2017), Light (2017), HiveMQ[4]) resemble MEx middleware only by the fact they are IoT middleware systems operating directly on IoT devices. However, they are static, meaning they are not adaptive, which limits their ability to deal with the dynamism and frequent changes of IoT environments. Once deployed, their configurations are limited to their operational capacity, and any update or reconfiguration is only possible with the complete stop of the IoT device, leading to potential disruptions in the system and making IoT applications less adaptable. In contrast, MEx is adaptive, allowing middleware and applications to dynamically adapt and evolve in response to changing conditions or uncertainties, making it more suited to handle IoT environments' dynamic and rapidly changing nature.

At the same time, those middleware solutions with adaptive capabilities share similarities

[4] Available in: https://www.hivemq.com/

with MEx middleware in the following aspects: they run in the Cloud and on the device, supporting adaptations motivated by changes in context or the availability of technical resources. In addition, most solutions enable reactive adaptations of the application by adjusting application parameters or system structures. Finally, some solutions implement external adaptation logic based on goals or rules approaches, which helps decide how and when to adapt. However, these middleware systems often address specific adaptation challenges within application domains, such as mobility management, resource optimization, or interoperability at the application or network layers. In contrast, MEx provides a more generalized and customizable solution. MEx middleware systems perform dynamic adaptations at the application and middleware levels. MEx's middleware supports diverse adaptation mechanisms to handle uncertainties and implement runtime adjustments across various IoT layers, making it particularly suitable for different uncertainties and resource-constrained environments.

Hence, MEx differs from all existing solutions by the fact that (1) it allows multi-layer adaptation running directly on the IoT device and at the middleware level, not just at the application level; (2) it is designed to be easily extensible, making it possible to incorporate new middleware components and new adaptation mechanisms. This flexibility takes advantage of the modular approach of MEx's design, enabling integration of new functionalities and adaptations, and (3) because it is easily customizable, it is possible to build different adaptive middleware architectures and provide different adaptation mechanisms.

6.7 CONCLUDING REMARKS

This chapter presented related works. First, existing works relevant to the development of MEx were discussed and organized into three main groups: middleware frameworks, adaptive middleware, and non-adaptive middleware. Next, other works, particularly energy-saving strategies, were also discussed, contributing to the design of MEx's DCAM mechanisms. Additionally, works on smart water management, especially those related to AquaMOM , were reviewed. Finally, a comparative analysis highlighted the differences between MEx and those IoT middleware solutions.

7 CONCLUSION AND FUTURE WORK

"We ourselves feel that what we are doing is nothing more than a drop in the ocean. But the ocean would be less because of that missing drop."

—Mother Teresa

This chapter presents the conclusions and main contributions of this thesis, along with its significant aspects. Furthermore, it also highlights some existing limitations and outlines future research directions. Finally, the scientific publications resulting from this research are listed.

7.1 CONCLUSION

IoT has attracted the attention of academia and industry due to its applications in smart homes, smart cities, IIoT, smart water management and so on. Advances in IoT devices' processing, storage, and communication capabilities have driven the development of distributed IoT applications. Middleware has become essential in building these distributed applications, leading to active research on IoT middleware.

IoT environments are highly dynamic and susceptible to frequent changes that introduce uncertainties. These uncertainties often arise from dynamic user requirements, environmental conditions, or resource availability fluctuations. Consequently, they can result in application failures or, more seriously, compromise safety, communication stability, or resource availability.

IoT applications are increasingly deployed in challenging environments, including nuclear plants, disaster response scenarios, and semi-arid regions. These environments also introduce significant uncertainties related to sensor reliability, communication, power management, and overall system durability. These environments pose unique challenges, including aging infrastructure, excessive demand, and intensified operating conditions due to climate change. Such conditions amplify uncertainties and make it critical to assess and adjust systems before, during, and after these uncertainties occur to minimize service degradation, maximize coverage, and improve service quality.

In this context, this research hypothesized that a key strategy to face uncertainties in IoT environments is to develop self-adaptive solutions that can adjust their behavior or structure at runtime in response to performance changes, resource availability, or external dynamics. Middleware systems must incorporate adaptive capabilities to evolve and respond dynamically to these changes. This point refers to the research question of this thesis: *How can middleware systems for IoT be designed and implemented with adaptive capabilities as a first-class concept while addressing their fundamental requirements?*

In response, this research proposed, designed and implemented *M*iddleware *Ex*tendify (MEx), a comprehensive self-adaptive solution for developing and executing self-adaptive middleware systems for the IoT. MEx facilitates the implementation of middleware systems through a library of pre-implemented, loosely coupled components and supports the creation of distributed applications in IoT environments. By applying software architecture principles, MEx minimizes developer involvement, only requiring he/she to define a middleware architecture using a simple architecture description language (*p*ADL) and deploy it in the IoT environment.

MEx is designed to run directly on IoT devices, which often have limited resources, such as processing power, memory, connectivity failures and battery life. Additionally, MEx adopts a conceptual model that divides a self-adaptive system into two elements: the managed system, which undergoes adaptation, and the managing system, which executes the adaptations. This model is essential for enabling IoT application adaptation, once it separates the adaptation process, helping to manage the resource constraints of IoT devices.

MEx supports multi-layer adaptation, offering dynamic adaptation (at runtime) at middleware and application levels. It enables the exchange of middleware components and reconfiguring execution parameters for applications and middleware systems. It is essential to highlight that runtime uncertainties require different adaptation mechanisms. Hence, MEx may be extended to deal with as many uncertainties as possible by allowing the inclusion of new custom adaptation mechanisms. It also can switch mechanisms to meet specific adaptation goals. These adaptations aim to enhance middleware and application performance, adjust to environmental changes, fix bugs, and keep the middleware updated.

All adaptation mechanisms in MEx are implemented according to the MAPE-K control loop, often used in self-adaptive systems in general but is applied in MEx in a distributed and decentralized manner. Instances of the Executor (and sometimes the Monitor) run directly on the IoT device, while the Analyzer, Planner and Knowledge Database) are hosted in the managing system. This approach to deploying the MAPE-K helps to save computational and energy resources, which are essential in IoT environments.

In addition, it is worth observing that the adaption logic in MEx differs slightly from traditional MAPE-K implementations. The Executor on the device plays an active role in MEx, requesting the managing system to determine whether an adaptation is needed, while the decision-making process for adaptation takes place outside the device in the managing system. MEx defines the adaptation logic this away because it is the IoT environment, where adaptation needs to have greater control and be infrequent. For example, decision-making can consume a lot of computational resources, and devices can go into deep sleep mode, which leaves them inactive and causes unsubscribes.

Experimental results demonstrated that device-based adaptive IoT middleware is feasible. As expected, adaptations introduce some overhead on application performance. However, they bring benefits by enabling the tunning of IoT applications before (during development) or after deployment (at runtime) on IoT devices. For example, adaptation at the middleware level allowed changing middleware components to improve performance, such as reduced application publishing time despite adaptation overhead. Furthermore, adaptation at the application level enabled energy savings and prolonged battery life by adjusting the application based on context.

7.2 CONTRIBUTIONS

The main contribution of this thesis is Middleware Extendify (MEx), a customizable and extensible solution comprising a framework and an underlying execution environment for designing and implementing self-adaptive middleware systems tailored to IoT. MEx aims to face complexities of self-adaptive systems for IoT. MEx pioneers a comprehensive approach, supporting diverse MAPE-K-based adaptive mechanisms within its execution environment, enabling runtime adaptations to handle uncertainty in IoT applications. These mechanisms are interchangeable, and new ones can be added as needed, motivated by factors such as adaptation goals, changes in developer preferences, and environmental conditions. Thus, an appropriate adaptation mechanism can be selected depending on the developer's needs. Adaptation involves uploading, removing, or configuring middleware components and application parameters directly on IoT devices if necessary. To the best of our knowledge, MEx is the first solution explicitly designed for the development of device-based self-adaptive IoT middleware. Likewise, it is the first to combine customizability, extensibility, and multi-layer adaptivity at both the middleware and application levels, making contributions to the field of IoT middleware.

The contributions resulting from the development of MEx are highlighted as follows:

- **Middleware Framework**: At the core of MEx is its framework, designed to simplify the development of self-adaptive middleware systems for IoT. This simplification is achieved through pre-implemented and loosely coupled middleware components, allowing developers to define only a high-level artifact of the implemented middleware, i.e., the software architecture. MEx also supports adding custom components, enabling developers to implement various self-adaptive middleware configurations by reusing existing components. This modular approach, grounded in software architecture principles, reduces the effort required to build and maintain self-adaptive IoT systems while ensuring flexibility and adaptability.

- **Architecture Description Language**: MEx includes an ADL called *p*ADL, which provides a Python-based approach for defining self-adaptive middleware architectures. This language helps developers describe, deploy, and evolve middleware architectures directly in the execution environment. It serves as a tool for outlining the structure of middleware systems, aiding in the development process. Developers must only define and deploy the self-adaptive middleware architecture in the execution environment.

- **Adaptive Middleware Execution Environment**: MEx provides a runtime environment responsible for executing IoT applications and their self-adaptive middleware systems. This environment manages applications and orchestrates the adaptation process, ensuring smooth operation in dynamic IoT environments. MEx currently includes three adaptation mechanisms: *Evolutive*, *Duty Cycle Adaptation*, and the *Time-based Duty Cycle Adaptation*. The *Evolutive* mechanism allows composite adaptation by enabling continuous updates to the middleware, incorporating new features, fixing bugs, or improving performance. This mechanism is enhanced through component replacement. The other two mechanisms are parametric adaptation and allow the adjustment of application parameters to save energy, which is a critical uncertainty for IoT applications. Adaptation occurs directly on IoT devices. Moreover, MEx provides developers with the flexibility to integrate custom adaptation mechanisms, thus enhancing the ability to meet various adaptation goals and address diverse uncertainties.

By adopting a distributed deployment model, MEx ensures that computationally intensive adaptation decisions occur in a managing system. This Managing System works in a distributed manner, with decision-making happening on a cloud server to offload processing from the device and execute adaptation inside the IoT device. This approach reduces the burden on resource-constrained IoT devices. This approach not only preserves device functionality and ensures the maintenance of IoT applications but also empowers developers to craft their own adaptation mechanisms to address specific uncertainties in their IoT environments.

Considering the contributions of MEx, particularly in dealing with the dynamic nature and uncertainties faced by IoT applications, this thesis presents additional contributions demonstrating the capabilities of MEx.

- **AquaMOM**: AquaMOM represents a practical application of MEx. It is a self-adaptive smart water consumption monitoring system designed to assist humans (e.g., customers) improve water conservation in challenging environments, such as semi-arid regions. Leveraging MEx's capabilities, AquaMOM integrates a low-cost device built with off-the-shelf components, along a web application for user interaction. AquaMOM's adaptive mechanisms ensure reliable water monitoring even in environments characterized by limited connectivity, energy constraints, and variable user needs.

  Beyond its technical achievements, AquaMOM emphasizes social impact by empowering communities to manage water resources more effectively. The AquaMOM's digital feedback promotes behavioral change, encouraging households to conserve water and adopt sustainable practices. Simulation results validate AquaMOM's potential to significantly reduce water consumption in resource-scarce regions, demonstrating its broader applicability to other sustainability-focused IoT applications.

- **IoT Device**: A key innovation of this research is the adaptability of MEx directly on IoT devices with limited resources. Unlike other adaptive solutions that often rely on powerful servers, MEx is optimized for constrained environments, balancing local execution and cloud-based decision-making. A specialized low-cost IoT device was designed and built to showcase this capability. This device, equipped with sensors to measure water levels and energy consumption, can execute MEx, run IoT applications, collect data, connect to the Internet, and automatically adapt to manage water and energy consumption efficiently, ensuring effective monitoring.

These contributions demonstrate how MEx can foster innovative and practical solutions for sustainable resource management, addressing the dynamic challenges of IoT environments, such as those semi-arid regions.

7.3 LIMITATIONS

Although MEx meets the proposed design decisions and fulfills defined functionalities, some limitations need to be addressed:

- **Deal with Limited IoT Resources**: Providing self-adaptation capabilities in IoT is challenging due to inherent resource constraints in IoT environments, such as processing power, memory, storage and energy. Although MEx takes these limitations into account and optimizes its architecture to reduce the performance overhead, it still faces challenges, mainly when uploading components over the network. Addressing these challenges further remains essential to fully meeting the demands of resource-constrained IoT environments.
- **Limited Adaptation Strategies**: Currently, MEx implements three adaptation mechanisms using reactive strategies. These are effective for responding to uncertainties as they occur, but the lack of proactive mechanisms limits the system's ability to anticipate and address uncertainties in advance. Corrective strategies are also missing, which could improve the system's ability to fix issues after they arise.
- **Limited Adaptation Scope**: MEx provides multi-layer adaptation, focusing on application and middleware levels but not extending to the device and communication levels of IoT architecture. This limitation restricts its ability to address uncertainties comprehensively across the IoT stack.
- **High Power Consumption due to Wi-Fi**: One limitation of the IoT device built is its high power consumption caused mainly by Wi-Fi communication. Although the adaptation mechanisms have extended battery life, the device cannot operate using only a battery for extended periods. Alternatives like more energy-efficient communication

technologies (e.g., Bluetooth Low Energy or LoRa) or devices optimized for low power consumption should be explored. Additionally, adopting energy harvesting systems could improve battery life and device autonomy.

- **Need for Testing in Highly Dynamic Scenarios**: Although this thesis mentions highly dynamic applications, they are not fully explored. While MEx performs well in dynamic but relatively stable scenarios like AquaMOM, its capabilities in more dynamic environments—such as vehicle traffic monitoring or real-time smart city applications—have yet to be tested. These scenarios involve rapidly changing conditions, which require the system to adapt more quickly and efficiently. The current adaptation mechanisms of MExmight need further refinement to handle such environments' increased complexity and dynamics. Testing MEx in these more dynamic settings is essential to evaluate its performance and identify any areas that need improvement to handle these environments better.
- **Controlled Experimental Evaluations**: The evaluations and simulations of MEx and AquaMOM were performed in controlled laboratory environments, which, while effective for demonstrating the system's capabilities, do not fully replicate real-world conditions. Testing in real-world environments, such as semi-arid regions with cisterns and other constraints, would confirm the system's robustness and effectiveness. It is essential to mention that some empirical limitations were identified during experiments. For example, inaccuracies in the sensor readings were observed, especially when measuring more considerable distances or when the battery was near depletion. These issues suggest that improving the sensing accuracy may require device modifications or the addition of supplementary sensors to enhance reliability and performance.

7.4 FUTURE WORK

Some future directions are proposed to address these limitations and expand the contributions of MEx:

- **Integration of Advanced Adaptation Mechanisms**: One of the main future works is to focus on implementing proactive and corrective adaptation mechanisms in MEx to complement the existing reactive ones. For example, proactive mechanisms powered by artificial intelligence techniques, such as prediction algorithms (e.g., ARIMA, LSTM, Random Forest), could anticipate uncertainties and optimize performance. Additionally, by using Control Theory, corrective mechanisms could ensure precise and reliable adaptations by providing mathematical guarantees. For instance, controllers could be designed to provide more guarantees that a device's battery never depletes completely, even under high usage, by applying mathematical control methods.

- **Code and Resource Optimization in MEx**: Another future direction should focus on optimizing MEx for resource-constrained IoT environments by reducing adaptation overhead and improving code efficiency. Techniques like static analysis, function inlining, and adaptation mechanisms optimizations can reduce execution time and memory usage. Minimizing dependencies and using profiling tools to identify bottlenecks can further enhance performance in resource-limited devices.
- **Extending Adaptation Scope**: Integrating new adaptation mechanisms will also help expand the adaptation capabilities to include the device and communication layers. By addressing adaptation goals at these levels, MEx can offer more comprehensive solutions to handle uncertainties throughout the IoT stack.
- **Improving Energy Efficiency in AquaMOM**: Efforts are being made to explore alternative communication technologies, such as LoRa and BLE, which could significantly reduce power consumption in AquaMOM. In addition, research is being conducted to integrate energy harvesting techniques, which could further extend device autonomy.
- **Implementing IoT Applications for Highly Dynamic Environments**: As part of future work, it is essential to explore the implementation of MEx in IoT applications that deal with more dynamic environments, such as vehicle traffic monitoring or urban mobility management. These applications present challenges, including high-frequency data processing, frequent changes in message devices, and dynamic location changes. By applying MEx to these use cases, we can assess how well it handles complex, fast-changing conditions and determine if its adaptation mechanisms need further refinement to support these highly dynamic scenarios.
- **Real-World Evaluations**: Finally, as future work, it is essential to include experiments in real-world conditions, such as semi-arid regions with actual cisterns and customer participation, to validate MEx and AquaMOM under practical conditions. These

evaluations will refine the system to address challenges in diverse IoT applications and assess the effectiveness of AquaMOM in its functionality, usability, and user perception. Additionally, addressing sensor limitations by improving device design or integrating supplementary sensors will enhance measurement accuracy and ensure reliable operation.

These future works will make MEx a more robust and versatile solution for developing adaptive IoT middleware systems, advancing the IoT middleware field and enabling more impactful applications.

7.5 PUBLICATIONS

During the development of this research, several papers were published presenting the results obtained in different journals and conferences. Some publications are directly related to the thesis, while others contributed to the overall knowledge construction that supported this research. The articles are listed below, ordered from the most recent to the earliest publications.

- Cavalcanti, D. J. M.; JR., E. B.; Rosa, N.; Oliveira, A.; Hughes, D. **AquaMOM: Adaptive IoT System for Water Monitoring in Challenging Environments**. In: Proceedings of the 10th IEEE International Smart Cities Conference, 2024. (Accepted for publication)
- Cavalcanti, D. J. M.; JR., E. L. B.; Santos, S. C.; Rosa, N. S. **A Hybrid Intervention Applied to IoT Course Using Problem-Based Learning and Maker Culture in the Global South**. Proceedings of the 27th International Conference on Interactive Collaborative Learning, 2024. (Accepted for publication)
- Rosa, N. S.; Cavalcanti, D. J. M. **Exploiting Controllers to Adapt Message-Oriented Middleware**. IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), p. 91-100. DOI: 10.1109/ACSOS61780.2024.000272024, 2024.
- Cavalcanti, D. J. M.; Rosa, N. S. **Customizable and Adaptable Middleware of Things**. International Journal of Communication Systems, v. 37, n. 15, Wiley, DOI: 10.1002/dac.5887, 2024.
- Rosa, N. S.; Cavalcanti, D. J. M. **A Control-Theoretical Approach to Adapt Message Brokers**. International Conference on Advanced Information Networking and Appli-

cations (AINA-2023). Cham, Switzerland: Springer, v. 1. p. 261-273, DOI: 10.1007/978-3-031-29056-5_24, 2023.

- Cavalcanti, D. J. M., Hughes, D., and Rosa, N. **An Adaptive Energy Saving Mechanism for Middleware of Things**. In: 31st International Conference on Software, Telecommunications and Computer Networks (SoftCOM2023), Split, Croatia, 2023, p. 1-6, DOI: 10.23919/SoftCOM58365.2023.10271577.

- Rosa, N. S.; Cavalcanti, D. J. M. **Using Controllers to Adapt Messaging Systems: An Initial Experience**. In: Workshop de Visualização, Evolução e Manutenção de Software, Brasil. (VEM 2022). v. 1, p. 46-50. DOI: https://doi.org/10.5753/vem.2022.226809, 2022.

- Cavalcanti, D. J. M., Carvalho, R., and Rosa, N. **Adaptive Middleware of Things**. In: Proceedings of the 26th IEEE Symposium on Computers and Communications (ISCC '21). Athens, Greece, 2021, pp. 1-6, DOI: 10.1109/ISCC53001.2021.9631408.

- Rosa, N., Cavalcanti, D. J. M., Campos, G. et al. **Adaptive Middleware in Go - A Software Architecture-Based Approach**. J Internet Serv Appl 11, 3, 2020, DOI: https://doi.org/10.1186/s13174-020-00124-5.

- Rosa, N. S., CG. M. M. N., Cavalcanti, D. J. M. **Lightweight Formalisation of Adaptive Middleware**. Journal of Systems Architecture, v. 97, p. 54-64, 2029, DOI: 10.1016/j.sysarc.2018.12.002.

- Rosa, N., Campos, G., and Cavalcanti, D. J. M. **An Adaptive Middleware in Go**. In: Proceedings of the 19th Workshop on Adaptive and Reflexive Middleware (ARM '18), 2018, p. 1–6, DOI: https://doi.org/10.1145/3289175.3289176.

These publications demonstrate the impact and dissemination of the research, contributing to the scientific community and fostering further advancements in the field.

REFERENCES

ABDUL-QAWY, A. S. H.; ALMURISI, N. M. S.; TADISETTY, S. Classification of energy saving techniques for IoT-based heterogeneous wireless nodes. *Procedia Computer Science*, p. 2590–2599, 2020. ISSN 1877-0509. Third International Conference on Computing and Network Communications.

ACHILLEOS, A. P.; GEORGIOU, K.; MARKIDES, C.; KONSTANTINIDIS, A.; PAPADOPOULOS, G. A. Adaptive runtime middleware: Everything as a service. *Computational Collective Intelligence (ICCCI).*, Springer International Publishing, Cham, v. 10448, p. 484–494, 2017.

AGOSTINHO, B. M.; ROTTA, G.; PLENTZ, P. D. M.; DANTAS, M. A. R. Smart comm: A smart home middleware supporting information exchange. *44th Annual Conference of the IEEE Industrial Electronics Society*, IEEE, Washington, DC, USA, p. 4678–4684, 2018. ISSN 1553-572X. Available at: <https://ieeexplore.ieee.org/document/8591251>.

AHMED, N. MPaS: A micro-services based publish/subscribe middleware system model for IoT. In: *2022 5th Conference on Cloud and Internet of Things (CIoT)*. [S.l.: s.n.], 2022. p. 220–225.

AL-FUQAHA, A.; GUIZANI, M.; MOHAMMADI, M.; ALEDHARI, M.; AYYASH, M. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials*, v. 17, n. 4, p. 2347–2376, Fourthquarter 2015. ISSN 1553-877X.

ALAGAR, V.; WAN, K. Understanding and measuring risk due to uncertainties in iot. In: *2019 IEEE International Conference on Smart Internet of Things (SmartIoT)*. [S.l.: s.n.], 2019. p. 484–488.

ALBANY, M.; ALSAHAFI, E.; ALRUWILI, I.; ELKHEDIRI, S. A review: Secure internet of thing system for smart houses. *Procedia Computer Science*, v. 201, p. 437–444, 2022. ISSN 1877-0509. The 13th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 5th International Conference on Emerging Data and Industry 4.0 (EDI40). Available at: <https://www.sciencedirect.com/science/article/pii/S1877050922004707>.

ALFONSO, I.; GARCéS, K.; CASTRO, H.; CABOT, J. Self-adaptive architectures in IoT systems: a systematic literature review. *Journal of Internet Services and Applications*, Springer, USA, v. 14, p. 2471–2488, 12 2021. ISSN 1869-0238. Available at: <https://doi.org/10.1186/s13174-021-00145-8>.

ASLANPOUR, M. S.; GILL, S. S.; TOOSI, A. Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research. *Internet of Things*, v. 12, p. 100273, 08 2020. Available at: <https://www.sciencedirect.com/science/article/pii/S2542660520301062>.

ATZORI, L.; BELLIDO, J.; BOLLA, R.; GENOVESE, G.; IERA, A.; JARA, A.; LOMBARDO, C.; MORABITO, G. Sdn&nfv contribution to iot objects virtualization. *Computer Networks*, v. 149, p. 200–212, 2019. ISSN 1389-1286. Available at: <https://www.sciencedirect.com/science/article/pii/S1389128618312933>.

ATZORI, L.; IERA, A.; MORABITO, G. The internet of things: A survey. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, Elsevier North-Holland, Inc., v. 54, n. 15, p. 2787–2805, 08 2010. ISSN 1389-1286. Available at: <https://doi.org/10.1016/j.comnet.2010.05.010>.

Bandyopadhyay, S.; Sengupta, M.; Maiti, S.; Dutta, S. A survey of middleware for internet of things. *Recent Trends in Wireless and Mobile Networks.*, Springer Berlin Heidelberg, Berlin, Heidelberg, v. 16, p. 288–296, 2011.

BARBOSA, H. Understanding the rapid increase in drought stress and its connections with climate desertification since the early 1990s over the brazilian semi-arid region. *Journal of Arid Environments*, p. 1–19, 2024. ISSN 0140-1963.

BARESI, L.; GHEZZI, C. The disappearing boundary between development-time and run-time. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. New York, NY, USA: Association for Computing Machinery, 2010. (FoSER '10), p. 17–22. ISBN 9781450304276. Available at: <https://doi.org/10.1145/1882362.1882367>.

BERNSTEIN, P. A. Middleware: a model for distributed system services. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 39, n. 2, p. 86–98, Feb. 1996. ISSN 0001-0782. Available at: <https://doi.org/10.1145/230798.230809>.

BISHOP, T. A.; KARNE, R. K. A survey of middleware. In: DEBNATH, N. C. (Ed.). *Proceedings of the ISCA 18th International Conference Computers and Their Applications*. [S.l.]: ISCA, 2003. p. 254–258.

BLAIR, G.; SCHMIDT, D.; TACONET, C. Middleware for Internet distribution in the context of cloud computing and the Internet of Things. *Annals of Telecommunications - annales des télécommunications*, Springer, v. 71, n. 3, p. 87 – 92, Apr. 2016. Available at: <https://hal.science/hal-01298015>.

BORGES, P. V.; TACONET, C.; CHABRIDON, S.; CONAN, D.; CAVALCANTE, E.; BATISTA, T. Taming internet of things application development with the iotvar middleware. *ACM Trans. Internet Technol.*, Association for Computing Machinery, New York, NY, USA, v. 23, n. 2, May 2023. ISSN 1533-5399. Available at: <https://doi.org/10.1145/3586010>.

BRITO, A. D.; LOPES, J. C.; NETA, M. M. S. dos A. Tripé da governança: Poder público, setor privado e a sociedade civil em busca de uma gestão integrada dos recursos hídricos. *Revista Gestão & Sustentabilidade Ambiental*, v. 8, n. 4, p. 506–522, 2019.

BURHAN, M.; REHMAN, R. A.; KHAN, B.; KIM, B.-S. Iot elements, layered architectures and security issues: A comprehensive survey. *Sensors*, v. 18, n. 9, p. 1–37, 2018. ISSN 1424-8220. Available at: <https://www.mdpi.com/1424-8220/18/9/2796>.

CALINESCU, R.; GHEZZI, C.; KWIATKOWSKA, M.; MIRANDOLA, R. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, v. 55, p. 69–77, 09 2012.

CAVALCANTI, D.; CARVALHO, R.; ROSA, N. Adaptive middleware of things. *IEEE*, p. 1–6, 09 2021. ISSN 2642-7389. IEEE Symposium on Computers and Communications (ISCC).

CAVALCANTI, D.; HUGHES, D.; ROSA, N. An adaptive energy saving mechanism for middleware of things. *IEEE*, p. 1–6, 2023. International Conference on Software, Telecommunications and Computer Networks (SoftCOM).

CAVALCANTI, D.; JR., E. B.; ROSA, N.; OLIVEIRA, A.; HUGHES, D. Aquamom: Adaptive iot system for water monitoring in challenging environments. In: *Proceedings of the 10th IEEE International Smart Cities Conference (ISC2)*. Pattaya City, Thailand: IEEE, 2024. p. To appear. Accepted for publication. Smart Cities: Revolution for Mankind.

CAVALCANTI, D.; ROSA, N. Customizable and adaptable middleware of things. *International Journal of Communication Systems*, v. 37, n. 15, p. 1–34, 2024. Available at: <https://onlinelibrary.wiley.com/doi/abs/10.1002/dac.5887>.

CAVALCANTI, D. J. M.; JR., E. L. B.; SANTOS, S. C.; ROSA, N. S. A hybrid intervention applied to iot course using problem-based learning and maker culture in the global south. In: ICL. *Proceedings of the 27th International Conference on Interactive Collaborative Learning (ICL2024)*. Tallinn, Estonia, 2024. p. To appear. Accepted for publication.

Costa, D. G.; Peixoto, J. P. J.; Jesus, T. C.; Portugal, P.; Vasques, F.; Rangel, E.; Peixoto, M. A survey of emergencies management systems in smart cities. *IEEE Access*, v. 10, p. 61843–61872, 2022. ISSN 2169-3536.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T.; BLAIR, G. *Distributed Systems: Concepts and Design*. 5th. ed. USA: Addison-Wesley Publishing Company, 2011. ISBN 0132143011.

CRUZ, M. A. A. da; RODRIGUES, J. J. P. C.; AL-MUHTADI, J.; KOROTAEV, V. V.; ALBUQUERQUE, V. H. C. de. A reference model for internet of things middleware. *IEEE Internet of Things Journal*, IEEE, v. 5, n. 2, p. 871–883, 2018.

DUNKELS, A.; GRONVALL, B.; VOIGT, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. *29th Annual IEEE International Conference on Local Computer Networks.*, IEEE, Tampa, FL, USA, p. 455–462, 11 2004. ISSN 0742-1303.

ELKHODR, M.; SHAHRESTANI, S. A.; CHEUNG, H. A middleware for the internet of things. *International Journal of Computer Networks & Communications (IJCNC)*, v. 8, n. 2, p. 159–178, 03 2016.

ESSAMLALI, I.; NHAILA, H.; El Khaili, M. Advances in machine learning and iot for water quality monitoring: A comprehensive review. *Heliyon*, v. 10, n. 6, p. e27920, 2024. ISSN 2405-8440. Available at: <https://www.sciencedirect.com/science/article/pii/S2405844024039513>.

FAHMIDEH, M.; AHMAD, A.; BEHNAZ, A.; GRUNDY, J.; SUSILO, W. Software engineering for internet of things: The practitioners' perspective. *IEEE Transactions on Software Engineering*, v. 48, n. 8, p. 2857–2878, Aug 2022. ISSN 1939-3520.

FORUM, W. E. *The Global Risks Report 2018, 13th Edition*. [S.l.], 2018. REF: 09012018. Available at: <https://www3.weforum.org/docs/WEF_GRR18_Report.pdf>.

FORUM, W. E. *The Global Risks Report 2023*. 91-93 route de la Capite, CH-1223 Cologny/Geneva, Switzerland, 2023. Published in January 2023. Available at: <https://www.weforum.org/publications/global-risks-report-2023/>.

FRANçA, F. M. C.; OLIVEIRA, J. B. de; ALVES, J. J.; FONTENELE, F. C. B.; FIGUEIREDO, A. Z. Q. *Slab Cistern: Construction, Use, and Maintenance*. [S.l.], 2010. (in Portuguese).

FUXMAN, A. D. *A Survey of Architecture Description Languages*. Toronto, 1999. Technical Report CSRG-407.

GARLAN, D.; SCHMERL, B.; CHENG, S.-W. Software architecture-based self-adaptation. *Autonomic Computing and Networking.*, Springer US, Boston, MA, p. 31–55, 2009. Available at: <https://doi.org/10.1007/978-0-387-89828-5_2>.

GARLAN, D.; SCHMERL, B.; CHENG, S.-W. Software architecture-based self-adaptation. In: _____. [S.l.]: Springer, 2009. p. 31–55. ISBN 978-0-387-89827-8.

GARLAN, D.; SHAW, M. An introduction to software architecture. *Advances in Software Engineering & Knowledge Engineering*, New Jersey, NY, v. 2, p. 1–39, 1993.

GETIRANA, A.; LIBONATI, R.; CATALDI, M. Brazil is in water crisis — it needs a drought plan. *Nature*, v. 600, p. 218–220, 12 2021.

GOEL, S.; SHARDA, H.; TANIAR, D. Message-oriented-middleware in a distributed environment. *Innovative Internet Community Systems. IICS 2003*, Springer Berlin Heidelberg, Berlin, Heidelberg, vol 2877, p. 93–103, 2003.

GONDIM, J.; FIOREZE, A. P.; ALVES, R. F. F.; SOUZA, W. G. d. A seca atual no semiárido nordestino–impactos sobre os recursos hídricos. *Parcerias Estratégicas*, v. 22, n. 44, p. 277–300, 2017.

GUBBI, J.; BUYYA, R.; MARUSIC, S.; PALANISWAMI, M. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, v. 29, n. 7, p. 1645–1660, 2013. ISSN 0167-739X. Available at: <https://www.sciencedirect.com/science/article/pii/S0167739X13000241>.

HAN, Q.; MEHROTRA, S.; VENKATASUBRAMANIAN, N. Aquaeis: Middleware support for event identification in community water infrastructures. *Proceedings of the 20th International Middleware Conference*, Association for Computing Machinery, New York, NY, USA, p. 293–305, 2019. Available at: <https://doi.org/10.1145/3361525.3361554>.

HASSAN, H.; MAZLAN, M.; IBRAHIM, T.; KAMBAS, M. Iot system: Water level monitoring for flood management. *IOP Conference Series: Materials Science and Engineering*, v. 917, p. 012037, 09 2020.

HASSAN, H. H.; BOULOUKAKIS, G.; KATTEPUR, A.; CONAN, D.; BELAïD, D. Planiot: A framework for adaptive data flow management in iot-enhanced spaces. In: *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. [S.l.: s.n.], 2023. p. 157–168. ISSN 2157-2321.

HASSAN, H. H.; BOULOUKAKIS, G.; SCALZOTTO, L.; KHALED, N.; CONAN, D.; KATTEPUR, A.; BELAïD, D. A message broker architecture for adaptive data exchange in the iot. In: *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*. [S.l.: s.n.], 2024. p. 151–158. ISSN 2768-4288.

HEZAVEHI, S. M.; WEYNS, D.; AVGERIOU, P.; CALINESCU, R.; MIRANDOLA, R.; PEREZ-PALACIN, D. Uncertainty in self-adaptive systems: A research community perspective. *ACM Trans. Auton. Adapt. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 15, n. 4, dec 2021. ISSN 1556-4665. Available at: <https://doi.org/10.1145/3487921>.

ITU-T Study Group 20. *Overview of the Internet of things*. [S.l.], 2012.

JAIN, R. *The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling*. New York, United State of America: John Wiley & Sons, 1991. 720 pages p. ISBN 0471503361.

JOSEPH, T.; JENU, R.; ASSIS, A.; KUMAR, V.; SASI, P.; ALEXANDER, G. Iot middleware for smart city: (an integrated and centrally managed iot middleware for smart city). *2017 IEEE Region 10 Symposium (TENSYMP).*, IEEE, Cochin, India, p. 1–5, 07 2017.

JUNG, K.; MITRA, G.; GOPALAKRISHNAN, S.; PATTABIRAMAN, K. Immunoplane: Middleware for providing adaptivity to distributed internet-of-things applications. In: *2024 IEEE/ACM Ninth International Conference on Internet-of-Things Design and Implementation (IoTDI)*. [S.l.: s.n.], 2024. p. 13–24.

KANT, K.; JOLFAEI, A.; MOESSNER, K. IoT systems for extreme environments. *IEEE Internet of Things Journal*, v. 11, n. 3, p. 3671–3675, Feb 2024. ISSN 2327-4662.

KARIE, N. M.; SAHRI, N. M.; HASKELL-DOWLAND, P. IoT threat detection advances, challenges and future directions. In: *2020 Workshop on Emerging Technologies for Security in IoT (ETSecIoT)*. [S.l.: s.n.], 2020. p. 22–29.

KASSAB, W.; DARABKH, K. A–z survey of Internet of Things: Architectures, protocols, applications, recent advances, future directions and recommendations. *Journal of Network and Computer Applications*, v. 163, p. 102663, 04 2020.

KAVRE, M.; GADEKAR, A.; GADHADE, Y. Internet of things (iot): A survey. In: *2019 IEEE Pune Section International Conference (PuneCon)*. [S.l.: s.n.], 2019. p. 1–6.

KEPHART, J.; CHESS, D. The vision of autonomic computing. *Computer*, v. 36, n. 1, p. 41–50, Jan 2003. ISSN 1558-0814.

KHAN, R.; KHAN, S. U.; ZAHEER, R.; KHAN, S. Future internet: The internet of things architecture, possible applications and key challenges. In: *2012 10th International Conference on Frontiers of Information Technology*. [S.l.: s.n.], 2012. p. 257–260.

KIM, G.; KANG, J.-G.; RIM, M. Dynamic duty-cycle mac protocol for iot environments and wireless sensor networks. *Energies*, v. 12, n. 21, p. 1–13, 10 2019. ISSN 1996-1073.

KRAMER, J.; MAGEE, J. Self-managed systems: an architectural challenge. In: *Future of Software Engineering (FOSE '07)*. [S.l.: s.n.], 2007. p. 259–268.

KRUPITZER, C.; ROTH, F. M.; VANSYCKEL, S.; SCHIELE, G.; BECKER, C. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, v. 17, p. 184–206, 2015. ISSN 1574-1192. 10 years of Pervasive Computing' In Honor of Chatschik Bisdikian. Available at: <https://www.sciencedirect.com/science/article/pii/S157411921400162X>.

KUMAR, J.; GUPTA, R.; SHARMA, S.; CHAKRABARTI, T.; CHAKRABARTI, P.; MARGALA, M. Iot-enabled advanced water quality monitoring system for pond management and environmental conservation. *IEEE Access*, v. 12, p. 58156–58167, 2024. ISSN 2169-3536.

KUMAR, S.; YADAV, S.; M, Y. H.; SALVI, S. An iot-based smart water microgrid and smart water tank management system. In: ______. [S.l.: s.n.], 2019. p. 417–431. ISBN 978-981-13-6000-8.

LACERDA, D.; DRESCH, A.; PROENçA, A.; JúNIOR, J. A. V. A. Design science research: A research method to production engineering. *Gestão Produção*, v. 20, p. 741–761, 12 2012.

LAKSHMIKANTHA, V.; HIRIYANNAGOWDA, A.; MANJUNATH, A.; PATTED, A.; BASAVAIAH, J.; A, A. A. Iot based smart water quality monitoring system. *Global Transitions Proceedings*, v. 2, 08 2021.

LEE, K.-F.; NG, Z.-N.; TAN, K.-B.; BALACHANDRAN, R.; CHONG, A. S.-I.; CHAN, K.-Y. Artificial intelligence-integrated water level monitoring system for flood detection enhancement. *International Journal of Intelligent Systems and Applications in Engineering*, v. 12, n. 19s, p. 336–340, Mar. 2024. Available at: <https://ijisae.org/index.php/IJISAE/article/view/5071>.

LEMOS, R. de; GIESE, H.; MÜLLER, H. A.; SHAW, M.; ANDERSSON, J.; LITOIU, M.; SCHMERL, B.; TAMURA, G.; VILLEGAS, N. M.; VOGEL, T.; WEYNS, D.; BARESI, L.; BECKER, B.; BENCOMO, N.; BRUN, Y.; CUKIC, B.; DESMARAIS, R.; DUSTDAR, S.; ENGELS, G.; GEIHS, K.; GÖSCHKA, K. M.; GORLA, A.; GRASSI, V.; INVERARDI, P.; KARSAI, G.; KRAMER, J.; LOPES, A.; MAGEE, J.; MALEK, S.; MANKOVSKII, S.; MIRANDOLA, R.; MYLOPOULOS, J.; NIERSTRASZ, O.; PEZZÈ, M.; PREHOFER, C.; SCHÄFER, W.; SCHLICHTING, R.; SMITH, D. B.; SOUSA, J. P.; TAHVILDARI, L.; WONG, K.; WUTTKE, J. Software engineering for self-adaptive systems: A second research roadmap. In: ______. *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 1–32. ISBN 978-3-642-35813-5. Available at: <https://doi.org/10.1007/978-3-642-35813-5_1>.

LIGHT, R. Mosquitto: server and client implementation of the mqtt protocol. *The Journal of Open Source Software*, v. 2, p. 1–2, 05 2017.

LIU, A.; MUKHEIBIR, P. Digital metering feedback and changes in water consumption – a review. *Resources, Conservation and Recycling*, v. 134, p. 136–148, 07 2018.

MAGRUK, A. The most important aspects of uncertainty in the internet of things field – context of smart buildings. *Procedia Engineering*, v. 122, p. 220–227, 2015. ISSN 1877-7058. Innovative solutions in Construction Engineering and Management. Flexible Approach. Available at: <https://www.sciencedirect.com/science/article/pii/S1877705815031173>.

MAHAMUNI, C. V. Exploring iot-applications: A survey of recent progress, challenges, and impact of ai, blockchain, and disruptive technologies. In: *2023 7th International Conference on Electronics, Communication and Aerospace Technology (ICECA)*. [S.l.: s.n.], 2023. p. 1324–1331.

MALCHE, T.; MAHESHWARY, P. Internet of things (iot) based water level monitoring system for smart village. In: . [S.l.: s.n.], 2017. ISBN 9789811027499.

MARZALL, V.; NASCIMENTO, N. Determination of residential potable water consumption profile supported by smart metering technology. *Revista de Gestão de Água da América Latina*, p. 3, 2023. (in Portuguese).

MATHUR, S.; KALLA, A.; GüR, G.; BOHRA, M. K.; LIYANAGE, M. A survey on role of blockchain for iot: Applications and technical aspects. *Computer Networks*, v. 227, p. 109726, 2023. ISSN 1389-1286. Available at: <https://www.sciencedirect.com/science/article/pii/S1389128623001718>.

MCKENZIE, F.; PETTY, M.; XU, Q. Usefulness of software architecture description languages for modeling and analysis of federates and federation architectures. *Simulation*, v. 80, p. 559–576, 11 2004.

MCKINLEY, P.; SADJADI, S.; KASTEN, E.; CHENG, B. Composing adaptive software. *Computer*, v. 37, n. 7, p. 56–64, 2004. ISSN 0018-9162.

MEDEIROS, R.; FERNANDES, S.; QUEIROZ, P. Middleware for the internet of things: a systematic literature review. *JUCS - Journal of Universal Computer Science*, v. 28, p. 54–79, 01 2022.

MEDVIDOVIC, N.; TAYLOR, R. N. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, IEEE, v. 26, n. 01, p. 70–93, 2000. ISSN 1939-3520. Available at: <https://ieeexplore.ieee.org/document/825767>.

MEDVIDOVIC, N.; TAYLOR, R. N. Software architecture: foundations, theory, and practice. In: *2010 32nd International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, 2010. v. 2, p. 471–472. ISSN 0270-5257. Available at: <https://doi.ieeecomputersociety.org/10.1145/1810295.1810435>.

MENASCE, D. Mom vs. rpc: communication models for distributed applications. *IEEE Internet Computing*, v. 9, n. 2, p. 90–93, March 2005. ISSN 1941-0131.

MOGHADDAM, M. T.; RUTTEN, E.; GIRAUD, G. Protocol for a systematic literature review on adaptative middleware support for IoT and cps. Working paper or preprint. 2020. Available at: <https://inria.hal.science/hal-02948347>.

MOHALIK, S. K.; NARENDRA, N. C.; BADRINATH, R.; JAYARAMAN, M. B.; PADALA, C. Dynamic semantic interoperability of control in iot-based systems: Need for adaptive middleware. *IEEE 3rd World Forum on Internet of Things (WF-IoT).*, IEEE, p. 199–203, 2016.

MORENO, G. A.; CáMARA JAVIER ANDGARLAN, D.; SCHMERL, B. Proactive self-adaptation under uncertainty: A probabilistic model checking approach. *In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Association for Computing Machinery, New York, NY, USA, p. 1–12, 08 2015. Available at: <https://doi.org/10.1145/2786805.2786853>.

MUCCINI, H.; MOGHADDAM, M. T. A cyber-physical space operational approach for crowd evacuation handling. In: *International Workshop on Software Engineering for Resilient Systems*. [S.l.: s.n.], 2017. p. 81–95. ISBN 978-3-319-65947-3.

MUCCINI, H.; SHARAF, M.; WEYNS, D. Self-adaptation for cyber-physical systems: A systematic literature review. In: *2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. [S.l.: s.n.], 2016. p. 75–81.

MUCCINI, H.; SPALAZZESE, R.; MOGHADDAM, M. T.; SHARAF, M. Self-adaptive IoT architectures: an emergency handling case study. In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. New York, NY, USA: Association for Computing Machinery, 2018. (ECSA '18). ISBN 9781450364836. Available at: <https://doi.org/10.1145/3241403.3241424>.

MUKHOPADHYAY, A.; SREENADH, M.; ANOOP, A. ehealth applications: A comprehensive approach. In: *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*. [S.l.: s.n.], 2020. p. 1–6.

MUKTA, M.; ISLAM, S.; BARMAN, S. D.; REZA, A. W.; KHAN, M. S. H. Iot based smart water quality monitoring system. In: *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS)*. [S.l.: s.n.], 2019. p. 669–673.

MUNIR, D.; SHAH, S. T.; MUGHAL, D. M.; PARK, K. H.; CHUNG, M. Y. Duty cycle optimizing for wifi-based IoT networks with energy harvesting. *IMCOM '18: The 12th International Conference on Ubiquitous Information Management and Communication.*, p. 1–6, 2018.

NIžETIć, S.; ŠOLIć, P.; López-de-Ipiña González-de-Artaza, D.; PATRONO, L. Internet of things (iot): Opportunities, issues and challenges towards a smart and sustainable future. *Journal of Cleaner Production*, v. 274, p. 122877, 2020. ISSN 0959-6526. Available at: <https://www.sciencedirect.com/science/article/pii/S095965262032922X>.

NIžETIć, S.; ŠOLIć, P.; López-de-Ipiña González-de-Artaza, D.; PATRONO, L. Internet of things (iot): Opportunities, issues and challenges towards a smart and sustainable future. *Journal of Cleaner Production*, v. 274, p. 122877, 2020. ISSN 0959-6526. Available at: <https://www.sciencedirect.com/science/article/pii/S095965262032922X>.

NUNDLOLL, V.; PORTER, B.; BLAIR, G. S.; EMMETT, B.; COSBY, J.; JONES, D. L.; CHADWICK, D.; WINTERBOURN, B.; BEATTIE, P.; DEAN, G.; SHAW, R.; SHELLEY, W.; BROWN, M.; ULLAH, I. The design and deployment of an end-to-end IoT infrastructure for the natural environment. *Future Internet*, v. 11, n. 6, 2019. ISSN 1999-5903. Available at: <https://www.mdpi.com/1999-5903/11/6/129>.

OSGi Working Group. *OSGi Core Specification, Release 8, Version 2.0*. [S.l.], 2024. Last accessed in November 2024. Available at: <https://osgi.github.io/osgi/core/index.html>.

PARK, S.; SONG, J. Self-adaptive middleware framework for internet of things. *IEEE 4th Global Conference on Consumer Electronics (GCCE).*, IEEE, Osaka, Japan, p. 81–82, 2015.

PATEL, P.; ALI, M. I.; SHETH, A. On using the intelligent edge for iot analytics. *IEEE Intelligent Systems*, v. 32, p. 64–69, 09 2017.

PEKARIC, I.; GRONER, R.; WITTE, T.; ADIGUN, J.; RASCHKE, A.; FELDERER, M.; TICHY, M. A systematic review on security and safety of self-adaptive systems. *Journal of Systems and Software*, v. 203, p. 111716, 2023. ISSN 0164-1212. Available at: <https://www.sciencedirect.com/science/article/pii/S0164121223001115>.

PEROS, S.; JOOSEN, W.; HUGHES, D. Ermis: a middleware for bridging data collection and data processing in iot streaming applications. In: *2021 17th International Conference on Distributed Computing in Sensor Systems (DCOSS)*. [S.l.: s.n.], 2021. p. 259–266. ISSN 2325-2944.

PETER, O.; PRADHAN, A.; MBOHWA, C. Industrial internet of things (iiot): opportunities, challenges, and requirements in manufacturing businesses in emerging economies. *Procedia Computer Science*, v. 217, p. 856–865, 2023. ISSN 1877-0509. 4th International Conference on Industry 4.0 and Smart Manufacturing. Available at: <https://www.sciencedirect.com/science/article/pii/S1877050922023602>.

PORTOCARRERO, J. M. T.; DELICATO, F. C.; PIRES, P. F.; RODRIGUES, T. C.; BATISTA, T. V. Samson: Self-adaptive middleware for wireless sensor networks. *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, Association for Computing Machinery, New York, NY, USA, p. 1315–1322, 2016. Available at: <https://doi.org/10.1145/2851613.2851766>.

PRADEEP, P.; KRISHNAMOORTHY, S.; VASILAKOS, A. A holistic approach to a context-aware iot ecosystem with adaptive ubiquitous middleware. *Pervasive and Mobile Computing*, v. 72, p. 101342, 02 2021.

PUJAR, P.; KENCHANNAVAR, H.; KULKARNI, R.; KULKARNI, U. Real-time water quality monitoring through internet of things and anova-based analysis: a case study on river krishna. *Applied Water Science*, v. 10, 01 2020.

QADRI, Y. A.; NAUMAN, A.; ZIKRIA, Y. B.; VASILAKOS, A. V.; KIM, S. W. The future of healthcare internet of things: A survey of emerging technologies. *IEEE Communications Surveys & Tutorials*, v. 22, n. 2, p. 1121–1167, Secondquarter 2020. ISSN 1553-877X.

RAHMAN, M.; RAHMAN, A.; HONG, H.-J.; PAN, L.; UDDIN, M. Y. S.; VENKATASUBRAMANIAN, N.; HSU, C.-H. An adaptive iot platform on budgeted 3g data plans. *Journal of Systems Architecture*, v. 97, p. 65–76, 2018.

RAMACHANDRAN, G.; PROENçA, J.; DANIELS, W.; PICKAVET, M.; STAESSENS, D.; HUYGENS, C.; JOOSEN, W.; HUGHES, D. Hitch hiker 2.0: a binding model with flexible data aggregation for the Internet-of-Things. *Journal of Internet Services and Applications*, v. 7, p. 15 pages, 04 2016.

RAMACHANDRAN, G. S.; MATTHYS, N.; DANIELS, W.; JOOSEN, W.; HUGHES, D. Building dynamic and dependable component-based Internet-of-Things applications with dawn. *19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE).*, IEEE, Venice, Italy, p. 97–106, 2016.

RANIERI, C. M.; FOLETTO, A. V.; GARCIA, R. D.; MATOS, S. N.; MEDINA, M. M.; MARCOLINO, L. S.; UEYAMA, J. Water level identification with laser sensors, inertial units, and machine learning. *Engineering Applications of Artificial Intelligence*, v. 127, p. 107235, 2024. ISSN 0952-1976. Available at: <https://www.sciencedirect.com/science/article/pii/S0952197623014197>.

RAUSCH, T.; DUSTDAR, S.; RANJAN, R. Osmotic message-oriented middleware for the Internet of Things. *IEEE Cloud Computing*, v. 5, p. 17–25, 03 2018.

RAUSCH, T.; NASTIC, S.; DUSTDAR, S. Emma: Distributed qos-aware mqtt middleware for edge computing applications. *2018 IEEE International Conference on Cloud Engineering (IC2E).*, IEEE, Orlando, FL, USA, p. 191–197, 4 2018.

RAZZAQUE, M. A.; MILOJEVIC-JEVRIC, M.; PALADE, A.; CLARKE, S. Middleware for Internet of Things: A survey. *IEEE Internet of Things Journal*, v. 3, n. 1, p. 70–95, 2016. ISSN 2327-4662.

RODRIGUEZ, R. del G.; PRUSKI, F.; SINGH, V. Cistern project for domestic water use in semi-arid regions. *International Journal of Engineering Research and Technology*, v. 5, p. 695–702, 03 2016.

ROSA, N.; CAMPOS, G.; CAVALCANTI, D. Using software architecture principles and lightweight formalisation to build adaptive middleware. *Proceedings of the 16th Workshop on Adaptive and Reflective Middleware.*, Association for Computing Machinery, New York, NY, USA, p. 1–7, 2017. Available at: <https://doi.org/10.1145/3152881.3152882>.

ROSA, N.; CAVALCANTI, D.; CAMPOS, G.; SILVA, A. Adaptive middleware in go - a software architecture-based approach. *Journal of Internet Services and Applications*, v. 11, p. 1–23, 12 2020. Available at: <https://jisajournal.springeropen.com/articles/10.1186/s13174-020-00124-5>.

SALEHIE, M.; TAHVILDARI, L. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, Association for Computing Machinery, New York, NY, USA, v. 4, n. 2, p. 1–42, 5 2009. ISSN 1556-4665.

SARITHA, G.; ISHWARYA, R.; SARAVANAN, T.; SUDARSHANA, P. A. S.; SOWMIYA, S. Water quality monitoring system using iot. In: *2023 Eighth International Conference on Science Technology Engineering and Mathematics (ICONSTEM)*. [S.l.: s.n.], 2023. p. 1–5.

SCHMIDT, D.; BUSCHMANN, F. Patterns, frameworks, and middleware: Their synergistic relationships. *25th International Conference on Software Engineering.*, IEEE, p. 694–704, 05 2003. ISSN 0270-5257.

SETHI, P.; SARANGI, S. Internet of things: Architectures, protocols, and applications. *Journal of Electrical and Computer Engineering*, v. 2017, p. 1–25, 01 2017.

SHELTAMI, T.; AL-ROUBAIEY, A.; MAHMOUD, A. A survey on developing publish/subscribe middleware over wireless sensor/actuator networks. *Wireless Networks*, v. 22, p. 2049–2070, 08 2016.

SINGH, M.; AHMED, S. IoT based smart water management systems: A systematic review. *Materials Today: Proceedings*, v. 46, p. 5211–5218, 2021. ISSN 2214-7853. International Conference on Innovations in Clean Energy Technologies (ICET2020). Available at: <https://www.sciencedirect.com/science/article/pii/S2214785320364701>.

SINREICH, D. *An Architectural Blueprint for Autonomic Computing*. Hawthorne, NY 10532, United States of America, 2005. Technical Report. Available at: <https://api.semanticscholar.org/CorpusID:16909837>.

SISINNI, E.; SAIFULLAH, A.; HAN, S.; JENNEHAG, U.; GIDLUND, M. Industrial Internet of Things: Challenges, opportunities, and directions. *IEEE Transactions on Industrial Informatics*, IEEE, v. 14, n. 11, p. 4724–4734, 2018. ISSN 1941-0050.

SOOJIN, P.; SUNGYONG, P. A cloud-based middleware for self-adaptive IoT-collaboration services. *Sensors (Basel, Switzerland)*, v. 19, p. 19 pages, 10 2019.

SULISTYOWATI, R.; SUJONO, H. A.; MUSTHOFA, A. K. Design and field test equipment of river water level detection based on ultrasonic sensor and sms gateway as flood early warning. *AIP Conference Proceedings*, v. 1855, p. 1–9, jun 2017.

SUNNY, A. I.; ZHAO, A.; LI, L.; SAKILIBA, S. K. Low-cost IoT-based sensor system: A case study on harsh environmental monitoring. *Sensors*, v. 21, n. 1, 2021. ISSN 1424-8220.

TACONET, C.; BATISTA, T.; BORGES, P.; BOULOUKAKIS, G.; CAVALCANTE, E.; CHABRIDON, S.; CONAN, D.; DESPRATS, T.; MUÑANTE, D. Middleware supporting pis: Requirements, solutions, and challenges. In: ______. *The Evolution of Pervasive Information Systems*. Cham: Springer International Publishing, 2023. p. 65–97. ISBN 978-3-031-18176-4. Available at: <https://doi.org/10.1007/978-3-031-18176-4_4>.

TANENBAUM, A. S.; STEEN, M. v. *Distributed Systems: Principles and Paradigms (2nd Edition)*. USA: Prentice-Hall, Inc., 2006. ISBN 0132392275.

TARKOMA, S. *Publish / Subscribe Systems: Design and Principles*. 1st. ed. United Kingdom: John Wiley & Sons, 2012. 352 pages p. (Wiley Series in Communications Networking & Distributed Systems.). ISBN 9781119951544.

THIRUMARAI, C.; R.S, S.; M, M.; P, G. Iot-enabled flood monitoring system for enhanced dam surveillance and risk mitigation. *International Research Journal of Multidisciplinary Technovation*, p. 144–153, 05 2024.

TUBIO, I.; ALLOSO, N.; RABAGO, M.; LACSA, J.; SUDARIA, P. R. A.; GUMONAN, K. M. V.; LACSA, M.; II, N. T.; RABAGO, J. M. Aquamag: Smart water quality monitoring through internet of things. *International Journal of Science, Technology, Engineering and Mathematics*, v. 3, p. 1–18, 03 2023.

URIBARREN, A.; PARRA, J.; IGLESIAS, R.; URIBE, J. P.; IPIñA, D. López-de. A middleware platform for application configuration, adaptation and interoperability. *IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, p. 162–167, 10 2008.

VENANZI, R.; FOSCHINI, L.; BELLAVISTA, P.; KANTARCI, B.; STEFANELLI, C. Fog-driven context-aware architecture for node discovery and energy saving strategy for internet of things environments. *IEEE Access*, IEEE, v. 7, p. 134173–134186, 2019.

VOELTER, M.; KIRCHER, M.; ZDUN, U. *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. Hoboken, NJ, USA: John Wiley & Sons, 2004. (Wiley Series in Software Design Patterns). Available at: <http://eprints.cs.univie.ac.at/2380/>.

VöGLER, M.; SCHLEICHER, J. M.; INZINGER, C.; DUSTDAR, S. A scalable framework for provisioning large-scale iot deployments. *ACM Trans. Internet Technol.*, Association for Computing Machinery, New York, NY, USA, v. 16, n. 2, mar 2016. ISSN 1533-5399. Available at: <https://doi.org/10.1145/2850416>.

WEYNS, D. Engineering self-adaptive software systems – an organized tour. *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W).*, IEEE, p. 1–44, 09 2018.

WEYNS, D. Software engineering of self-adaptive systems. *In: Handbook of Software Engineering.*, Springer International Publishing, Cham, p. 399–443, 5 2020. Available at: <https://doi.org/10.1007/978-3-030-00262-6_11>.

WEYNS, D. *An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective*. United Kingdom: John Wiley & Sons, 2021. 288 pages p. ISBN 9781119574910. Available at: <https://ieeexplore.ieee.org/servlet/opac?bknumber=9261286>.

WEYNS, D.; RAMACHANDRAN, G.; SINGH, R. Self-managing internet of things. *SOFSEM 2018: Theory and Practice of Computer Science: 44th International Conference on Current Trends in Theory and Practice of Computer Science.*, Springer, p. 67–84, 01 2018.

WEYNS, D.; SCHMERL, B. et al. On patterns for decentralized control in self-adaptive systems. In: ______. *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 76–107. ISBN 978-3-642-35813-5.

WONG, T.; WAGNER, M.; TREUDE, C. Self-adaptive systems: A systematic literature review across categories and domains. *Information and Software Technology*, v. 148, p. 106934, 2022. ISSN 0950-5849. Available at: <https://www.sciencedirect.com/science/article/pii/S0950584922000854>.

WU, M.; LU, T.-J.; LING, F.-Y.; SUN, J.; DU, H.-Y. Research on the architecture of Internet of Things. In: *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*. [S.l.: s.n.], 2010. v. 5, p. V5–484–V5–487.

XU, L.; CHEN, L.; GAO, Z.; FAN, X.; SUH, T.; SHI, W. Diota: Decentralized-ledger-based framework for data authenticity protection in IoT systems. *IEEE Network*, v. 34, n. 1, p. 38–46, January 2020. ISSN 1558-156X.

YANG, Z.; YUE, Y.; YANG, Y.; PENG, Y.; WANG, X.; LIU, W. Study and application on the architecture and key technologies for iot. In: *2011 International Conference on Multimedia Technology*. [S.l.: s.n.], 2011. p. 747–751.

ZDUN, U.; KIRCHER, M.; VöLTER, M. Remoting patterns: Design reuse of distributed object middleware solutions. *Internet Computing, IEEE*, v. 8, p. 60 – 68, 12 2004.

ZHANG, J.; MA, M.; WANG, P.; SUN, X. dong. Middleware for the Internet of Things: A survey on requirements, enabling technologies, and solutions. *Journal of Systems Architecture*, v. 117, p. 102098, 2021. ISSN 1383-7621. Available at: <https://www.sciencedirect.com/science/article/pii/S1383762121000795>.

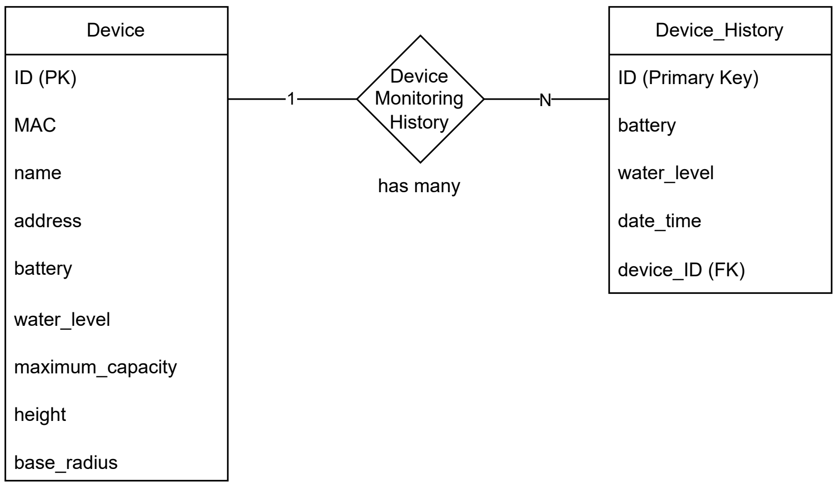
APPENDIX A – BNF SPECIFICATION FOR *P*ADL LANGUAGE

```
<padl> ::= "padl" "=" "{" <components> <attachments> <adaptability> <configuration> "}"


<components> ::= "\"components\":" "{" <component> {"," <component>} "}"

<component> ::= <string> ":" <string>


<attachments> ::= "\"attachments\":" "{" <attachment> {"," <attachment>} "}"

<attachment> ::= <string> ":" <string>


<adaptability> ::= "\"adaptability\":" "{" <adaptability_type> "," <timeout> "}"

<adaptability_type> ::= "\"type\":" <string>

<timeout> ::= "\"timeout\":" <integer>


<configuration> ::= "\"configuration\":" "{" <start> "," <other_configs> "}"

<start> ::= "\"start\":" <string>

<other_configs> ::= "\"other_configs\":" "{" { <config> } "}"

<config> ::= <string> ":" <value>


<string> ::= "\"" {character} "\""

<integer> ::= digit {digit}

<value> ::= <string> | <integer> | <object>

<object> ::= "{" <config> {"," <config>} "}"
```

APPENDIX B – AQUAMOM DATABASE

The database schema consists of two main tables: *Device* and *Device_History*, as shown in Figure B.1.

Figure B.1 – Entity-Relationship Diagram of AquaMOM

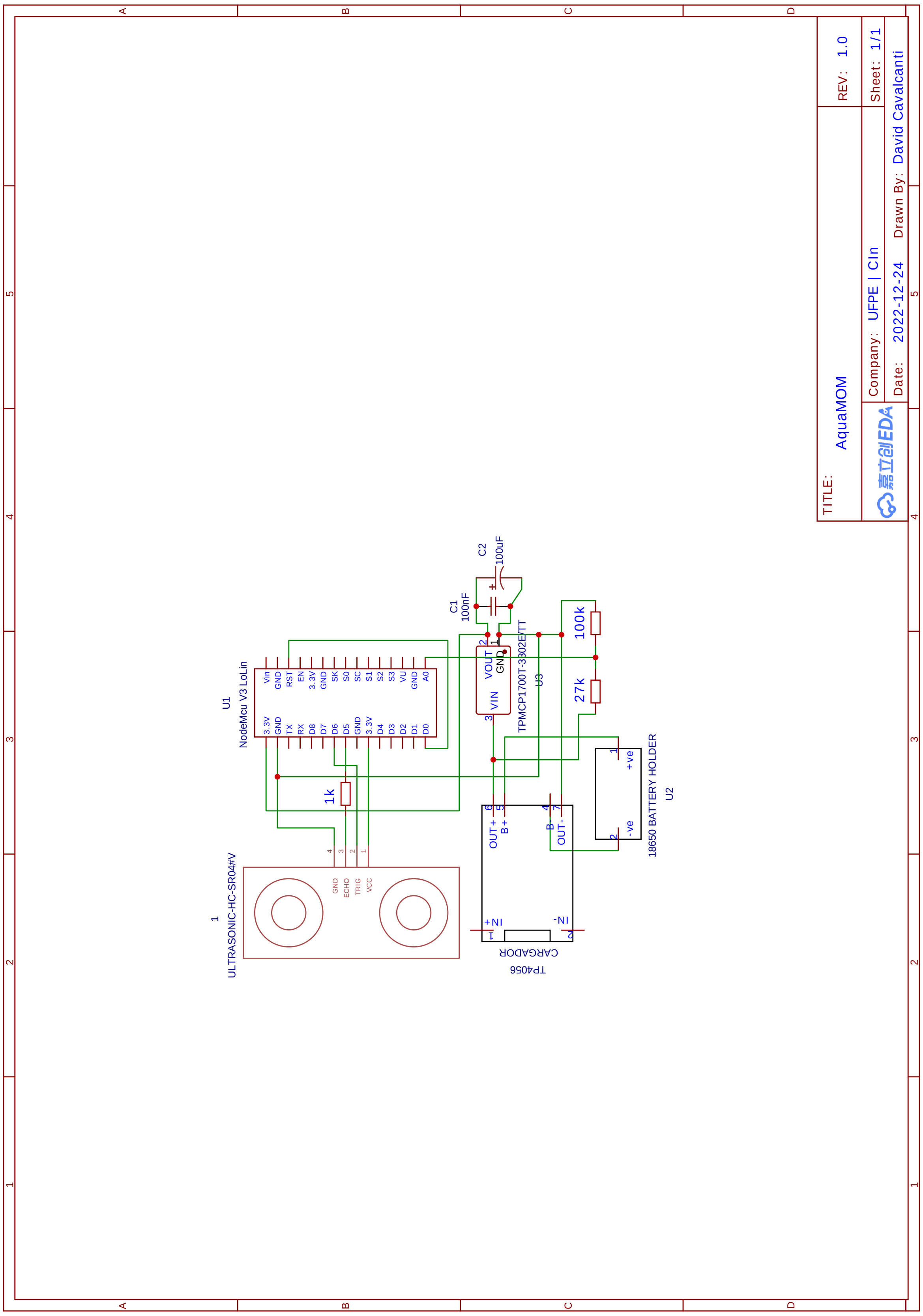


The *Device* table is the primary entity, responsible for storing information about each cistern and AquaMOM device. It includes attributes such as a unique identifier (ID), the MAC address, device name, location, current battery level, current water level, maximum water capacity, cistern height, and base radius.

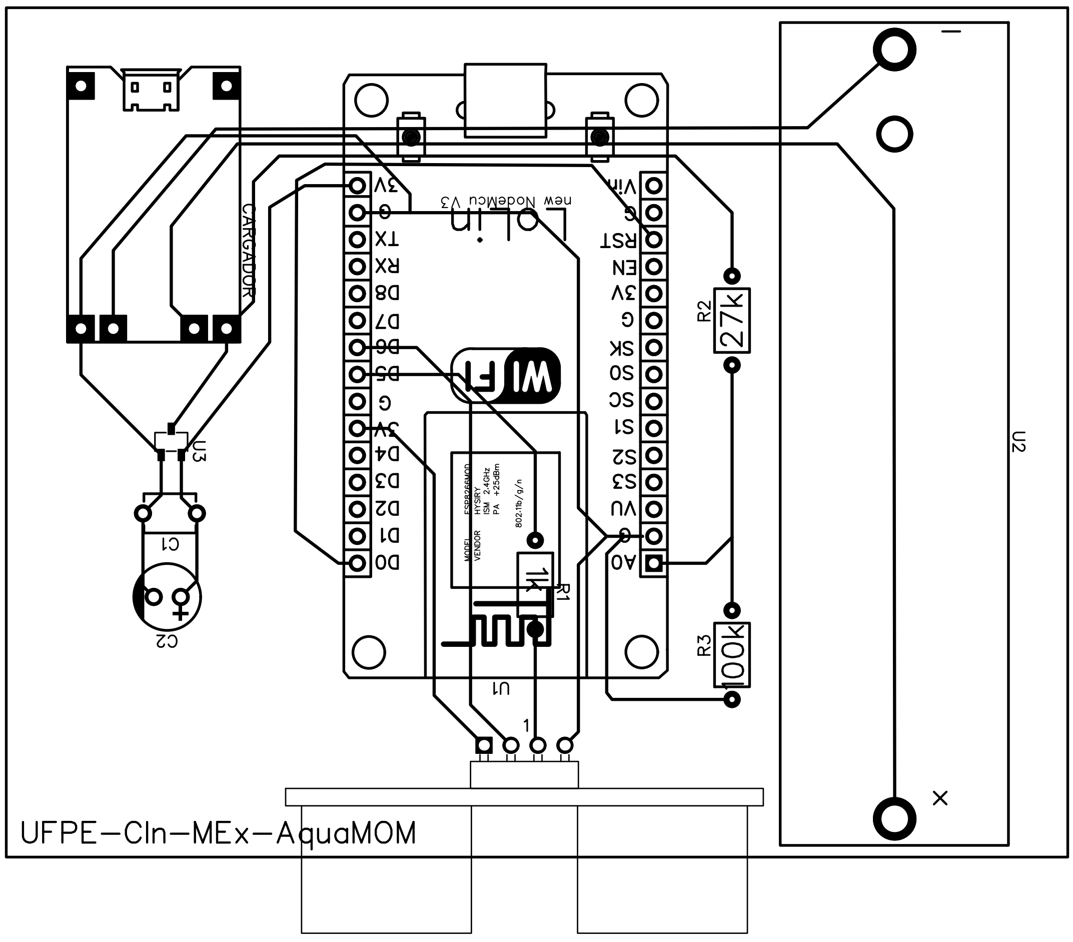
The table *Device_History* stores historical data collected by each device, which can be consulted for historical analysis or behavior tracking. It includes attributes such as a unique identifier (ID), battery level at a specific time, water level at that time, data collection timestamp, and a foreign key (Device_ID) referencing the corresponding device in the table *Device*.

The relationship between the tables *Device* and *Device_History* is one-to-many, which means that each device can have multiple associated historical records, allowing one entry in the table *Device* to relate to many entries in the *Device_History*. Each historical record is linked to a specific device through the Device_ID foreign key, facilitating easy tracking and analysis of data over time. In general, this relational model efficiently organizes the data and supports various queries and analyses related to the operation and performance of the water monitoring system.

APPENDIX C – SCHEMATIC OF THE AQUAMOM



APPENDIX D – PRINTED CIRCUIT BOARD (PCB) OF THE AQUAMOM


CARGADOR
U3
C1
C2
new NodeMcu V3
3V
G
TX
RX
D8
D7
D6
D5
G
3V
D4
D3
D2
D1
D0
Vin
G
RST
EN
3V
G
SK
S0
SC
S1
S2
S3
VU
G
A0
R1
1k
R2
27k
R3
100k
U1
U2
1
UFPE–CIn–MEx–AquaMOM