creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학박사학위논문

서비스 지향 IoT를 위한 새로운 계층적 엣지 기반 아키텍처

A Novel Hierarchical Edge-based Architecture for Service Oriented IoT

2023년 8월

서울대학교 대학원
컴퓨터공학부
김 의 석

서비스 지향 IoT를 위한 새로운 계층적 엣지 기반 아키텍처

A Novel Hierarchical Edge-based Architecture for Service Oriented IoT

지도교수 하 순 회

이 논문을 공학박사 학위논문으로 제출함

2023년 4월

서울대학교 대학원

컴퓨터공학부

김 의 석

김의석의 공학박사 학위논문을 인준함

2023년 6월

위 원 장	권 태 경	(인)
부위원장	하 순 회	(인)
위 원	엄 현 상	(인)
위 원	이 영 기	(인)
위 원	양 회 석	(인)

Abstract

A Novel Hierarchical Edge-based Architecture for Service Oriented IoT

Euiseok Kim
Department of Computer Science and Engineering
College of Engineering
The Graduate School
Seoul National University

The Internet of Things (IoT) is becoming more common in our daily lives, and there are many commercially available IoT platforms. As IoT platforms are the core of building IoT solutions, the design of IoT platforms determines the characteristics of IoT systems, so various designs of platforms are being researched. Among them, edge-based platforms have the advantage of responsiveness and privacy protection because the middleware is located close to the device, and service-oriented IoT has the strength of allowing users to communicate with the IoT without specifying a device. Various research efforts are underway to overcome the weakness of edge-based platforms, which is the lack of scalability, but they all consider only a flat structure, i.e., edge-to-edge connectivity through a single layer.

In this thesis, we propose a hierarchical architecture for edge middleware that maintains the advantages of edge-based IoT and service-oriented IoT while allowing flexible scaling along a tree structure. We identify design requirements and technical challenges to realizing this new architecture and propose solutions. The five main requirements of the hierarchical middleware structure, resource sharing, and scheduling are addressed through two novel techniques. First, we introduce a new type of service called a super

service to enable privacy-preserving edge-to-edge communication and resource sharing. It acts as a service request arbiter in the layer and takes over service requests between each middleware. Second, to enable flexible mapping between devices and services in the service-oriented IoT, we devise a distributed algorithm for the scheduling problem that is made more complex by the addition of super services. This enables real-time responsiveness of super services while distributing the scheduling load across multiple edges to avoid computational overload on any one edge.

As another contribution of this thesis, we develop a simulation framework to evaluate the proposed platform. When building an IoT system, its characteristics can vary depending on the purpose and scale, such as a smart home or a smart factory, and the goals of the system administrator can also vary, such as saving energy and improving user experience. IoT systems are characterized by a large number of different devices, making them difficult to model. We implement a hardware-in-the-loop simulator that creates a hierarchical IoT environment on top of multiple real devices and observes the actual interaction between middlewares and end devices by triggering randomly generated events according to the settings. Based on this simulator, we provide a framework for evaluating the performance of different scheduling algorithms in different IoT environments. This allows users to pre-create IoT environments based on the size and characteristics of their target system and find the optimal scheduling algorithm. This framework has been released on GitHub for further research.

Finally, we verify the feasibility of the proposed platform through the implementation of a smart campus testbed and various experiments using the simulation framework. We installed Raspberry Pi in two buildings and five rooms at Seoul National University to build a hierarchical structure and run various IoT applications connecting different edges to verify that it works in the real world. To check the performance of the super service, we adjusted the height, width, and number of applications in the tree and measured the response speed as the load increased, proving that the performance of the super service

is sufficient even in a large hierarchy. We also show that different scheduling policies give different results when applied to the same environment, demonstrating the need for a scheduling framework to help develop scheduling algorithms.

Keywords : IoT Platform, Edge-based, Service-oriented, Hierarchical Architecture

Student Number : 2015-21232

Contents

Abstract . . . . . **i**

Contents . . . . . **iv**

List of Figures . . . . . **vii**

List of Tables . . . . . **ix**

List of Algorithms . . . . . **x**

Chapter 1 Introduction **1**

1.1 Motivation 1

1.2 Contribution 7

1.3 Dissertation Organization 9

Chapter 2 Related Works **10**

2.1 Cloud-based and Edge-based Platforms 10

2.2 Large-scale System Construction with Edge-based IoT 12

2.3 Scheduling in IoT 15

2.4 Service Composition in IoT 17

Chapter 3 Background **19**

3.1 Service-oriented IoT 19

3.2 SoPIoT 21

3.2.1 Overview 22

3.2.2 Device Management 24
3.2.3 Script Language 26
3.2.4 How to Define an IoT Application 29
3.3 Comparison of Mesh and Tree Structures 31
3.4 Scheduling in Service-oriented IoT 34

Chapter 4 Proposed Hierarchical IoT Platform 39
4.1 Overview 39
4.2 Design Requirements 40
4.2.1 How to assign the edges to the levels of the tree structure? 40
4.2.2 How to request services that the other middleware can serve? . . . 41
4.2.3 How to maintain the privacy of the edge-based platform? 43
4.2.4 How to map and schedule the service requests to devices in each middleware? 43
4.2.5 How to determine if the newly added application can be served in the current IoT system? 45
4.3 Super Service 46
4.4 Scheduler 49
4.4.1 Scheduling of Super Services 52
4.5 Scheduling Policies 56
4.5.1 Handling Events on Device State Changes 58
4.5.2 Handling Events on Execution Request 58
4.5.3 Handling Events on Execution Result 59
4.5.4 Handling of Timeout Events 60
4.6 Test-bed: Toy-level Smart Campus 61

Chapter 5 IoT Simulation Framework 69
5.1 IoT System Simulator 69

5.2 IoT Environment Generation 71
5.3 IoT Event Simulation 73
5.4 IoT System Evaluation 75
5.5 Scheduling Framework 77
5.6 Experiments 78
5.6.1 Experimental Setup 79
5.6.2 Super Service Latency 81
5.6.3 Scheduling Policy 88

Chapter 6 Conclusion and future work 90

Bibliography 92

요 약 99

List of Figures

Figure 1.1 Overview of the proposed hierarchical middleware structure 4

Figure 2.1 Mesh-based vs. tree-based 13

Figure 3.1 Overview of SoPIoT system 22

Figure 3.2 Abstract model of SoPIoT 25

Figure 3.3 A script editor in SoPIoT 26

Figure 3.4 Syntax definition of the SoPIoT script language 27

Figure 3.5 Script samples 28

Figure 3.6 An example showing the different mapping choices 35

Figure 3.7 A scheduling example of two IoT applications 37

Figure 4.1 Service list shared between middlewares in the middleware tree. . . 47

Figure 4.2 Overview diagram of the scheduler 48

Figure 4.3 State transition diagram of a service 50

Figure 4.4 State transition diagram of an application 50

Figure 4.5 Smart Campus 64

Figure 4.6 Two use-cases of Super Service 65

Figure 4.7 Sensor data collected in the smart campus test-bed 66

Figure 4.8 Exhibition Room 68

Figure 5.1 Overview of Simulation Process 70

Figure 5.2 Simulation environment generation 71

Figure 5.3 Workflow of the proposed scheduling framework 79

Figure 5.4 A simulation result example 80

Figure 5.5 The average latency of a super service and the overhead as we vary the height of the middleware tree 82
Figure 5.6 The average latency of a super service and the overhead as we vary the width of the middleware tree 83
Figure 5.7 The average latency of a super service and the overhead as we vary the number of services per super service 84
Figure 5.8 The average latency of a super service and the overhead as we vary the number of super service requests per middleware 85
Figure 5.9 The average latency of a super service and the overhead as we vary the number of virtual devices per bottom middleware 86
Figure 5.10 A sequence diagram of super service execution with the overhead information of each step 87
Figure 5.11 The average latency of services as we vary the range of application periods 88

List of Tables

Table 2.1 Comparison of scalable edge-based IoT platforms 14

Table 3.1 Comparison of mesh and tree structures in distributed systems . . . 32

Table 4.1 Service state table: an example 50

Table 4.2 Application state table: an example 51

Table 4.3 Mapping table: an example 52

Table 4.4 Events to be handled by scheduling policies 67

Table 5.1 Events to be supported by the simulator 74

List of Algorithms

Algorithm 1 Super Service Mapping Algorithm 55
Algorithm 2 Super Service Execution Algorithm 56

Chapter 1

Introduction

1.1 Motivation

As IoT (Internet of Things) is becoming increasingly prevalent in everyday life, it is predicted that there will be 19.1 billion IoT devices by 2025 [1]. Early IoT devices such as Google Nest [2] and Amazon Alexa [3] have demonstrated how IoT can change our daily lives. Smart devices can sense the environment and react according to our intention without our intervention [4, 5]. These days, the emergence of the Matter protocol [6] is making IoT devices more interoperable even though their manufacturer is different.

Hundreds of IoT platforms have been developed to keep pace with this IoT growth, and as of the end of 2021, more than 400 IoT platforms exist in the market, either as open-source or commercialized products [7, 8, 9]. An IoT platform is a software framework that enables the management and connectivity of various smart devices and applications over the Internet, serving as a bridge between the layers of an IoT system, from the device layer to the application layer.

IoT platforms can be categorized into two main types based on the location of their middleware: cloud-based and edge-based. A cloud-based platform deploys a PaaS (Platform as a Service) middleware in the cloud and connects IoT devices through the Internet network [10]. While cloud-based platforms are more popular because of their ease of implementation, low price, and scalability, edge-based platforms have several benefits over

cloud-based platforms by bringing the capabilities of cloud computing to an edge device. An edge-based IoT platform provides improved responsiveness with lower latency and additional safety benefits since data does not have to be transmitted to the cloud. In addition, it has the advantage of being able to continue operations even when the network situation is bad. To mitigate their weaknesses, many cloud-based platforms tend to adopt a hybrid approach by incorporating edge middleware between the device and the cloud or allowing for connectivity between edges through the cloud, which forms a hierarchical IoT architecture. Distinguished from these hybrid approaches, the proposed IoT platform in this work is a purely edge-based platform that solves the scalability problem of an edge-based platform by introducing a hierarchical structure of edge middlewares.

Recent privacy concerns have led to the emergence of applications such as messaging apps that prioritize data protection by not storing personal information. This is true for IoT systems as well, where there is a lot of information flowing around, so the edge-based IoT approach of not storing any information externally at all is the best way to protect privacy. Edge-based IoT systems are built around edge middleware, and as long as there are secure connections between them, the privacy-preserved safe system can be scaled up incrementally. So, it's worth thinking about how to scale up an edge-based IoT system with these advantages, which brings us to the next question: How can edge-based IoT be applied to large-scale systems?

There exist edge-based IoT platforms that try to solve the scalability problem by providing a mesh network for interconnections between edges, such as openHAB[11] and HomeAssitant[12]. Although mesh topology offers advantages such as link robustness and short communication paths, as the number of edges grows, the complexity of managing them grows exponentially, which can be overwhelming for an edge device. Hence mesh-style interconnection is applicable to small-scale IoT systems like smart homes.

In contrast, we propose a novel hierarchical structure of middlewares through a tree topology of edges. An example of the proposed structure is illustrated in Fig. 1.1. As

shown in the figure, at the bottom level, an edge-based platform makes a smart office. A floor has an upper-level middleware that is connected to bottom-level middlewares. Multiple floors compose a building, and multiple buildings cover the entire campus. Since an addition or deletion of a middleware can be done by simply changing the connection between a middleware and its parent middleware, it offers scalability and flexibility. Section 4.6 presents how a toy-level test-bed has been built in our campus as a proof-of-concept implementation of the proposed IoT platform.

IoT platforms can also be classified into two approaches: device-oriented and service-oriented. A user can control an IoT device directly to define an IoT application in a device-oriented platform. In a service-oriented platform, on the other hand, an IoT device is abstracted as a set of services it provides, and an IoT application is specified by a set of services requested by the user [13, 14]. Such abstraction enables us to attach any computing resource, including the cloud, to the IoT platform by assigning each service provided by the computing resource to a virtual device. The proposed IoT platform is a service-oriented platform, as shown in Fig. 1.1, where two IoT applications (User App 1 and User App 2) are programmed by a script that requests services without specifying which device to use. For reusability and convenience, we may define a *composite* service that is a sequence of services that can be conditionally triggered or repeated with a given termination condition. For instance, User App 1 can be defined as a composite service. An IoT application can be modeled using a directed graph, referred to as a service graph, in which nodes represent services and directed edges represent dependencies between them [14]

We first identify the five main design requirements to design the new edge-based hierarchical middleware structure for a service-oriented IoT platform: how to organize the tree, how to share resources between edges, how to preserve privacy, etc. The details of the proposed structure are derived from these requirements and associated technical challenges, which are also a technical contribution to this work. The proposed platform

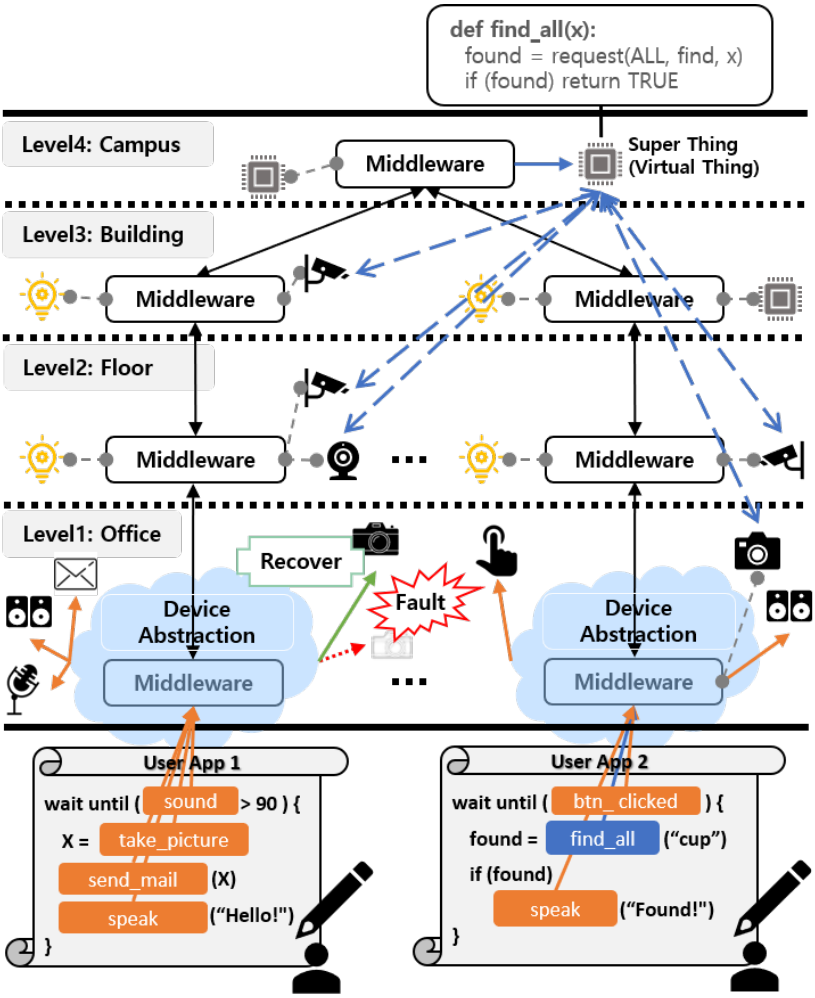


Figure 1.1: Overview of the proposed hierarchical middleware structure

includes techniques for connecting many edges, and for specifying and executing services on each edge in a service-oriented manner. Reliability issues arising from hierarchical indirect communication between nodes rather than peer-to-peer connections can be another challenge, but we believe that they can be complemented by currently known solutions and that maximizing the management and performance benefits of hierarchical structures is well suited to edge-based IoT systems.

To ensure effective communication and resource sharing while maintaining the privacy advantages of the edge-based platform, we introduce a novel element called the

super service This service type is a composite service defined at an upper level above Level 1 in the hierarchical tree structure illustrated in Fig. 1.1. The example showcases a Python-based super service, *find_all(x)*, situated at the top level. When invoked, this service sends the *find* service request down through intermediate nodes to the bottom level. Super services serve multiple purposes within the system. They can be utilized for upper-level middleware to manage and monitor lower-level nodes, enabling efficient hierarchical control. Additionally, they are invaluable when one middleware requires access to a service provided by another middleware. To achieve this, the super service is made visible to the lower-level middleware, allowing the latter to call the defined super service from the upper level. This is exemplified in Fig. 1.1, where User App 2 successfully calls the *find_all* super service located at the bottom layer.

In a service-oriented IoT platform, the middleware assumes a critical role as the central scheduler responsible for mapping and scheduling services onto available devices. Unlike traditional distributed systems, an IoT environment presents dynamic variations in computing resource status, necessitating real-time monitoring by the middleware before making scheduling decisions. The complexity intensifies with the introduction of super services, as their service requests from upper-level nodes may arrive unpredictably and with varying deadline constraints. As a distributed scheduler, the middleware faces the challenge of efficiently managing multiple concurrent applications across networked computing resources while considering the constantly changing device states. The scheduling process must adapt to the dynamic nature of IoT environments and efficiently map service requests to devices. Moreover, the presence of super services introduces additional scheduling complexities, as their calls may have different priority levels and dynamic timing requirements. The middleware must effectively handle such scenarios to ensure timely and reliable service execution.

For the study of scheduling techniques, we have developed a comprehensive simulation framework to evaluate various scheduling policies. This framework empowers users

to configure the IoT system by adjusting hyper-parameters such as the number of nodes at each level, services, applications, and more. Leveraging these configurable settings, the simulation automatically constructs the IoT system by deploying middlewares on real hardware platforms and virtual devices on the simulation host. The framework offers flexibility in testing different scheduling policies, allowing users to seamlessly switch between policies and observe their impact on the performance of the simulated IoT system. By analyzing the system under different scheduling policies, researchers and practitioners gain valuable insights into the strengths and weaknesses of each policy in various scenarios. This framework[1] is publicly available to let anyone use it for further study on the scheduling policies.

[1] https://github.com/sopiot/simulation-framework

1.2 Contribution

The contributions of this work can be summarized as follows.

- We present the hierarchical middleware structure of edge-based IoT platforms and associated design requirements. In this platform, unlike the existing edge-based IoT platform with one middleware in the center, multiple middlewares are hierarchically connected to form an extension structure. At this time, technical challenges related to the design requirements are described, and solutions to resolve them are presented.

- We introduce super service, a new type of service that enables collaboration among middlewares at the service level, as well as management of middlewares in a hierarchy. A super service is a composite service made up of a combination of multiple services and can request the execution of other middleware services. Users exploit desired functionalities through IoT applications that include super services.

- We propose a scheduler to map and schedule service requests to devices in each level of middleware. Since each middleware manages only the devices registered to it, it supports super service through a distributed scheduler. Efficient distributed algorithms for each super service scheduling and execution are devised.

- We present a simple test bed implementation with the proposed hierarchical IoT platform. At Seoul National University, we have built a small smart campus system that includes two buildings and five rooms.

- A simulator that creates an IoT environment with the proposed platform and triggers various events is implemented. It automatically builds a middleware tree and generates codes of virtual devices according to the configuration. It also dynamically registers/unregisters the devices and observes interactions between middlewares and virtual devices by executing IoT applications in parallel.

- An IoT simulation framework of the IoT system based on the simulator is developed to estimate the performance of the given IoT system and evaluate the scheduling algorithms.

- Various experiments are performed with the simulator to confirm the effectiveness of the proposed platform. It shows the negligible overhead of the super service and the improvement of the latency when changing the scheduling policy. Moreover, additional measurements that break down the overhead and show the importance of the scheduling policies are conducted.

1.3 Dissertation Organization

The remaining chapters of this thesis are organized as follows. In the next chapter, related works are reviewed. Chapter 3 explains some background knowledge of this work, and Chapter 4 presents the proposed hierarchical platform in detail, including the design requirements and associated technical challenges with our solutions for those. In this chapter, the practical use case of the proposed technique is demonstrated by implementing a simple smart campus test bed. Next, the IoT simulation framework for this platform is proposed in Chapter 5. The IoT simulator that generates IoT environments and evaluates the performance of the scheduler is developed, and with this simulator as an engine, the performance and scalability of the proposed IoT platform are evaluated through extensive experiments in this chapter. Finally, the conclusion and future work are outlined in Chapter 6.

Chapter 2

Related Works

2.1 Cloud-based and Edge-based Platforms

In the landscape of IoT platforms, many popular platforms initially adopted a cloud-based approach due to its ease of installation and rapid user adoption. However, to address the limitations of cloud-based IoT platforms, such as reliance on internet connectivity and potential latency issues, a trending shift towards a hybrid architecture, known as fog computing, has emerged. Fog computing includes edge devices as intermediate hubs, reducing response time and network dependence. Despite these benefits, concerns regarding privacy persist as information is ultimately transmitted to the cloud [15].

Among the cloud-based IoT platforms, several well-known systems have gained significant recognition. AWS IoT Core [16], for instance, constructs an IoT platform utilizing a general-purpose cloud infrastructure. It establishes connectivity between devices and the AWS cloud, enabling the development of IoT systems that accesses cloud resources such as databases and computing capabilities. The platform offers an easily manageable initial setup process through pre-written installation scripts. However, customization of the system may require a deeper understanding of its intricacies. Samsung SmartThings [17] and Apple HomeKit [18], on the other hand, are exemplary cloud-based IoT platforms offered by device companies. These platforms enable users who purchase their devices to enjoy IoT capabilities through a free cloud-based platform. In

an effort to improve latency and enhance user experiences, both platforms have recently adopted a hybrid approach. This approach uses computing resources in devices, such as televisions, as local hubs. However, it is important to note that while the hybrid approach improves performance, the majority of the management process still occurs in the cloud, raising concerns regarding privacy and data security.

With the edge-based approach, there is no need for a cloud infrastructure in IoT. Instead, an edge middleware is deployed within the local network. This placement enables resource management and computation to occur directly at the edge, offering several advantages such as improved responsiveness, privacy, and network robustness [19, 20]. Exploiting these benefits, numerous edge-based IoT platforms have been proposed across various application domains. In the agricultural sector, where internet connectivity in farming areas may be unstable, edge-based platforms with edge nodes have been introduced to enable local storage and lightweight data processing for livestock and crop monitoring [21]. These platforms address the specific needs of the industry, ensuring reliable and efficient operations even in challenging network conditions. For smart home applications, openHAB [22] and Home Assistant [23] have garnered attention as popular edge-based IoT platforms. These open-source solutions have vibrant communities contributing to the development of useful plugins and extensions, flourishing the functionality and customization options available to users. These platforms let users set up a smart home environment without any dependency on external cloud services. In the healthcare industry, Pasquale Pace et al. [24] propose an edge-based platform as an alternative to cloud-based solutions. This platform aims to address the unique requirements and privacy concerns in healthcare by enabling data processing and analytics at the edge, closer to the source of data generation. By capitalizing on edge computing, healthcare providers can enhance data privacy, reduce latency, and ensure reliable operation, ultimately improving the quality of care and patient outcomes.

These edge-based IoT platforms exemplify the growing trend of deploying IoT so-

lutions without relying on the cloud. By bringing computation and data processing capabilities closer to the edge, users can benefit from improved performance, privacy, and reliability within their local network environments. The adoption of edge computing in diverse application domains continues to expand, catering to specific requirements and unlocking the potential of edge-based IoT solutions.

2.2 Large-scale System Construction with Edge-based IoT

Some existing edge-based platforms have attempted to set connectivity between multiple edges via add-ons to make the system scalable. For example, openHAB offers a plug-in called ”Remote openHAB Binding” that enables the communication between edges, allowing for the exchange of service information and the creation of multi-edge applications. Similarly, Home Assistant provides ”Remote Home-Assistant” integration, which lets one edge middleware connect to other edge middlewares and request services based on events or service requests from the other edge middlewares. However, the scalability offered by this approach is insufficient to build a large-scale IoT system, such as a smart campus, as the resource capacity of an edge device is limited. Both platforms force one edge to manage all other edges, which exponentially increases the computational workload. These plug-ins are designed for integration within a single user’s system and do not support system-level management functions, such as edge-specific access control, distributed scheduling, etc.

There exist some platforms designed for scalable edge-based IoT. Alonso et al. [9] proposed a scalable architecture for mobile healthcare applications based on edge computing to achieve the benefits of privacy and latency. They have an edge gateway that communicates with other edges from different areas to support distributed services. Rivera et al. [25] proposed a scalable edge-based blockchain platform. The edge layer and blockchain layer of this architecture let edge nodes form a distributed peer-to-peer

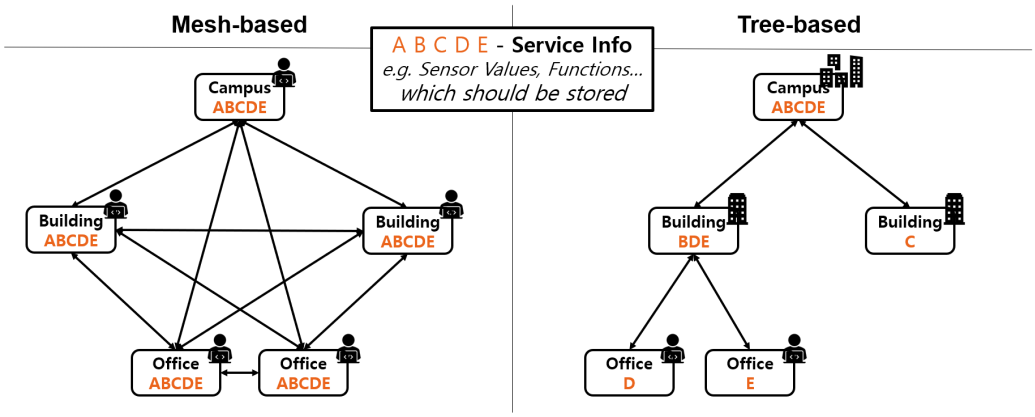


Figure 2.1: Mesh-based vs. tree-based

network to manage secure data transactions between IoT devices. They try to distribute the data transaction between IoT nodes which can surpass the capability of a single server with the distributed ledger technique, but the other type of IoT service and scheduling issue is not considered. In all of these approaches, edges are connected with a single-layer network such as mesh and star.

On the other hand, it may be possible to connect the edges vertically as proposed in SoPIoT [14]. Fig. 2.1 compares two interconnection structures between edge devices: mesh-based and tree-based. The letters in the diagram represent service information data that must be stored at an edge. Due to the fact that the mesh-based approach requires the system to store and manage information for every linked node, it requires more effort to maintain the topology.

There are several benefits to using a tree structure over a mesh structure. In comparison to a mesh-based structure, a tree structure requires less overall communication volume between edge devices, although it may require more hops of communication between edges. Moreover, the tree structure inherently supports the clustering of edges based on their geographic location. Adding a new edge device to the existing system is also a straightforward process, as it can be connected to a parent node without affecting other middlewares. This scalability makes management and maintenance simpler.

Table 2.1: Comparison of scalable edge-based IoT platforms

Aspect	openHAB [22]	HA [23]	Alonso et al. [9]	Rivera et al. [25]	SoPIoT [14]	Ours
Hierarchical Edge MW	X	X	X	X	△	O
Service-oriented	△	△	△	X	O	O
Mapping and Scheduling	△	△	X	X	O	O
Load Balancing	X	X	O	O	X	O

Even though the authors of [14] have proposed a tree-based hierarchical structure of edge devices, no real implementation is shown and associated technical challenges are not identified nor solved. In contrast, we present the technical issues to consider in the construction of a tree-based hierarchical IoT platform and propose a real implementation in this work.

Table 2.1 compares five existing scalable edge-based platforms with the proposed platform in four aspects: edge middleware topology, device-oriented or service-oriented, mapping and scheduling, and load balancing. All existing platforms but SoPIoT use a mesh topology to connect multiple edge middlewares for scalability. Even though SoPIoT proposes a hierarchical tree structure of edge middlewares, no implementation has been made.

While openHAB and HomeAssistant abstract a device with a set of services the device offers, they are not service-oriented in a strict sense since multiple devices may not share a service. Device-oriented platforms do not need a mapping of services to devices. Instead, openHAB and HomeAssistant allow scheduling the timing of execution through time conditions when running IoT applications. Since SoPIoT and the proposed platform are service-oriented, they perform mapping of service to devices as well as service scheduling.

Alonso et al. [9] targets microservices running through Docker containers and provides load balancing through dynamic microservice deployment. Rivera et al. [25] fo-

cuses solely on data transactions between edges and decentralizes data communication in IoT through blockchain technology. These two works consider load balancing as a feature of middleware.

In summary, the proposed platform is distinguished from the existing works as the only service-oriented edge-based IoT platform, providing scalability with a hierarchical tree structure of edge middlewares. Note that hierarchical edge middleware is suggested and implemented with technical considerations only in this work.

2.3 Scheduling in IoT

The scheduling problem in IoT systems, which involves allocating incoming applications to devices, has been a subject of study due to the heterogeneous, complex, and dynamic nature of such systems [26, 27, 28, 29, 30]. Mabrouk et al. [26] proposed a heuristic approach that dynamically composes multiple services, optimizing both the quality of service (QoS) and energy consumption. However, their method did not consider service dependencies or real-time constraints, which are crucial factors in many IoT scenarios.

Li et al. [28] tackled the scheduling problem in service-oriented IoT systems. They developed a model that represents an IoT system as a composition of application, network, and sensing layers. QoS metrics associated with each layer were determined and optimized during the scheduling process. While their approach enabled rapid service deployment in the IoT environment, it did not take into account the dependencies between IoT services, limiting its applicability in scenarios where service interdependencies are significant.

Kolomvatsos et al. [30] proposed a two-stage task allocation methodology for edge computing environments. Their approach involved assigning tasks to things (devices) that could execute them locally or delegate them to peers. To determine the group of peers capable of executing a task, they employed the k-nearest Neighbor (KNN) algo-

rithm. Rejected tasks were then allocated to another set of peers using the multi-attribute utility theory. While their methodology demonstrated effective load balancing in a distributed manner, it did not consider latency constraints, which are crucial in real-time IoT applications.

Extensive research has been conducted on hybrid scheduling approaches that bridge the cloud-edge continuum in IoT systems [31, 32, 33]. Bhatia et al. [34] proposed a quantum computing-inspired scheduler, exploring the potential of quantum computing techniques in optimizing task scheduling. Pusztai et al. [35] introduced an edge-aware scheduler specifically designed for asynchronous IoT applications, aiming to achieve load balancing in fog computing environments. Lim [36] presented an IoT application scheduling method based on artificial intelligence, targeting small-scale fog computing environments. These studies showcase the diverse range of approaches in hybrid scheduling, each with its own objectives and assumptions. However, they may differ from the focus of this work, which revolves around a purely edge-based IoT platform and addresses challenges such as device failures and real-time requirements.

In the realm of hierarchical fog IoT systems, researchers have proposed architectures and scheduling techniques that utilize the hierarchical structure of fog layers. De et al. [37] and Chekired et al. [38] introduced architectures that construct hierarchies within the fog layer, enabling efficient resource management and scheduling based on the hierarchical structure. While these studies provide valuable insights into hierarchical fog IoT systems, their assumptions and objectives are for systems with the cloud and may differ from the scope of this study.

In the context of runtime modifications in distributed systems, researchers have explored techniques for dynamically adapting task execution and system configuration. Yerraballi et al. [39] and Zhu et al. [40] investigated variations in task execution times in real-time distributed systems, considering the dynamic nature of task execution. Bate and Emberson [41] focused on enhancing the flexibility of embedded systems by considering

runtime modifications to task graph topology. However, these studies typically require comprehensive offline optimization before any runtime reconfiguration can be applied.

A recent study introduced a hybrid scheduling technique that considers the real-time requirements of IoT applications and the dynamic changes in system configuration due to device addition and deletion [42]. This algorithm incorporates both global and incremental steps to effectively respond to the dynamic nature of the IoT environment. However, it does not specifically address the hierarchical structure of middlewares and the associated technical challenges, which are the focus of this work.

2.4 Service Composition in IoT

Service composition stands as a pivotal advantage offered by service-oriented IoT platforms, allowing users to amalgamate abstracted services from individual devices to create new, integrated services. The capability to compose services within a single IoT system to craft customized applications is a prominent feature provided by most service-oriented platforms, as highlighted in [43] and [44]. Moreover, service composition can extend beyond the boundaries of a singular IoT environment, encompassing heterogeneous IoT landscapes in what is termed as federated service composition. This advanced concept facilitates the development of large-scale IoT systems by seamlessly integrating diverse types of IoT systems, as exemplified by the work of [45]. IoT ecosystem can transcend individual administrative domains, culminating in a synergistic collaboration between distinct IoT environments through federated service composition.

Federated service composition allows for the seamless integration of services from disparate IoT environments, enabling collaboration and interaction between different systems. It extends the capabilities of service-oriented IoT platforms beyond a single administrative entity, facilitating the creation of larger, more comprehensive IoT systems. The challenges associated with federated service composition are multifaceted. Interoperability becomes a critical concern, as services from different administrative domains may

employ different communication protocols, data formats, or security mechanisms. Standardization efforts, such as the development of common data models and interoperability frameworks, play a vital role in addressing these challenges.

Furthermore, ensuring security, privacy, and trust across federated IoT environments is of utmost importance. Establishing secure communication channels, implementing access control mechanisms, and adhering to privacy regulations become vital components of federated service composition. Efforts in federated service composition have gained attention in various domains, including smart cities, industrial IoT, and healthcare, where collaboration between multiple entities is necessary to create comprehensive IoT solutions. Research and industry initiatives are underway to develop frameworks, protocols, and best practices for federated service composition to integrate diverse IoT environments into a cohesive ecosystem.

However, it is important to note that this work focuses on a single administrating entity for the IoT system, and therefore, the intricacies surrounding federated service composition and distributed scheduling issues lie beyond the scope of this research.

Chapter 3

Background

3.1 Service-oriented IoT

Since the emergence of IoT, various types of IoT platforms have been proposed [46]. Device-oriented IoT refers to an approach where the focus is primarily on individual devices and their direct communication with each other. In a device-oriented IoT system, devices are considered the central entities and interactions primarily occur between devices. The main objective is to enable devices to connect and exchange data without significant intermediation or abstraction. Device-oriented IoT platforms often emphasize device-level functionalities and provide protocols and frameworks for device discovery, connectivity, and data exchange. These platforms typically allow direct communication between devices, enabling developers to build applications based on device interactions.

On the other hand, service-oriented IoT takes a higher-level view, focusing on the services provided by devices rather than individual devices themselves. In a service-oriented IoT system, devices are abstracted and organized based on the services they offer. Services represent specific functionalities or capabilities that devices provide, such as sensing, actuation, or data processing. Service-oriented IoT platforms aim to unify and manage the services offered by various devices from different manufacturers. They provide frameworks and protocols for service discovery, composition, and consumption. Users interact with the IoT system by requesting services rather than directly accessing

individual devices. The platform handles the underlying complexities of device discovery, connectivity, and data exchange, allowing users to create applications by composing services from multiple devices.

The shift from device-oriented to service-oriented IoT architectures offers several advantages. Service-oriented IoT enables higher levels of abstraction and interoperability by focusing on the capabilities and functionalities offered by devices rather than their specific implementations. It allows for greater flexibility, scalability, and composability as services can be combined and reused across different applications. Service-oriented IoT platforms facilitate easier integration of devices from various manufacturers and provide a more user-friendly and intuitive interface for application development. Additionally, service-oriented approaches can enable advanced features such as dynamic service composition, adaptive service selection, and context-awareness, enhancing the overall capabilities and intelligence of the IoT system.

While the service-oriented IoT architecture has been studied for more than a decade [47, 48, 49], early IoT platforms were mostly device-oriented, allowing direct communication between devices [50, 51, 52] as the concept is clear to implement. However, service-oriented IoT has gained significant attention in recent years because it offers a higher-level view of the system to users, as evidenced by the popularity of state-of-the-art platforms [17, 22, 53]. They claim to be service-oriented platforms based on the fact that they can abstract and view devices as services.

Since they do not allow multiple devices to share the same service name, however, they are not service-oriented in a strict sense. In the strict sense of service orientation, an IoT system can be considered as a distributed system in which devices act as processing elements (PEs) and services as tasks. Service Oriented Middleware (SOM) is an OS-like manager supporting adaptive service composition, management of heterogeneous devices, and scheduling of services to devices. Exploiting this position, the middlewares are able to support useful smart features such as auto-recovery and auto-optimization to

users by adjusting service providers for their requirements in run-time. An IoT platform can interact with users through a composite service that can be represented as a directed acyclic graph (DAG) of service nodes [14].

As IoT technologies continue to develop and evolve, it is becoming increasingly clear that service-oriented IoT will emerge as the dominant paradigm shaping the future of IoT. The inherent advantages of service-oriented architectures, such as high levels of abstraction, interoperability, and composability, are driving widespread adoption of personal IoT technologies. Service-oriented platforms that provide optimization and recommendation capabilities through intelligent service composition will enable IoT to reach its full potential. With service-oriented IoT at the forefront, users can expect a future where IoT technologies seamlessly integrate into their daily lives, simplifying and enhancing their experiences across various domains such as smart homes, healthcare, transportation, and beyond. The unparalleled scalability, flexibility, and intelligence offered by service-oriented IoT will revolutionize the way we interact with our environment, unleashing the true potential of the IoT in transforming our world into a seamlessly interconnected and personalized ecosystem.

3.2 SoPIoT

The proposed platform is based on a service-oriented IoT platform called SoPIoT [14]. This platform is the foundation of an independent IoT system, each node in the proposed hierarchy. Since the basic design and the local middleware as individual nodes follow SoPIoT, a background knowledge of the platform's features is necessary to understand this research. In this section, we will provide a detailed overview of SoPIoT, including its application specifications, service mapping, and service scheduling.

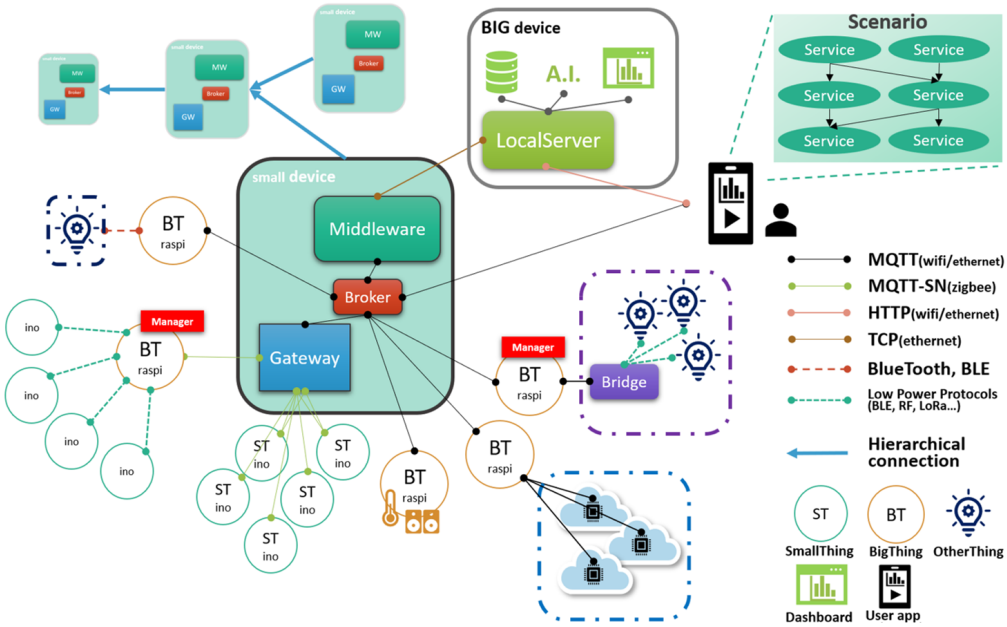


Figure 3.1: Overview of SoPIoT system

3.2.1 Overview

The baseline IoT platform of this thesis, SoPIoT, adopts a comprehensive architecture as depicted in Figure 3.1. At the heart of this architecture is the middleware, which encompasses key functionalities commonly associated with SOM, including device management and programmability [54, 55]. The middleware serves as a crucial component facilitating communication between IoT devices and the broader system.

To establish the connection between the middleware and IoT devices, the MQTT (Message Queuing Telemetry Transport) communication protocol is employed. MQTT is a widely adopted ISO standard publish-subscribe-based messaging protocol built on top of the TCP/IP protocol [56]. In SoPIoT, the Mosquitto open-source project [57] is adapted as the MQTT broker. This broker process operates as a daemon, running simultaneously with the middleware. Both the middleware and the IoT devices run as MQTT clients, enabling the exchange of messages. The Mosquitto broker offers various options, includ-

ing Secure Sockets Layer (SSL) [58] for communication security and Quality of Service (QoS) settings for balancing speed and reliability. With the chosen QoS-0 protocol, no handshaking occurs after message transmission, resulting in faster communication. However, it is worth noting that this approach introduces certain risks, such as the potential for communication order reversal. By adopting the MQTT communication protocol and the Mosquitto broker, SoPIoT ensures efficient and secure communication between the middleware and IoT devices. This foundation enables seamless data exchange and integration within the IoT platform, enhancing its overall functionality and reliability.

SoPIoT is designed with a strong focus on safety and privacy. It embraces a pure edge-based IoT design, eliminating the reliance on cloud infrastructure. This architectural choice ensures structural privacy, as all personal information generated by IoT systems remains stored on edge devices. Moreover, there is a local server that collects the IoT sensor data and analyzes the patterns. This is a separate server that stores IoT data and logs that the middleware only forwards. For system security, it can provide AI solutions for anomaly detection or provide a system monitoring user interface.

In SoPIoT, the middleware plays a crucial role as a centralized administrator within the edge network. It manages the communication between devices and acts as a gatekeeper to validate the authenticity and integrity of device communications. When a device's identity is registered with the middleware, it gains permission to communicate within the network. By centralizing communication and enforcing validation, the design mitigates concerns regarding potential hacks and unauthorized access to IoT devices. Even if one device is compromised, the restricted direct communication between devices prevents the occurrence of widespread catastrophic damage.

The safety-conscious design of SoPIoT prioritizes the protection of user data and the integrity of the IoT system. By relying on local servers and a centralized middleware, the framework provides a secure and controlled environment for IoT communication. This approach addresses the potential risks associated with cloud-based IoT platforms

and empowers users with enhanced privacy and safety measures. With its edge-based architecture and stringent communication management, SoPIoT offers a robust solution for deploying IoT systems with privacy and security in mind.

Since each service included by an application specified in SoPIoT does not directly specify a device, it is the role of the SoPIoT scheduler to map the user's service request to the best device for the situation. The mapping algorithm that selects the device to execute the requested functionality has a significant impact on system performance. Depending on which mapping algorithm is applied, the system can be tailored to the user's needs, such as the size and speed of the system. Research has been conducted on this topic and proposed an adaptive scheduling technique on SoPIoT in [42]. The scheduling algorithms improve overall system performance, such as energy and utilization, while maintaining rapid response.

3.2.2 Device Management

SoPIoT offers a flexible approach to discovering and registering a wide range of IoT devices. To facilitate device integration, the platform provides a comprehensive Software Development Kit (SDK) that device developers can utilize. By embedding the code from the SDK and communicating with the middleware via MQTT, developers can seamlessly integrate their devices into the SoPIoT ecosystem.

In addition to supporting various IoT devices, SoPIoT extends its device coverage through the integration of the MQTT-SN (Message Queuing Telemetry Transport for Sensor Networks) gateway [59, 60]. This gateway serves as a bridge, enabling connectivity with constrained devices via Zigbee [61, 62]. By incorporating the MQTT-SN gateway, SoPIoT expands its compatibility and interoperability with a broader range of devices. To accommodate embedded devices with constraints, such as Arduino [63], a dedicated SDK is provided. This SDK allows devices running firmware developed with it to connect to the middleware using Zigbee communication through the MQTT-SN gateway.

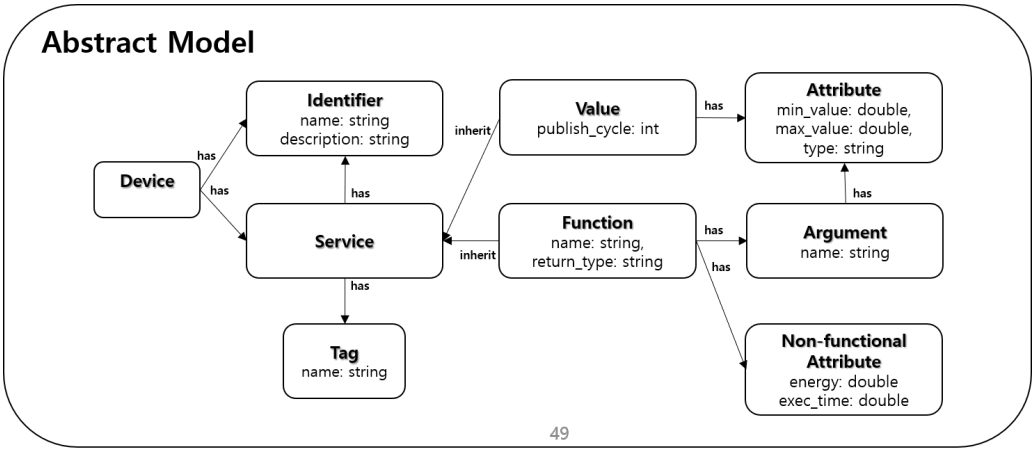


Figure 3.2: Abstract model of SoPIoT

This approach caters to the specific requirements of embedded devices, ensuring their seamless integration into the SoPIoT platform.

Furthermore, SoPIoT offers a versatile integration mechanism for devices from other platforms that may not have dedicated SDKs provided by the platform. This is made possible through the *manager thing*, which acts as a plug-in capable of discovering third-party devices or devices that employ unsupported communication protocols. The manager thing serves as a bridge, converting these devices into SoPIoT-compatible things. With this capability, the platform can embrace additional protocols, such as Bluetooth Low Energy (BLE) for low-power communication. Each platform may have a different integration approach, and developers can easily add a manager thing for each specific platform. In addition, starting in 2023, SoPIoT will facilitate easy integration of devices from even more platforms through the adoption of the Matter protocol, further expanding its interoperability and device integration capabilities.

In SoPIoT, devices are registered to an abstract model through a uniform interface mechanism without affecting other devices at runtime. Because the abstract model hides the hardware characteristics of the device, it can easily accommodate a wide variety of IoT devices, including third-party devices originally developed for other IoT platforms.

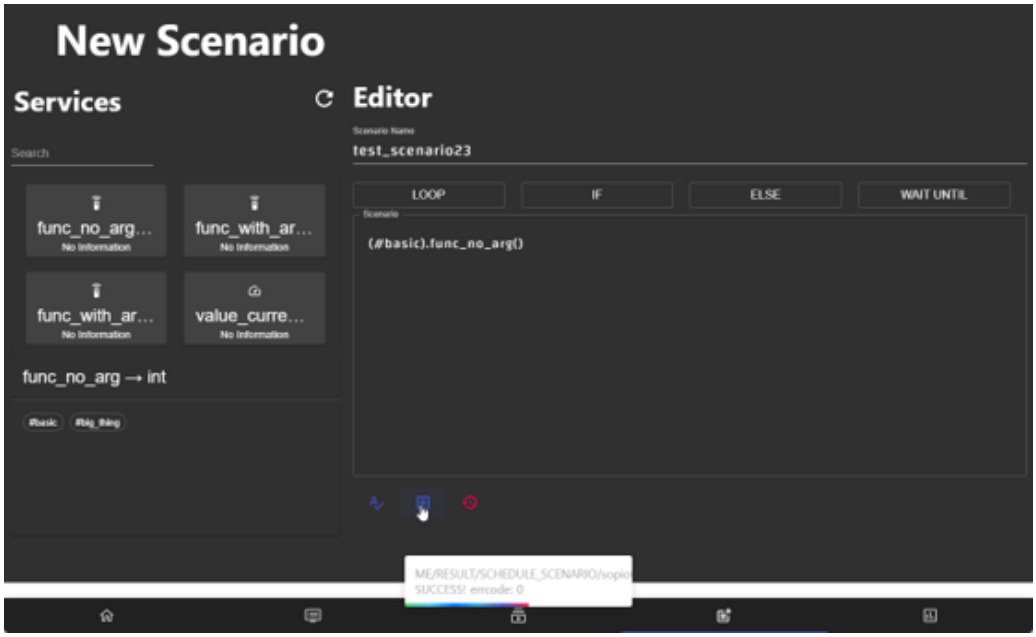


Figure 3.3: A script editor in SoPIoT

The abstract model of a device comprises identifiers and services, as depicted in Figure 3.2. Identifiers contain the name and description of the device. Services are categorized into value type and function type. Value-type services refer to the values reported periodically to the middleware. Functions represent the services that the device can offer, which may involve input arguments and return values. Additionally, functions can include optional non-functional attributes like latency and energy consumption for service execution. When provided, these properties are considered in the middleware's service mapping and scheduling process.

3.2.3 Script Language

Several IoT platforms, such as Watson IoT [64], ThingWorx [53], IFTTT [65], and ThingSpeak [66], offer users the ability to define composite services using the Event-Condition-Action (ECA) paradigm. This paradigm allows actions to be triggered based on specific events that satisfy predefined conditions. While some platforms, like Things-

```
Statementlist → Statement
              |  Statementlist; Statement
Statement     → if( Conditionlist ) Statement
              |  if(Conditionlist ) Statement else Statement
              |  loop(LoopCondition) Statement
              |  wait until LoopCondition) Statement
              |  RangeType(TagList).SERVICE
LoopCondition → /* void */
              |  PeriodTime, Conditionlist
              |  PeriodTime
              |  Conditionlist
Conditionlist → Condition
              |  not Condition
              |  Conditionlist or Conditionlist
              |  Conditionlist and Conditionlist
Condition     → Expression operator Expression
Expression    → RangeType(TagList).SERVICE
              |  N
              |  R
              |  STRING_LITERAL
RangeType     → /* void */
              |  all
              |  any
TagList       → Tag
Tag           → #STRING_LITERAL
```

Figure 3.4: Syntax definition of the SoPIoT script language

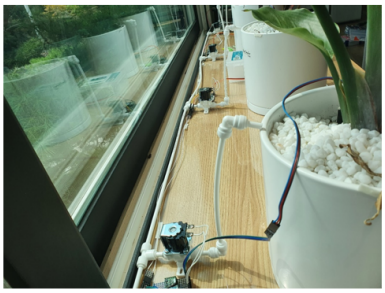
Board [67] and SmartThings [17], provide support for rich programming languages like Groovy, these options can still be challenging for users without traditional programming knowledge.

To address this issue, SoPIoT introduces a high-level scripting language designed specifically for users without programming backgrounds to easily develop composite services. The syntax of this scripting language is illustrated in Figure 3.4. In this language, a composite service is defined as a list of services, with each service specified by its range type, tag list, and name. Tags play a crucial role in specifying the intended mapping of things. Users can attach tags to services and utilize them to express their intentions when creating applications. The scripting language supports three types of control structures.

```
loop(1 MIN) {
  if((#cloton).soil_moisture < 500) {
    (#cloton #valve).on()
    (#pump).on_while(20)
    (#cloton #valve).off()
  }

  … 8 more plants

  if((#areca_palm).soil_moisture < 397) {
    (#areca_palm #valve).on()
    (#pump).on_while(20)
    (#areca_palm #valve).off()
  }
}
```



(a) Smart gardener

```
loop(1 SEC) {
  if((#mov #door).detected == 1) {
    if((#door).mask > 0 or (#door).no_mask  > 0) {
      if((#door #fever).temperature > 37.5) {
        (#tts).speak("Someone with fever detected.")
      }
      if((#mask).no_mask  > 0) {
        (#tts).speak(“Please were a mask.”)
      }
    }
  }
}
```

(b) Corona detector

```
loop(1 MIN) {
    if((#env).temperature > 26) {
        (#air_conditioner).on()
    }

    if((#air).dust > 300) {
        (#air_purifier #samsung).on()
        (#vaccum #samsung).on()
        (#air_circulator).on()
        (#tts).speak(“open the window!”)
    }
}}
```

(c) Environment manager

Figure 3.5: Script samples

Conditional execution of services is achieved using an **if-else** construct, while iterative execution is facilitated by a **loop** construct. The loop execution can be repeated within a defined period, with the option to specify termination conditions. Additionally, the **wait until** statement allows the blocking of loop execution until a specific condition is met. Services are executed sequentially, and the middleware waits for the return of results from the devices. To specify service candidates for mapping in an application, the **#** construct is utilized. For more fine-grained control, the scripting language offers the **all** and **any** constructs. The **all** construct maps to all devices that satisfy the tag specifier for function services, as opposed to mapping to only one device by default. Conversely, the **any** construct allows a conditional statement to be true if any of the candidates for a value service are satisfied, as opposed to requiring all candidates to be satisfied by default.

Figure 3.5 provides some examples of application scripts written in this language, showcasing the creation of a smart garden and an access control system within a single script. This scripting language offers a more expressive interface compared to simple condition-based specifications, enabling users to define complex applications that cater to their specific needs. The hashtag approach provides flexibility and clarity in issuing commands, while the simple syntax facilitates the creation of applications. Furthermore, the easy conversion from block coding or voice commands to this scripting language is possible, ensuring user-friendly application development in SoPIoT.

3.2.4 How to Define an IoT Application

SoPIoT includes a client that features a script editor, as depicted in Figure 3.3. This editor empowers users to define IoT applications, specifically composite services, during runtime. It supports the simple language mentioned earlier, enabling users to express their requirements at a high level without needing detailed knowledge about the underlying physical devices.

In SoPIoT, a service-oriented platform, users approach application specification

from a service-based perspective. The left part of Figure 3.3 shows how users create applications by selecting services provided by the middleware. These services can be categorized into two types: function types and value types. Function types in SoPIoT encompass a wide range of actions that can be performed by IoT devices. They are designed to accept parameters and produce return values, enabling users to invoke the internal functions of the devices. This flexibility allows users to simply exploit the capabilities of their IoT devices. For example, users can initiate hardware actuation to control physical components, such as turning on a smart light bulb or adjusting the temperature of a smart thermostat. Additionally, they can make cloud API calls to access various services and functionalities offered by cloud platforms, such as retrieving weather data or performing language translation. Moreover, SoPIoT supports AI inference functions, allowing users to harness the power of machine learning algorithms deployed on IoT devices for tasks like object recognition or predictive maintenance. By utilizing these function types, users can create sophisticated and intelligent IoT applications that enhance the automation and interactivity of their IoT systems.

On the other hand, value types in SoPIoT represent data collected periodically from IoT devices, providing crucial information for evaluating conditional statements within IoT applications. These values are typically sensor readings or timestamps that reflect the state or behavior of the devices. For instance, a temperature sensor may provide temperature readings, enabling users to create conditional statements based on specific temperature thresholds. This could be used to trigger actions such as turning on an air conditioner if the temperature exceeds a certain level. Similarly, a motion sensor may provide binary values indicating the presence or absence of motion, allowing users to create conditional statements that activate security measures when motion is detected. Timestamps, on the other hand, provide temporal information and enable users to track the occurrence of events over time. For example, users can create conditional statements that execute specific actions at specific times, such as turning on outdoor lights at dusk. By leveraging

these value types, users can incorporate real-time data from IoT devices into their applications and make informed decisions based on the current state of their environment.

To ensure seamless integration and prevent conflicts caused by duplicated service names across devices, SoPIoT employs the powerful specifier tag mechanism. By assigning tags to services of any type, users can easily construct their applications by simply referencing the associated tags from the service list. This approach eliminates the complexity of managing unique service names across multiple devices, streamlining the application development process. Before finalizing an application, SoPIoT performs a thorough verification process, meticulously checking the compatibility of the application with the available services and tags provided by the middleware. Additionally, the syntax is rigorously validated to guarantee correctness. Once the application successfully passes the verification stage, the "Add Application" button becomes active, empowering users to request the creation of their own applications within the middleware. The middleware's advanced scheduler intelligently evaluates the schedulability of the application, seeing its comprehensive knowledge of device capabilities and availability. As a result, users are promptly informed of the scheduling outcome, receiving a success notification if a suitable device capable of executing the application is found, or a failure message if the requested application is not feasible. With SoPIoT's ingenious specifier tag system, users can make their IoT applications without the burden of managing complex service name conflicts, enabling a seamless and efficient development experience.

3.3 Comparison of Mesh and Tree Structures

There are two structures that edge-based IoT systems can adopt as they incrementally expand their systems with edge-to-edge connectivity: mesh-based and tree-based. In a distributed system, connecting multiple edges refers to establishing communication paths or links between nodes to realize data transmission and message exchange. In a mesh structure, each node is directly connected to every other node in the network, form-

Table 3.1: Comparison of mesh and tree structures in distributed systems

Aspect	Mesh-based	Tree-based
Fault Tolerance	High fault tolerance with multiple comm. paths	Vulnerable to single-node failures.
Total Comm. Volume	Higher overhead due to many direct links	Lower overhead
Latency Between Nodes	Lower Direct communication	Higher Long communication paths
Structure Management	Difficult to configure roles and relationships	Easier to configure with well-defined roles
Connection Maintenance	Higher overhead as the network grows	Lower overhead due to less connections

ing a fully connected topology. This means that multiple edges exist between each pair of nodes. For example, if there are N nodes in the mesh network, each node will have N-1 edges connected to it. In a tree structure, nodes are connected in a hierarchical manner, with a single root node at the top and child nodes branching out from the root. Each node has only one parent except for the root, and multiple child nodes can be connected to a parent node.

Table 3.1 shows the comparison of mesh and tree structures in distributed systems. First and foremost, the mesh structure excels in fault tolerance due to its interconnected nature. In a mesh network, all edges are connected to each other, forming a fully connected topology. This interconnectivity ensures that even if one node fails or a link breaks, there are alternative paths available for data transmission. Data can be rerouted through different edges, bypassing the failed node or link. This fault-tolerant characteristic of the mesh structure enhances the resilience and robustness of the system, allowing it to continue functioning even in the presence of failures. The tree structure, in contrast to the mesh structure, is more vulnerable to a single point of failure. In a tree-based network, nodes are connected hierarchically, with a single root node at the top and child nodes branching out from it. If the top node (root) or a critical link in the tree fails, it can have a cascading effect on all nodes in the network. The failure at the top affects the entire

hierarchy, requiring repair or restoration to ensure the normal operation of the network.

The large number of connections in a mesh structure leads to increased communication overhead. When a node sends a packet or message, it needs to transmit copies of that packet to all connected nodes. This redundancy is necessary to ensure that the packet reaches all intended destinations. Consequently, as the number of connections increases, the overall communication overhead grows significantly. The repeated transmission of packets to connected nodes during the communication process escalates resource utilization and can strain the system's capacity. With a high number of connections in a mesh structure, the overhead of maintaining and managing these links becomes a significant concern. Each connection requires monitoring, routing, and ensuring proper connectivity. As the number of nodes increases, the maintenance overhead becomes even more pronounced. The task of managing a large number of connections and associated overhead can pose challenges in terms of system resources, scalability, and complexity. The mesh structure presents a management burden in terms of setting up roles and relationships for each connection. With numerous connections, configuring and maintaining distinct roles and relationships for each one becomes complex. Defining the responsibilities of each node, coordinating communication flows, and ensuring efficient data exchange requires additional effort and resources. The management burden intensifies as the number of nodes and connections grows, potentially hindering the scalability and operational efficiency of the system.

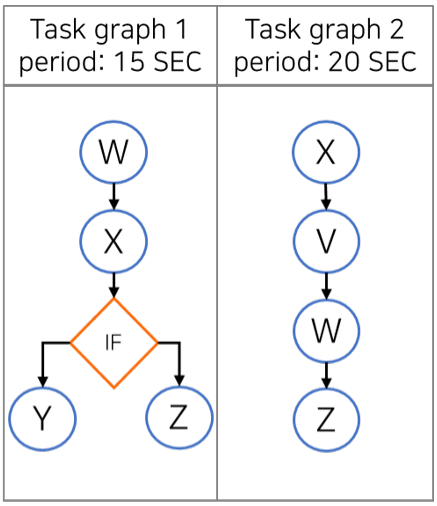
A remarkable advantage of the tree structure is that it provides well-defined roles and communication paths. Nodes are arranged hierarchically, with a single root node at the top and child nodes branching out from it. This hierarchical arrangement offers a clear structure with distinct roles assigned to each node. The root node typically plays a coordinating or controlling role, while child nodes have specific responsibilities within their subtree. This structural organization simplifies management and facilitates efficient communication and coordination within the system. Although the tree structure may involve

higher latency compared to direct communication due to the requirement of several hops, it offers a significant advantage in terms of the overall amount of communication. Since each node has only one parent (except the root), communication flows along predefined paths, resulting in a reduced volume of communication compared to a fully connected mesh structure. This reduction in communication amount translates to lower communication overhead and more efficient resource utilization. The tree structure requires less management and maintenance overhead compared to a mesh structure. With well-defined roles and communication paths, the complexity of configuring and maintaining connections is significantly reduced. The limited number of connections and clear hierarchical relationships make it easier to manage and maintain the system. This advantage becomes particularly valuable for constrained edge devices, which often have limited resources and processing capabilities.

Shortly, the tree structure offers structurally well-defined roles, lower overall communication volume, and reduced management and maintenance overhead. These advantages make the tree structure a suitable choice, particularly for edge devices with limited resources and the need for efficient communication and management.

3.4 Scheduling in Service-oriented IoT

The biggest difference between service-oriented IoT and device-oriented IoT is that middleware provides operating system-like scheduling. Applications can be seen as task graphs with a period, and the task graphs are mapped to registered things based on the device information that contains supporting tasks as well as the execution time and energy consumption. This is an increasingly important area of research today, as systems become increasingly complex and AI features with long execution times are becoming more widely used in current IoT platforms. Scheduling is broadly divided into service mapping, which is the delivery of service requests to devices, and execution scheduling, which is the timing and ordering of those mapped requests.



Device	Supporting Tasks	Execution Time	Energy Consumption
Device1	W	4	3
	X	3	7
	Y	7	5
Device2	X	2	9
	Y	5	7
	Z	6	4
Device3	W	3	5
	X	9	9
	V	6	2

Mapping Choice 1

Device	Mapped tasks	Util.	Energy
Device1	**W (Graph1)** Y (Graph1) X (Graph2)	**0.88**	**15**
Device2	X (Graph1) Z (Graph1) Z (Graph2)	0.83	17
Device3	V (Graph2) W (Graph2)	**0.45**	**7**

Mapping Choice 2

Device	Mapped tasks	Util.	Energy
Device1	Y (Graph1) X (Graph2)	**0.61**	**12**
Device2	X (Graph1) Z (Graph1) Z (Graph2)	0.83	17
Device3	**W (Graph1)** V (Graph2) W (Graph2)	**0.65**	**12**

Figure 3.6: An example showing the different mapping choices

Fig. 3.6 shows the example of mapping service requests to IoT devices in an edge. In the example, the mapping choice for which thing to perform each task can exist in a variety of ways. The two tables show two possible mappings. Depending on how the mapping is done, it shows different device occupancy and energy consumption. Furthermore, this result also affects the next application that comes in. Let's assume a new task graph consisting of task X of period 10 Seconds comes in in this situation. In the case of choice 1, no matter what thing is mapped, utilization exceeds 1, so mapping is not possible. However, if task W mapped to thing 1 was mapped to thing 3, a new task X could be mapped to thing 1 or thing 3. In this case, since thing 3 is more energy efficient than

thing 1 to perform X, the optimal result is when thing 3 is selected. In this way, a good mapping algorithm can add more applications and save energy. Even in such a simple example, it is not easy to find a mapping that optimizes the acceptance ratio and energy consumption while satisfying the given timing constraints.

Fig. 3.7 shows the example of scheduling IoT applications in an edge. Given two applications represented by a DAG of service nodes, service requests must be made in an order based on the dependencies of each node, and the execution time required for each device to perform the service is taken into account to populate the timeline. The order of service requests from the two applications may need to be adjusted when they conflict due to real-time variations in overhead or application-specific priorities. This real-time coordination of service execution requests mapped to each device at the right time is a critical part of scheduling.

This problem is challenging because it must reflect the characteristics of IoT systems. First, IoT systems must consider real-time requirements. Within each application, there may be dependencies between service requests, and as these applications run concurrently, their order may change continuously. In a situation like this, where there are multiple applications running that are organized as service nodes, the middleware needs to schedule each service request accordingly by allocating time to fulfill multiple requirements and when to do so. The more efficient this placement is, the better the performance of the entire system.

Second, there are devices with very different capabilities and features. They may take different amounts of time to provide the same functionality or consume different amounts of energy. Therefore, the scheduler can get good mapping results when it considers the characteristics of each device. This makes service mapping a very important technique in service-oriented IoT, as mapping that takes into account the characteristics and current state of the devices can provide real benefits, such as minimizing power or load balancing. Finally, unexpected events happen. IoT schedulers don't know when

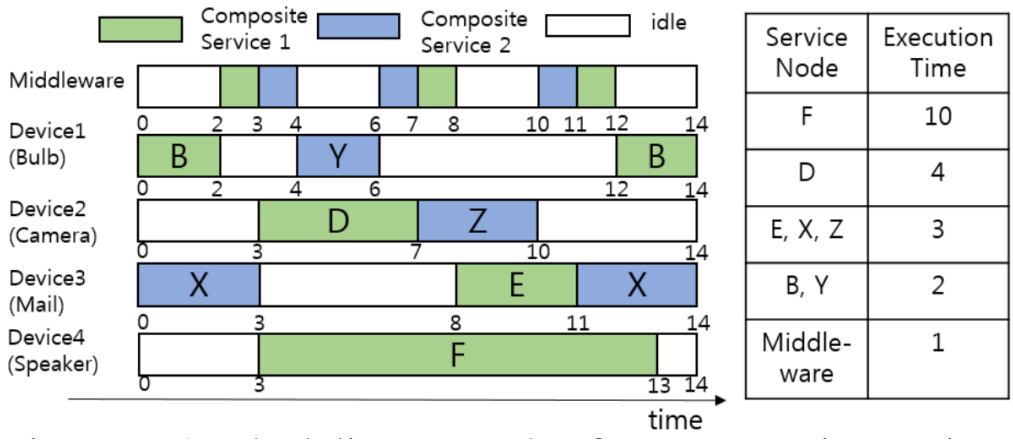


Figure 3.7: A scheduling example of two IoT applications

users will add or delete applications, and IoT devices can frequently fail due to insufficient specifications, battery life, or unstable environments. In a service-oriented system, the mapping function enables flexible response in this dynamic IoT environment. For example, if one camera fails unexpectedly, the middleware can switch the mapping to another device on its own in real time.

Since these scheduling problems become even more complex as we scale into the hierarchy, this is one of the main technical challenges in this work. The candidate mappings of services specified in the application now extend across the multiple edges in the hierarchical system. The scheduling algorithm needs to take into account who will make these decisions. It can be centralized, where the top node controls all service information and issues requests in chronological order, or it can be distributed, where the scheduling process is distributed to each node. As an example, how the information needed to make these decisions will be shared and maintained should be determined. When a composite service includes the services from many middlewares, the real-time state of the related devices in the edges is necessary for the scheduler to decide whether a mapping is possible or not.

The scheduler must handle unexpected events that occur within or in communica-

tion with other edges. This means that the information shared between edges can become invalid at any time, requiring additional work to validate it. Issues such as how to determine the validity of this information are also issues that need to be addressed in extended scheduling. We devise a distributed scheduling algorithm, which is described in Section 4.4.1, as a solution to these technical challenges.

Chapter 4

Proposed Hierarchical IoT Platform

4.1 Overview

In this chapter, we introduce a hierarchical architecture that addresses the design requirements and technical challenges associated with the requirements of a new edge-based IoT system. As illustrated in Fig. 1.1, the architecture incorporates an edge middleware at the bottom level, serving as a centralized middleware. The edge middleware is based on SoPIoT [14] described in Section 3.2. The primary function of this edge middleware is to facilitate the discovery and registration of IoT devices, while also providing an interface for the creation of user applications within the local area. It takes the responsibility of managing device and application states and establishes an IoT execution environment through its internal scheduler, constructing an independent IoT system. Notably, in order to safeguard the system against compromised or abnormal devices, device-to-device communication is strictly prohibited, as assumed in SoPIoT. Given the similarity of the proposed platform to conventional edge-based platforms, we omit specific details concerning the bottom-level middleware in this chapter. Instead, our focus lies predominantly on the hierarchical structure that enables the organization of the scalable system.

The architecture enables connectivity between multiple edges, enabling a hierarchical middleware structure. Child nodes establish connections with parent nodes, forming a

tree structure. To maintain connectivity, the middlewares periodically exchange live signals, with parents assuming greater authority in mediating communication among child nodes. The proposal of such a hierarchical middleware structure involves addressing several design requirements and related technical challenges, which can be considered technical contributions of this research. By tackling these problems, we aim to enhance the safety, scalability, and efficiency of edge-based IoT systems, paving the way for their effective operation in real-world scenarios.

4.2 Design Requirements

We first describe the design requirements that arose during the design phase since this thesis is about the study of a new architecture. Below are the design considerations of the proposed architecture in the construction of the tree structure of edge devices.

- How to assign the edges to the levels of the tree structure?
- How to request services that the other middleware can serve?
- How to maintain the privacy of the edge-based platform?
- How to map and schedule the service requests to devices in each middleware?
- How to determine if the newly added application can be served in the current IoT system?

Within this section, we describe our proposed structure addressing each design requirement through an explanation of our approach and the technical challenges we encountered one by one.

4.2.1 How to assign the edges to the levels of the tree structure?

Designing a tree structure with multiple vertically connected edge middlewares requires attention to the challenge of hierarchical connectivity. This involves considering

various factors, such as establishing and maintaining connections, as well as determining the permissions and roles of parent and child nodes.

In our proposed structure, each middleware operates as an independent node within the tree. This design allows us to set up the connection by a straightforward parent-child relationship, adopting the inherent geographic location of the edges. By organizing the tree structure based on spatial proximity, we enhance the efficiency and effectiveness of communication within the system, exploiting the benefits of the tree topology described in Section 3.3.

Whether the connection between a child node and its parent node is established over the local network or the Internet, it remains resilient until explicitly terminated or in the event of unresponsiveness. The parent node assumes a pivotal role as a privileged mediator, endowed with greater authority and responsibilities, facilitating seamless communication and coordination among the child nodes. This hierarchical approach not only ensures the swift exchange of data but also authorizes the parent nodes to distribute service requests and allocate resources, leading to a highly responsive and agile system.

Implementing the hierarchical middleware connection in our proposed platform poses no significant technical challenges. The seamless integration of parent-child relationships and the allocation of roles and permissions is trivial to implement.

4.2.2 How to request services that the other middleware can serve?

The question of resource sharing arises when there is a need to access services from devices residing in other edges within the middleware hierarchy. To address this, our proposed solution introduces the concept of a super service, strategically positioned at the common ancestor node in the tree structure. The super service acts as an arbiter, managing resource sharing by providing access to services residing in child middlewares. Super services are responsible for enabling resource sharing within the hierarchy. They

are defined by the parent middleware and invoked by the child middlewares. A detailed description of the implementation and functionality of super services is in section 4.4.1.

The introduction of new resource-sharing mechanisms brings forth a set of technical challenges. One such challenge is encountered when accessing resources from another middleware. The scheduler must consider the state of devices registered with that middleware to ensure non-conflicting access. Given that services can be dynamically added or removed from each node, and their execution state can vary dynamically (e.g., running, waiting), it becomes essential to assess the availability of services before making a request. This requires the scheduler to effectively track the current state of services and ensure that only available and appropriate resources are accessed.

Conversely, the structure must also be capable of processing resource access requests originating from other middlewares. These requests can be sporadic and unpredictable, adding complexity to the scheduler of the original system. The scheduler needs to accommodate these external requests and handle them to ensure effective resource sharing. This entails developing mechanisms for handling incoming requests, validating their compatibility with the local middleware, and service mapping to devices accordingly. Moreover, the scheduler needs to dynamically adapt to changing resource availability and prioritize conflicting requests, ensuring fair and efficient shared resource utilization.

Overcoming these technical challenges requires the design and implementation of sophisticated scheduling algorithms, resource management mechanisms, and communication protocols. The scheduler needs to integrate intelligent decision-making capabilities to assess resource availability, resolve conflicts, and optimize service mapping. These challenges are resolved by the proposed scheduler of this platform and the detailed explanation is in the Section 4.4.

4.2.3 How to maintain the privacy of the edge-based platform?

In the context of the proposed architecture, privacy concerns arise when higher-level middlewares have access to user data at lower-level edges. While edge-based IoT emphasizes the preservation of private information, unrestricted access by other edge middlewares to private devices can potentially violate the privacy of the edge owners. Therefore, it becomes imperative to devise a strategy to resolve this privacy issue clearly.

Our solution to this problem revolves around granting users the ability to determine the visibility of each service. Drawing inspiration from the principles of object-oriented programming, we empower users to exercise control over the access privileges of their services. By designating services as either *public* or *private*, users can selectively hide certain services from the hierarchical structure. In this manner, the upper-level middleware can only access the *public* services of lower-level middlewares, while *private* services remain concealed. A comprehensive explanation of this functionality can be found in the detailed description of the super service concept.

Addressing privacy concerns necessitates the establishment of well-defined rules and guidelines. However, from a technical standpoint, implementing this requirement poses no significant challenges. We set an additional property to the service object and make the visibility of each service according to it. With this simple extension, The system can readily accommodate user-defined access privileges, ensuring that privacy is respected and user data remains secure within the edge-based IoT environment.

4.2.4 How to map and schedule the service requests to devices in each middleware?

In service-oriented IoT systems, the middleware plays a pivotal role in the mapping and scheduling of services provided by registered devices. This requirement is also true for the extended middleware structure. To maintain the benefits of a service-oriented system, a scheduling algorithm for the proposed platform is needed to ensure that any kind

of service requests, including super service requests, are available to be scheduled. With the introduction of the super service concept, the complexity of this scheduling task is further amplified. Mapping service requests to a device extends beyond the confines of local space, encompassing devices across different edges. We devised a distributed algorithm that schedules not only local services but super services to address this requirement, of which detail is explained in the following sections.

The presence of super services presents new and complex technical challenges for edge middleware schedulers. First, if a failure occurs in the middle of the scheduling process, the progress up to that point should be rolled back. Since a super service can be composed of services from different edges, the schedulability of the services is checked individually. The super service is schedulable only when all its component services are schedulable. Therefore, it is necessary to schedule the super service in two stages, using temporal copies and verifying each scheduling result when the final result is available.

Next, even after scheduling has finished, unexpected device failures at other edges can invalidate previous mapping results. To maintain the integrity of the scheduling process, it is important to have the ability to immediately notify the scheduler at the other edge when a device failure occurs and for the scheduler to react quickly once informed.

Another challenge arises when responses arrive differently than expected during the scheduling process. The scheduler must determine whether the lack of response indicates a malfunctioning middleware or a delay in processing. This decision is crucial, as it impacts system performance, especially in scenarios with poor communication conditions. Developing mechanisms to accurately assess response reliability and appropriately adapt the scheduling process is essential.

These technical challenges are resolved by the development of a robust scheduling algorithm that includes two-phased scheduling and real-time fault-tolerant handling techniques in this platform. The scheduler exploits a temporal copy in the process, adapts to failures, assesses response reliability, and efficiently manages the scheduling process in

a dynamic, hierarchical, service-oriented IoT environment.

4.2.5 How to determine if the newly added application can be served in the current IoT system?

To realize the super service in the proposed platform, the scheduler of each middleware can determine the schedulability when the super service is specified in an application. The schedulability of the service should be considered based on the service provider's state information, which is registered in the local middleware. The decision could be made in the local node or the central (*root* node). In this platform, we adopt a distributed approach the schedulability of the local service is only determined by the owner middleware.

The mapping process in the context of super service necessitates the establishment of a governing entity responsible for mapping service requests to devices. The middleware assumes the role of a decision-maker, determining the optimal mapping based on various factors, such as device capabilities, availability, and performance metrics. Furthermore, robust mechanisms for storing and managing the information relevant to these mapping decisions must be in place.

To govern the mapping process effectively, the middleware requires access to comprehensive and up-to-date information about the devices and their associated services. This information can include device specifications, service capabilities, location data, and performance metrics. Efficient storage and management of this information are vital for enabling dynamic mapping decisions.

Accessing information of the other schedulers poses a technical challenge in a distributed service-oriented IoT environment. It is impractical for every middleware to maintain the real-time state of all devices in other middlewares, as it would result in information overload and resource burden. Therefore, it is important to set an efficient and scalable approach to gather information from edges while distributing the throughput across

the available edge computing resources.

In our proposed structure, we tackle this challenge through an on-demand approach. The super service plays a key role in this process by selectively requesting presence information from each middleware as needed. This ensures that the scheduler obtains up-to-date device states without overloading the system. Adopting an on-demand information retrieval mechanism optimizes resource utilization and minimizes the overhead of maintaining real-time information. By the on-demand approach and establishing appropriate access rules, our proposed structure achieves efficient resource sharing, enhances system performance, and ensures the privacy preservation of the service-oriented IoT environment. Further details regarding this approach can be found in Section 4.4.1.

Additionally, as the system extends to a tree structure, the scheduling process can be triggered by various events at different points in time. The diversity of events requiring scheduling decisions introduces further complexity.

To effectively handle these dynamic events and meet real-time constraints, such as latency and execution period, we compose the scheduling algorithm with the various event handling logics, called scheduling policy, and let developers be able to tune more precisely. The algorithm is explained in Section 4.5.

4.3 Super Service

In the proposed IoT platform, the second and third questions are addressed through the concept of super services. A super service represents a specialized composite service that can access the public services offered by child middlewares within the system. An illustrative example is the 'find_all' service depicted in Fig. 1.1, implemented as a Python script featuring a function called *request()*. This function allows access to the *find* service offered by all child middlewares.

To enable the execution of super services, a virtual entity called a "super thing" is created for each super service and registered with the middleware responsible for its

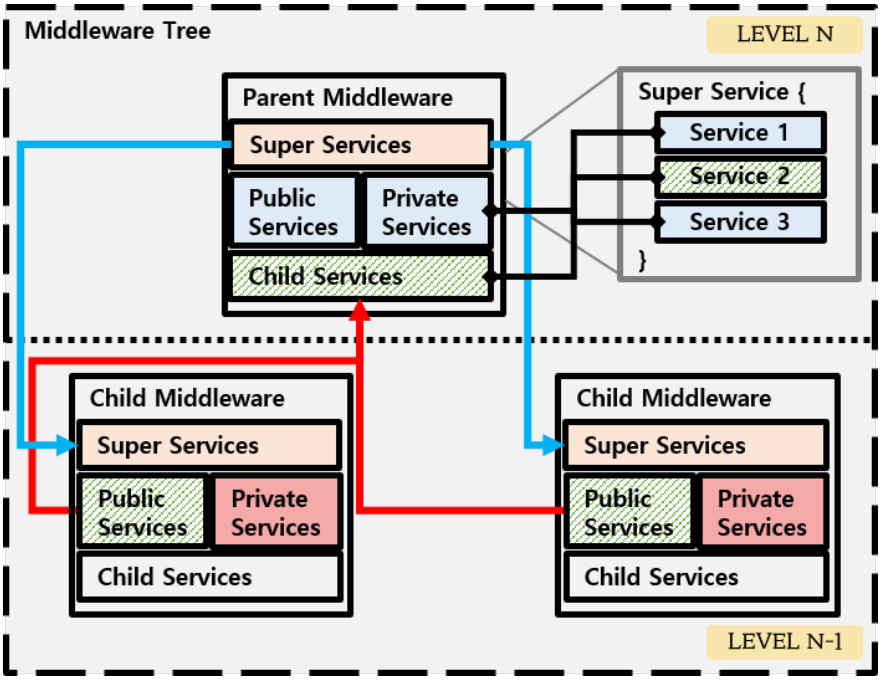


Figure 4.1: Service list shared between middlewares in the middleware tree.

management. By default, all services are designated as private, limiting their accessibility to the local space. However, users have the option to declare certain services as public, thereby making them shareable with upper-level middlewares. The list of public services is transmitted to the parent middleware, which gains the ability to access and utilize these services within the context of a super service. The decision to declare a super service as public or private determines its visibility to other middlewares. Figure 4.1 provides a visualization of how the service list is shared within the hierarchy. This shared list contains minimal information, such as service names, while each middleware retains the responsibility of managing detailed properties and runtime information for the devices registered within it.

In Fig. 1.1, the *find* service at the lowest level is declared as public, ensuring its visibility to all upper-level middlewares within the hierarchy. Similarly, the *find_all* service is declared as public, enabling User App 2 to access the service at the bottom level.

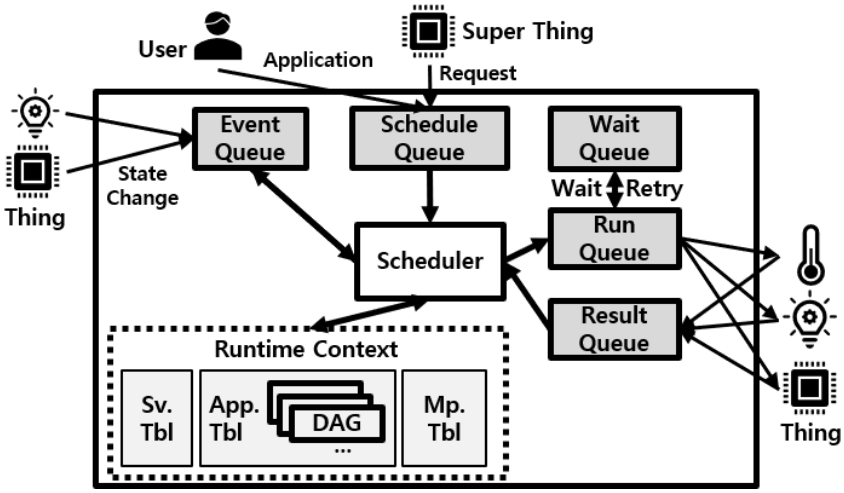


Figure 4.2: Overview diagram of the scheduler

Notably, middlewares at the same level are unable to access public services offered by other middlewares. Resource and information sharing within the hierarchy is exclusively achieved through super services.

It is important to note that when specifying an IoT application, public super services are indistinguishable from local services, as demonstrated in line 2 of User App 2 in Fig. 1.1. However, it is advisable to exercise caution when employing super services, as they entail significant communication overhead and resource utilization across different levels of the hierarchy. The use of super services should be minimized unless absolutely necessary, as this approach mitigates the risk of performance bottlenecks within upper-level middlewares and preserves the privacy of the local space.

By leveraging the concept of super services, the proposed IoT platform offers a flexible and controlled mechanism for accessing and utilizing services across the hierarchical structure, ensuring efficient resource management, privacy preservation, and optimized performance.

4.4 Scheduler

To answer the fourth and fifth questions above, a scheduler is developed whose overall structure is shown in Fig. 4.2. It maintains two tables that contain the run-time context information required for scheduling as follows:

- Service state table (Sv. table in Fig. 4.2): This table contains the run-time information about the current state and the execution time of a service on each local device that can execute the service as shown in Table 4.1. Available super services from the upper layer that an application can request are also registered in this table. While the execution time is set during the registration process of the device, the current state of the service varies dynamically at run-time and will be updated by the scheduler. The state transition diagram of a service is displayed in Fig. 4.3. Note that the table can be extended to include more information, such as energy consumption or battery lifetime, by adding more columns. That information, as well as execution time, can be used to determine the objective function of the scheduling problem.

- Application state table (App. table in Fig. 4.2): As shown in Table 4.2, this table contains information about the current state, name, and period of each IoT application. The period information is used to check if the target device can accept the additional service based on the current utilization. The services requested in an application and their dependency are stored separately as a directed acyclic graph (DAG), and the table contains a pointer to the DAG information. The run-time state of an application is managed by the scheduler at run time. The state transition diagram is shown in Fig. 4.4. If the mapping fails, the service is not available, or the requested real-time constraints cannot be met, the application is rejected. Once accepted, it goes into the *initialized* state and is moved to the *running* state if it is time to run. If an error or time-out event occurs during the execution of the applica-

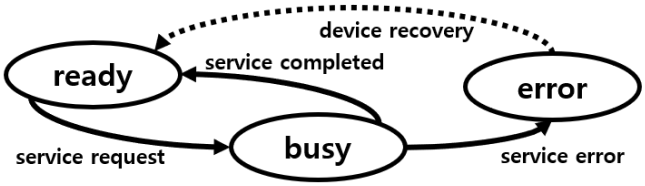


Figure 4.3: State transition diagram of a service

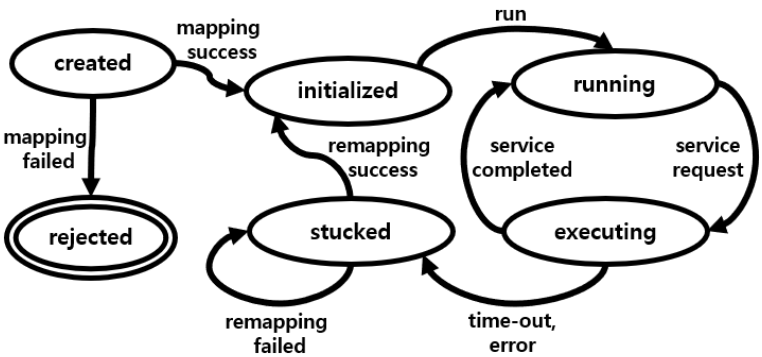


Figure 4.4: State transition diagram of an application

tion, it is stuck. The scheduler tries to remap the stuck application until remapping succeeds.

In addition, we maintain a mapping table (Mp. table in Fig. 4.2) that stores the mapping result of service requests to devices. Table 4.3 shows the internal structure of the mapping table. During the scheduling process, the content of this table is updated, and when the service is executed, requests are sent to the mapped devices based on this table. Since the same service can be requested multiple times in an application, the second attribute of the key element is used to distinguish the same services.

Table 4.1: Service state table: an example

Key	State	Energy	Execution Time	...
th1.sv1	READY	500mJ	100ms	...
th1.sv2	BUSY	150mJ	50ms	...
th2.sv1	ERROR	1000mJ	200ms	...
...				

Like the conventional scheduler of a computer system, the scheduler manages multiple queues for various types of events. There are two input queues to the scheduler. One is the schedule queue that stores the mapping requests from the user or the super thing. When the user adds an application or requests remapping, we put the mapping request into the queue. The mapping request from a super service is also put into this queue. Another input queue is the event queue that manages the system events caused by device addition or removal, device failure, service execution errors, etc. The corresponding handling logic will be triggered for each event by the scheduler.

The scheduler identifies the service requests that need to be mapped now from the schedule queue and performs mapping. When we map a service request, we check the schedulability of the request. If mapping is successful, they are put into the run queue. For the fetched service request from the run queue, the scheduler first checks the status of the service by looking at the service state table. If the mapped device of the service is ready, it sends the service request to the device and changes the state from ”ready” to ”busy”. If the mapped device is busy, the service is moved from the run queue to the wait queue. After the device returns the completion of the requested service, the service is put into the result queue. After completion of the service, the scheduler fetches the next service to run from the same application and puts it into the run queue. Also, it changes the state of the corresponding service in the service state table and moves the waiting service on the device from the wait queue to the run queue. In case an error is detected during the service execution on the device, a fault event is put into the event queue.

Table 4.2: Application state table: an example

Key	State	Period	DAG
app.1	RUNNING	1s	ptr
app.2	STUCKED	10s	ptr
app.3	CREATED	5s	ptr
...			

Table 4.3: Mapping table: an example

Key	Type	State	Mapped Things	...
app1.idx1.sv1	Local	READY	th1, th2	...
reqkey.sv2	Super	DONE	th3	...
app2.idx1.sv1	Local	ERROR	th4, th5	...
...				

4.4.1 Scheduling of Super Services

The execution of a super service in the proposed IoT platform requires collaboration between multiple middlewares, involving tasks such as mapping and execution. If a super service requests two services from descendant middlewares, the upper-level middleware responsible for executing the super service must verify if each service can be executed by the respective descendant middleware.

Scheduling a super service is contingent on all the required services being executable within the descendant middlewares. In cases where any of the services cannot be accommodated by the descendant middlewares, scheduling the super service becomes unfeasible. The absence of a centralized server to monitor the runtime status of all middlewares poses a significant challenge in scheduling super services. To address this challenge, we propose a distributed scheduling technique. Each middleware takes responsibility for the devices registered within its own domain and communicates with other middlewares as needed to support super services. By adopting this distributed scheduling technique, the proposed IoT platform enables efficient and decentralized scheduling of super services. It ensures that scheduling decisions are made in a distributed manner, exploiting local knowledge and promoting effective collaboration among middlewares. This approach enhances scalability, flexibility, and system robustness, enabling the execution of complex composite services that span multiple middlewares.

4.4.1.1 Mapping of super services

The first thing we need to do to schedule a super service is to map the internal services that make up the super service to the devices that are attached to the various middleware. At this point, we need to deal with the fact that there may be failures during the mapping process, as there is no one central server overseeing the entire process. For example, if one of the services in the super service mapping process does not have a mappable device, the super service's mapping request will be rejected, and the previously successful mapping will no longer be valid. Therefore, instead of writing directly to Table 4.3 during the mapping process, a temporary copy is kept and applied when the final mapping results are available.

In the proposed technique, the mapping of a super service consists of two phases as depicted in Algorithm 1: checking and confirming. Mapping of a super service is initiated by request from a middleware to the middleware where the super service is registered (line 5). It is triggered when a middleware adds an application calling the super service. In the checking phase, the middleware that receives the request asks child[1] middlewares if they can schedule each service that is contained in the super service (lines 7-10). The child middleware checks the scheduling possibility by referring to the service state information and the current mapping information of their registered devices. In this step, the scheduler performs an algorithm that considers the current utilization of the devices and the execution period of the applications, and so on. If there is residual utilization on at least one device that can accept the requested service, it is determined to be schedulable. If the scheduler judges that it can schedule the service, it replies positively. If no child middleware exists to schedule the service, the super service cannot be scheduled (line 9). Otherwise, the home middleware makes a separate mapping table for the super service to record which middlewares can schedule the constituent services.

[1]We call a descendant node in the tree structure as a child node for brevity even though a descendant node is not a direct child.

Only after the checking phase is completed successfully, does the confirming phase start (lines 12-14). For each service in the super service, the home middleware of the super service maps the service to one or multiple child middlewares that answered positively in the checking phase according to the range specifier. Another customizable policy is which child middleware the super ting chooses when the range specifier is single and there are multiple possible mapping candidates. There may be other options, such as selecting them in order of proximity to home middleware, or selecting them in order of historical responsiveness, but in this work, we leave this optimization as a future work and instead select them randomly. The selected child middleware puts the service mapping request to the schedule queue to determine which device will serve the service request after this mapping process as depicted in Fig. 4.2. The child middleware notifies the mapping completion to the super thing after the local mapping decision. After the mapping of all services in the super service is completed, the home middleware sends the mapping result to the middleware that runs the application (line 16).

Mapping a super service as above is not a trivial task. For the mapping of a super service, a parent middleware should communicate with each child middleware twice, once for checking the schedulability and once for actual mapping. As the number of services involved in the service increases and the number of levels between the home middleware of the super service and the bottom middleware increases, the communication overhead increases proportionally. Another overhead to pay for super service mapping is the computation overhead of the local scheduler for checking the scheduling possibility before mapping.

4.4.1.2 Execution of super services

The execution of a super service is a straightforward process, as outlined in Algorithm 2. When an application requests the execution of a super service, the home middleware initiates service execution by sending execution requests to the mapped middle-

Algorithm 1 Super Service Mapping Algorithm

Input : SS, the super service to be mapped
S: a set of services that should be mapped to execute SS
A: an application (composite service) that contains SS as a constituent service
C: a middleware that runs A, which is located to a lower layer
P: home middleware where SS is registered, which is located on an upper layer
C requests SS mapping to P in mapping process of A
P starts SS mapping
for each $s \in \mathcal{S}$ **do** ▷ *Checking Phase*
    P asks child middlewares if they can map s
    Each child checks the mapping possibility and returns
    If no child can map s, return "Fail"
end for
for each $s \in \mathcal{S}$ **do** ▷ *Confirming Phase*
    P maps s to the child middlewares that answered yes
    The mapped childs confirm device mapping for the request and respond to P
end for
P sends the mapping result to C

wares, referencing the mapping table associated with the super service. From the perspective of the local middleware, the service request from a super service is indistinguishable from the service request originating from IoT applications.

Given that the devices responsible for executing the service requests are determined during the mapping process, the scheduler's role in the execution phase is to efficiently deliver the service requests to the corresponding devices based on the mapping table. This routing workload is relatively trivial, resulting in minimal computing overhead even for edge devices. The execution process follows the same protocol as described earlier, where the mapped middleware verifies the runtime status of the corresponding device to ensure its availability. Upon confirmation, the service request is promptly executed, or if necessary, the mapped middleware patiently awaits the device's availability before proceeding. The outcome of the service execution is relayed back to the mapped middleware and subsequently forwarded to the home middleware of the super service.

Upon receiving the outcomes of all service requests within the super service, the home middleware consolidates the results and transmits the final outcome to the middle-

Algorithm 2 Super Service Execution Algorithm

Input : SS, the super service to be mapped
1: S: a set of services that should be mapped to execute SS
2: A: an application (composite service) that contains SS as a constituent service
3: C: a middleware that runs A, which is located to a lower layer
4: P: home middleware where SS is registered, which is located to an upper layer
5: CM_s: child middlewares where s is mapped
6: C requests SS execution to P
7: P starts SS execution
8: **for each** $s \in \mathcal{S}$ **do**
9: P asks CM_s to execute the mapped services
10: CM_s check the availability of the services.
11: CM_s execute the services or wait for the devices based on the availability (async)
12: CM_s send the execution results to SS (async)
13: **end for**
14: P sends the final result to C

ware that initially requested the super service. In addition, in scenarios where a service needs to be executed across multiple middlewares, parallel execution can be employed since there are no execution dependencies among them.

By following this execution process, the proposed platform ensures efficient and reliable execution of super services and makes the best use of the mapped middlewares' capabilities to distribute and execute service requests seamlessly.

4.5 Scheduling Policies

How to schedule the services is the key technical challenge of the scheduler in each middleware. Since IoT applications usually have real-time constraints such as latency or throughput, the scheduling problem is similar to the real-time scheduling problem of dependent tasks on a heterogeneous multi-processor system since an IoT application can be modeled as a DAG as explained in [14]. This problem is recognized as very difficult to solve even when the application characteristics are known a priori. However, in an IoT system, an IoT application without any prior information can be added dynamically, and we need to determine which device to perform each service included in the application.

Hence it is inevitable to schedule the added application incrementally without changing the mapping results of existing applications [42].

When we make a scheduling decision, we need to evaluate the performance according to the objective functions that are given by the user. If reducing the energy consumption is an objective function, it is good to prioritize the devices with low energy consumption first and map each service to an available device that has the lowest energy consumption while satisfying the real-time constraint.

Another feasible objective function is to maximize the quality of service (QoS) by reducing the response time of some selected IoT applications. Then we need to use a different scheduling technique that aims to optimize the QoS metric. Since the scheduling result by incremental scheduling is far from the optimal scheduling solution, it is desirable to perform global re-mapping in a background process [42]. How to perform a global remapping is beyond the scope of this work.

There is another key difference between the IoT scheduling problem and the conventional task scheduling problem. In an IoT system, the status of the device that is supposed to execute a service may vary dynamically. The device may not be available temporarily or permanently due to various reasons, such as battery discharge, breakdown, occupation by other services, and so on after the mapping decision is made. Quickly recovering from these situations is also part of the IoT scheduling problem.

In addition, the scheduler should properly handle various events that may come from the user or the system at run-time. Since finding an optimal mapping is not feasible at run-time, we propose to handle each event with a proper scheduling policy. Table 4.4 lists 15 events that are considered in the current implementation of the scheduler and the associated scheduling policies that handle the events. The proposed scheduling policies are not claimed to be optimal but serve as reference policies in future studies. In the following, we discuss more details on how to handle the events on four event types: device status change, execution request, execution result, and timeout.

4.5.1 Handling Events on Device State Changes

When a device is removed from the system, it renders the local applications or super service requests mapped to that device invalid. In such situations, it is preferable to actively seek alternative candidates capable of executing the same service and remap them accordingly. By doing so, the corresponding application or super service request can be maintained, ensuring continuity and avoiding the need for outright abandonment.

This remapping process should be incorporated into the system's event handling mechanism, which addresses various events such as device failures or unregistration in both local devices and external middleware. The scheduler needs to promptly respond to these events and initiate the remapping procedure to mitigate the impact of device unavailability.

Conversely, when a middleware or device is newly registered within the system, there are two potential approaches to consider. One option is to retain the existing mappings if they remain valid and continue serving their intended purposes effectively. Alternatively, remapping can be employed to optimize the objective function by considering the presence of additional mapping candidates. This approach allows for the consideration of newly registered devices or middleware components to potentially improve overall system performance.

By carefully managing the remapping process in response to device removals, failures, unregistrations, and the introduction of new devices or middleware, the scheduler can maintain operational efficiency, resource utilization, and the uninterrupted execution of services. Effective remapping strategies play a vital role in adapting to dynamic changes within the system and optimizing its overall functionality and performance.

4.5.2 Handling Events on Execution Request

When the scheduler processes service requests in the run queue, it engages with the corresponding mapped devices in a systematic manner. For local service requests,

the scheduler ensures the device's availability before initiating the service request. In scenarios where multiple applications target the same device, aggregating these requests into a single request can enhance performance optimization. However, if the device is currently occupied, the service request is temporarily placed in the wait queue, allowing for subsequent attempts.

In the case of service requests originating from a super service, which can consume a long time, it is preferable to provide an estimated worst-case waiting time to the upper middleware rather than subjecting the request to indefinite waiting in the queue. This information empowers the middleware to decide whether to enqueue the service request or explore alternative child middlewares for execution. The estimation of worst-case response time considers factors such as the services already in the wait queue and the service state table. Although this strategy may introduce additional communication overhead for super services during runtime, the current implementation enqueues the service request and triggers a time-out event if the response is not received within a predefined time threshold.

The handling of events on execution requests is an important aspect of the scheduler's role in managing service execution in an IoT system. By considering device availability, merging requests for performance optimization, and providing estimated waiting times, the scheduler optimizes resource utilization and ensures the timely processing of service requests.

4.5.3 Handling Events on Execution Result

The proper processing of service results in the complete queue is an important responsibility of the scheduler within the IoT system. When the execution of a service is successful, the scheduler proceeds to transmit the results to the application or super service that initiated the service request. In cases where multiple requests have been merged, the scheduler ensures that the results are delivered to the respective applications in a

suitable manner. This facilitates efficient communication and enables the applications to make use of the obtained results effectively.

However, in the event of an error during service execution, the application enters a *stuck* state, as depicted in Fig. 4.4, indicating the need for intervention. In such scenarios, the scheduler plays a critical role in attempting to remap the faulty service to another available device, thereby enabling continued service provision. The scheduler undertakes a thorough examination of the error's underlying cause to facilitate appropriate remedial actions.

If the error stems from a device malfunction or unavailability, the device may be removed from the mapping table, prompting the scheduler to remap all applications that rely on that device in subsequent operations. However, it is worth noting that when dealing with services requested from a super service, remapping actions have the potential to introduce significant delays in service execution, possibly leading to indefinite delays. In such cases, it may be more advantageous to promptly report the service failure to the home middleware, ensuring timely acknowledgment of the issue.

The handling of error cases within scheduling policies can significantly impact the performance of the IoT system. An effective error-handling approach improves system robustness, resilience, and reliability. It minimizes the impact of failures, reduces downtime, and facilitates the efficient utilization of available resources. On the other hand, inadequate error handling can result in prolonged service disruptions, decreased system performance, and dissatisfied users or applications. Therefore, devising a well-designed error-handling strategy within scheduling policies is important to ensure better performance and overall system efficiency in the context of an IoT environment.

4.5.4 Handling of Timeout Events

To ensure adherence to real-time constraints, it is necessary to establish a maximum time duration from the initiation of a service request to its completion. In the event that

the service exceeds this predefined duration, a timeout event is triggered to indicate the inability to meet the specified time constraints. However, determining an appropriate timeout value for each constituent service poses a considerable challenge, particularly when real-time constraints are imposed on an application or a super service comprising multiple services.

If excessively lengthy timeout values are assigned to each service, there is a risk of failing to satisfy the latency constraint of the overall application. Conversely, if the timeout values are set too short, even minor and temporary delays could lead to the cancellation of requests. It is evident that an optimal approach is required to strike a balance between these competing factors.

One possible approach is to distribute the latency constraint proportionately among the constituent services, thereby ensuring a fair allocation of time resources. Another strategy involves initially setting the timeout values conservatively, considering worst-case scenarios, and subsequently adapting them dynamically as the service continues to be requested. The specific methodology for establishing these timeout values remains a topic for future investigation and research.

In the event of a timeout occurrence, the scheduler can undertake certain actions, such as retrying the execution request or modifying the mapping of the affected service. These decisions may depend on the specific characteristics of the application, the nature of the services involved, and the desired trade-offs between responsiveness and reliability.

4.6 Test-bed: Toy-level Smart Campus

A toy-level test-bed has been built on our campus as a proof-of-concept implementation of the proposed IoT platform. As shown in Fig. 4.5, a 3-level hierarchical middleware structure is formed with 2 buildings and 4 rooms. A local server for collecting sensor data is installed at each building and the campus level. A total of 40 IoT devices are distributed, as depicted by icons in the figure. In particular, the office room has a

homemade smart watering system for potted plants using Arduino-controlled valves and pumps. In addition to common IoT applications such as temperature control and automatic power switches, the following customized IoT applications[2] are installed in this test-bed.

- When a fire is detected in any room, the IoT system activates an alarm sound simultaneously in all rooms, ensuring that occupants are promptly alerted to the potential danger.
- To prevent the spread of COVID-19, access to the student room is carefully monitored using a thermal imaging camera. This allows for the screening of individuals entering the room and helps identify anyone with an elevated body temperature, enabling appropriate measures to be taken to mitigate the risk of infection.
- The IoT system intelligently detects when people gather in the meeting room and automatically activates the TV screen. This ensures that presentations, discussions, or other multimedia content can be easily shared and viewed by the attendees without the need for manual intervention.
- As a security measure, if any movement is detected in the premises after 10 p.m., the IoT system promptly notifies the manager. This enables them to take appropriate action and ensure the safety and security of the facility during non-operational hours.
- In the student room, an innovative feature of the IoT system notifies students via email whenever a professor enters the room. This real-time notification allows students to be aware of the professor's presence and facilitates effective communication and engagement.

[2]Note that we do not claim that all applications are useful. They represent various kinds of cooperation between devices or middlewares.

- The IoT system incorporates automated features to maintain the well-being of plants in the office room. It periodically waters each pot, ensuring they receive adequate hydration. Additionally, if necessary, the system also activates plant lights to provide optimal lighting conditions for plant growth.

- When the water level in the water tank falls below a certain threshold, the IoT system generates a signal to notify the manager. This timely alert ensures that the water tank is refilled promptly, avoiding any disruption to the watering system and ensuring continuous supply.

- The IoT system actively monitors the air quality in the room and utilizes an air conditioner to maintain optimal conditions. If necessary, such as in the event of poor air quality, the system alerts the manager to ventilate the room, promoting a healthier and more comfortable environment for occupants.

Note that the majority of the applications mentioned above rely on the utilization of super services. Super services can be classified into two primary types, as illustrated in Fig. 4.6.

In the first use case, administrators create public services at the upper level and make them available to end users. End users can then develop applications utilizing these super services at the lower level. Examples of such super services include notifying the manager in case of an event, such as a fire alarm, or informing the student when a professor enters the room. Additionally, a secure resource brokerage service can be implemented, allowing users to access resources from other users through the system's managers.

The second type of super service pertains to administration services that are not accessible to end users. It is important to note that the assumed IoT system operates under a single administrating entity. At the upper-level middleware, a super service can be defined to manage lower-level edge devices, such as signaling alarm signals or periodically collecting specific information. This enables the creation of a management service that

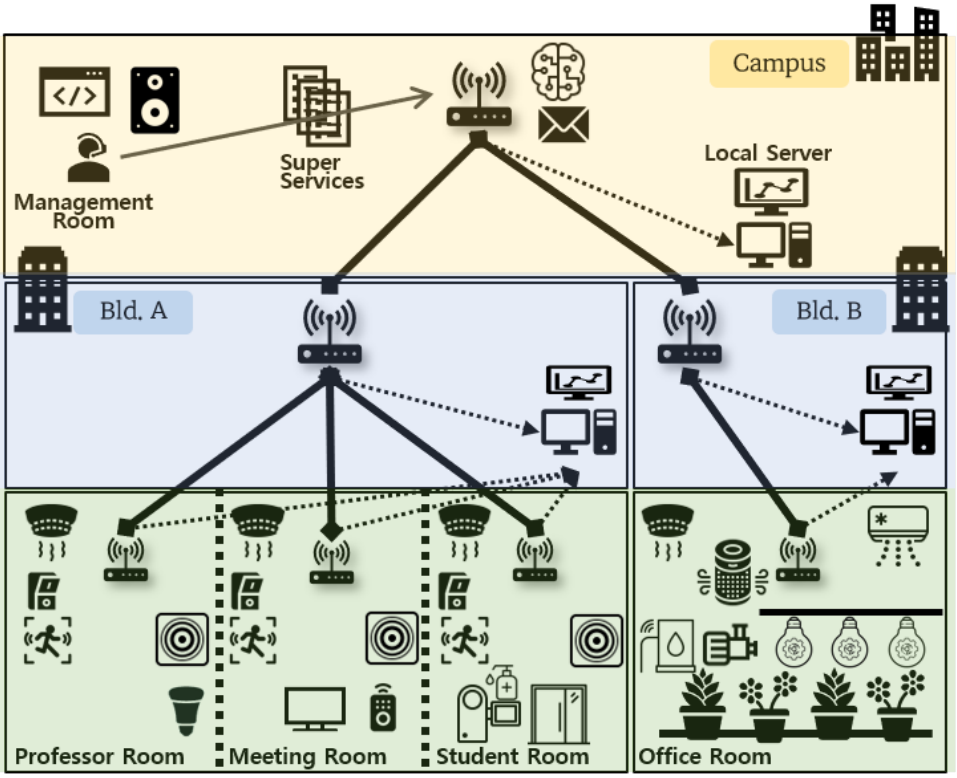


Figure 4.5: Smart Campus

administrators can utilize to effectively manage the IoT system. For instance, broadcasting the fire alarm to all rooms or checking the water level of the water tank are examples of super services falling under this category. By leveraging the concept of super services, the IoT system can provide enhanced functionality, facilitate user interactions, and enable efficient system administration and management.

Fig. 4.7 provides a visual representation of the sensor data collected from the smart watering system over a span of five days in the smart campus test-bed. The graph reveals a gradual decrease in the moisture level of the pot over time. To maintain the desired moisture level, the system activates pumps and valves whenever the moisture level reaches a predefined threshold. This automated replenishment process ensures that the plant receives adequate hydration. The second graph illustrates the water level in the water tank.

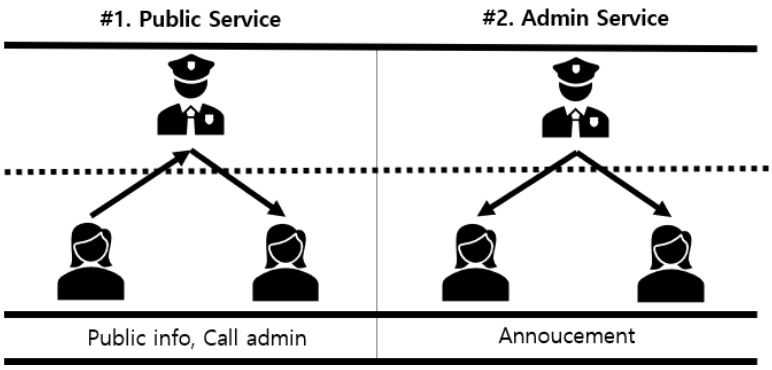
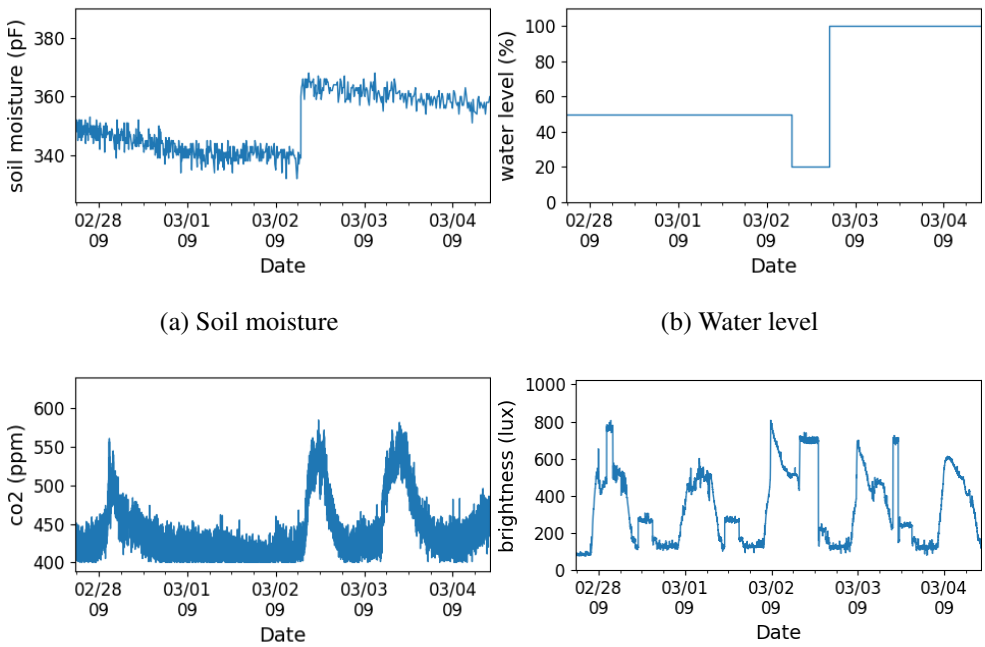


Figure 4.6: Two use-cases of Super Service

It shows that when the water level decreases significantly due to water activities, a notification signal is triggered to alert the manager to refill the tank. Interestingly, it can be observed that on the following day, the manager visited the office and refilled the water tank.

Furthermore, the sensor data indicates that the office room has low brightness and a relatively low CO_2 level in the morning. This is likely due to fewer people being present in the office during that time. However, as the day progresses and the light bulbs are turned on at night, the brightness level gradually increases, reaching approximately 300 lux.

Overall, the collected sensor data serves as a testament to the validity and effectiveness of the IoT platform implemented in the smart campus test-bed. It shows the hierarchical structure's ability to gather real-time data, facilitate autonomous control, and provide insights for informed decision-making and optimization. This validation enhances confidence in the platform's reliability and functionality, paving the way for broader applications and advancements in the IoT domain.



(a) Soil moisture

(b) Water level

(c) CO2 level

(d) Brightness

Figure 4.7: Sensor data collected in the smart campus test-bed

Table 4.4: Events to be handled by scheduling policies

Type	Event	Triggered By	Scheduling Policy
Scheduling Request	OnScheduleApplication	User	Map services in the application
	OnCancelApplication	User, Super Thing	Cancel and leave as is or update
	OnUpdateApplication	User, Scheduler	Re-schedule the application
	OnGlobalUpdate	Scheduler	Re-schedule all of applications
	OnSuperScheduleRequest	Super Thing	Schedule the service request
	OnSuperCancelRequest	Super Thing	Cancel and leave as is or update
Device Status Change	OnThingRegister	Thing	Leave as is or update
	OnThingUnregister	Thing	Re-schedule invalidated applications
	OnMiddlewareRegister	Other middleware	Leave as is or update
	OnMiddlewareUnregister	Other middleware	Re-schedule invalidated applications
Execution Request	OnServiceRequest	Scheduler	Ready: execute Busy: wait
	OnSuperServiceRequest	Super Thing	Ready: execute Busy: wait
Execution Result	OnServiceResult	Thing	Success: proceed Error: update
	OnSuperServiceResult	Thing	Success: reply Error: update
	OnServiceTimeout	Scheduler	Retry or update

(a) Overall view of exhibition room

(b) Smart garden

(c) Entrance keeper

(d) Room monitoring system

Figure 4.8: Exhibition Room

Chapter 5

IoT Simulation Framework

The performance of the scheduler depends on the scheduling policies that determine how to handle the events listed above. A simulation framework is developed to modify the scheduling policies and evaluate the performance of the modified scheduler in the proposed IoT platform. Since the performance of a scheduler also depends on the system configuration and varying the configuration of a real IoT system is not feasible, an IoT system simulator is developed.

5.1 IoT System Simulator

The purpose of the IoT system simulator is to evaluate the technique of the proposed platform, such as super service and scheduling policies that are included in the scheduler. We need to verify the actual performance of the super service and how effective the scheduling policies are in handling each event when they are made customizable. To choose a scheduling algorithm, one must comprehend its effectiveness, which can only be ascertained via real-world testing. Predicting outcomes can be arduous, especially when confronting unforeseen circumstances. Hence, we have developed a hardware-in-the-loop simulator to tackle this problem.

In the proposed simulator, we use real edge devices that run the middleware in the IoT system simulator, while other IoT devices are modeled as *virtual* things that are ex-

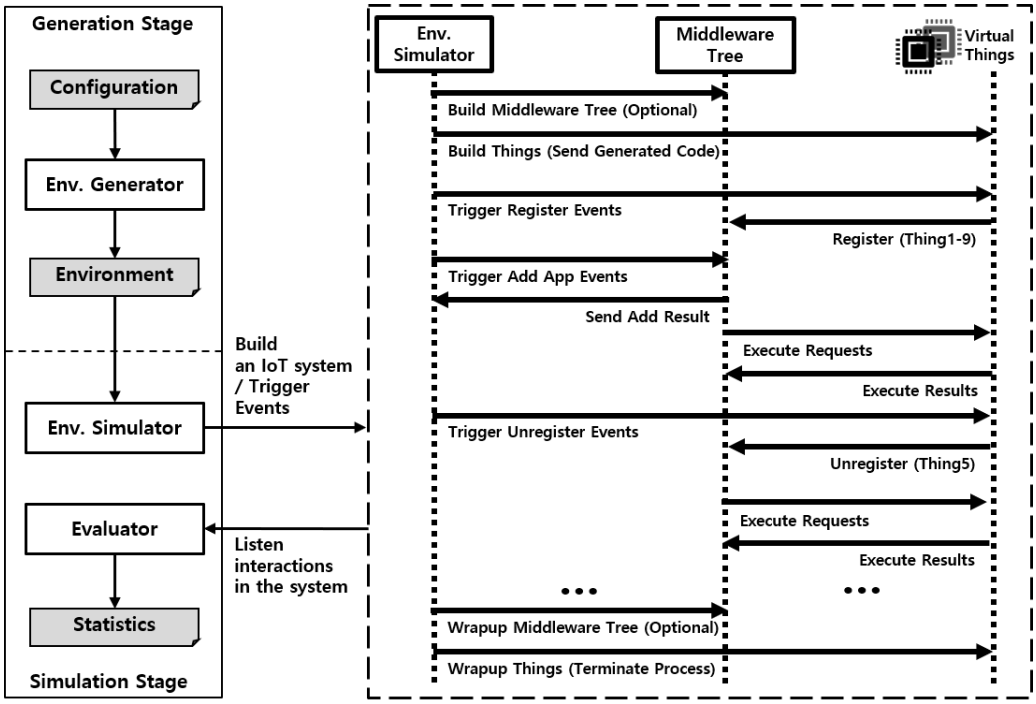


Figure 5.1: Overview of Simulation Process

ecuted in the simulation host. The events such as IoT applications addition and thing registration are also triggered by the simulation host. Note that communication between middlewares or between a middleware and the simulation host occurs through the air with the real communication protocols. Even though there are well-known analyses for modeling the communication workload [68, 69], the communication delay is not calculated by an analytical formula but estimated by direct measurement, which assures the most accurate result.

Figure 5.1 shows the overall simulation process that is divided into two main stages: simulation environment generation and actual simulation. In this way, the simulator enables users to build a variety of IoT environments of their choice and observe actual system performance by calculating statistics for each performance metric.

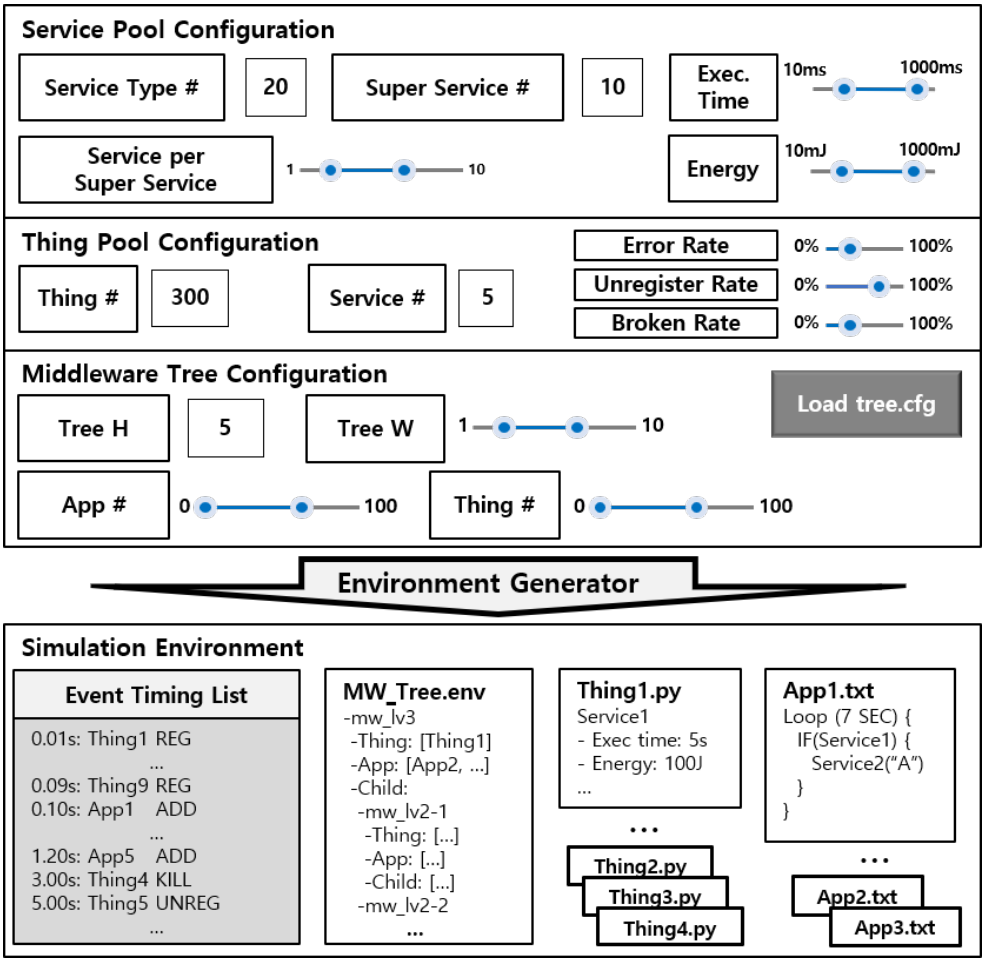


Figure 5.2: Simulation environment generation

5.2 IoT Environment Generation

The strength of this IoT simulator is that it allows the users to create IoT environments with a variety of settings. Users can create any tree topology at will, and fine-tune the configuration of the things, services, applications, etc. that belong to that environment. Most importantly, it gives users the option to randomly generate unexpected faults, so they can see how their IoT middleware responds when something like this happens in their system.

In the generation stage, the *Environment Generator* receives the configuration file

that specifies the system configuration and generates the simulation environment that contains all the information necessary to run the simulation. Examples of the input configuration file and the generated environment files are shown in Fig. 5.2.

The configuration information is composed of three parts. Firstly, we establish the service pool by identifying the quantity of service types and the scope of execution time and energy consumption. We also specify the number of super services, along with the amount of services within each super service. Moreover, the generator can determine a range of characteristics for the services to be produced, such as execution time and energy consumption. This segment determines the services to be employed in the simulation, which are subsequently chosen at random and positioned during the creation of the Things.

Next, we set the number of things and how many services each thing can serve in the thing pool configuration. In this part, we also configure the fault simulation, specifying the probability of unexpected error occurrence, failure of registration, and abnormal termination. Currently, we only support these three faults, but in the future, we will add more faults that may occur in IoT systems to help create a fault-tolerant scheduling algorithm. For instance, communication reversal when using the MQTT QoS-0 protocol can be simulated.

Finally, the tree structure of the middleware can be set up in two ways. The first is to specify the width and height of the tree so that the simulator automatically creates a structured tree, and the second is to allow the user the flexibility to design different tree structures through specific tree configuration files. This is where the user sets the size of the simulation environment since we can set the number of applications and the number of things for each middleware. These settings are specified as ranges and randomly generated within them. Based on the given configuration information, the Environment Generator generates a simulation environment that contains information on the distribution of devices and applications within the middleware hierarchy as well as thing codes

and application scripts.

This gives users the flexibility to create the IoT environments they want to test, with control over complexity, reliability, and scale. Users can create anything from a simple home environment with a single piece of middleware to an environment with multiple edges. They can also control the stability of this IoT environment through error rates to simulate an unstable environment, or they can increase the number of applications added to the system or control the unregistration rate, the probability that a device will disconnect unexpectedly, to simulate an environment with a lot of user interaction.

5.3 IoT Event Simulation

After the simulation environment is generated, the simulation stage begins. We first load the middleware code to the edge devices and thing codes to the virtual things through a secure shell connection to prepare them for execution, following the middleware tree structure specified in the configuration file. This is an advantage when testing in various IoT environments because it significantly reduces the time and effort to access and run programs on each device for simulation. If the middleware tree is already established and codes are not changed, this step can be omitted. Next, events are triggered according to the event timing list, starting with the events to register things and applications. Subsequently, dynamic events in the system, such as ”Unregister Thing”, ”Kill Thing”, and ”Stop/Resume application”, are triggered to examine how the IoT system handles them.

Table 5.1 shows all the types of events supported by the simulator. There are events that start and end the simulation and delay events that wait for a certain amount of time. Some events are simulated by the simulator, such as all kinds of user request events, while others are generated by other devices during the simulation, such as events related to super services. The events that the simulator generates are marked with asterisks. This is done by connecting to each middleware-managed MQTT broker as a single client to publish MQTT messages, adding little additional overhead to the IoT environment.

Table 5.1: Events to be supported by the simulator

Event	Description
SIM_START	Sim. initiates simulation.
SIM_END	Sim. stops simulation.
WAIT	Sim. waits for specified time.
MW_RUN	Sim. executes mw on device.
MW_KILL	Sim. terminates mw on device.
THING_RUN	Sim. executes thing on device.
THING_KILL	Sim. terminates thing on device.
THING_REG	Thing sends registeration request to mw.
THING_UNREG	Thing sends unregisteration request to mw.
THING_REG_RES	MW sends registeration ack to thing.
THING_UNREG_RES	MW sends unregisteration ack to thing.
SERVICE_EXEC	MW sends service execution request packet to thing.
SERVICE_EXEC_RES	Thing sends service execution result.
APP_ADD	Sim. sends app addition request to mw.
APP_RUN	Sim. sends app run request to mw.
APP_STOP	Sim. sends app stop request to mw.
APP_UPDATE	Sim. sends app re-mapping request to mw.
APP_DEL	Sim. sends app deletion request to mw.
APP_ADD_RES	Sim. receives app addition result from mw.
APP_RUN_RES	Sim. receives app run result from mw.
APP_STOP_RES	Sim. receives app stop result from mw.
APP_UPDATE_RES	Sim. receives app re-mapping result from mw.
APP_DEL_RES	Sim. receives app deletion result from mw.
SUP_SERV_EXE	MW sends super service execution to super thing.
SUP_SERV_EXE_RES	Super thing sends super service execution result.
SERVICE_EXE	Super thing sends service execution request to mw.
SERVICE_EXE_RES	MW sends service execution result to super thing.
SUP_SCHED	MW sends super service schedule request to super thing.
SUP_SCHED_RES	Super thing sends super service schedule result to mw.
SCHED	Super thing sends schedule request to mw.
SCHED_RES	MW sends schedule result to super thing.
THING_REG_WAIT	Sim. waits until the things are registered.
APP_ADD_CHECK	Sim. waits until the applications are added.
APP_RUN_CHECK	Sim. waits until the applications are started.

5.4 IoT System Evaluation

During the simulation process, the evaluator of the simulator plays a crucial role in collecting and analyzing packets exchanged between middlewares and virtual things. This is achieved by establishing a connection with an MQTT broker and subscribing to relevant topics to capture the packets. These packets contain vital information, including requests for services sent to the device, as well as responses from the device to the middleware, which may include service execution results or information exchange between different middleware levels.

By meticulously aggregating the arrival times of each packet, the evaluator is able to derive insightful data about the progress of currently running applications and the overall state of the device. This data is then utilized to generate valuable output such as timelines of service executions on each middleware and statistical metrics like average latency, energy consumption, and device utilization for each middleware. Furthermore, the evaluator provides percentage values that offer a measure of how well each application succeeded in scheduling and meeting real-time requirements.

By carefully analyzing these simulation results, users are empowered to identify specific devices that are overutilized and incapable of running applications effectively. Additionally, they can assess whether the current IoT device scale of a particular middleware is capable of meeting the demanding requirements of the running applications. Fig. 5.4 serves as a visual representation of a simplified example of the simulation result, aiding users in comprehending and interpreting these valuable insights.

The simulation results obtained by the evaluator are highly precise and reliable due to their basis on real hardware. By capturing and analyzing the packets exchanged between middlewares and virtual things, the evaluator operates in a realistic environment that closely emulates the actual behavior of hardware devices in an IoT system. This ensures that the derived timeline of service executions, statistical information, and success

metrics accurately reflect the performance and behavior of the system when deployed on real-world hardware.

The utilization of real hardware in the simulation process enhances the credibility of the results, allowing users to make informed decisions and optimizations based on accurate representations of the system's performance characteristics. This empowers users to identify potential bottlenecks, optimize resource allocation, and make necessary adjustments to meet the real-time requirements of applications efficiently.

The evaluator provides the profiling option, which calls the log profiler of the simulator, which is intended to provide even more detailed measurements and extract information by aggregating logs from each device to obtain information that is not available through the MQTT broker.

The log profiler is a powerful tool designed to assist users in identifying bottleneck steps within a system by providing a comprehensive view of every communication and inner-computation step. It enables detailed analysis and understanding of the performance of individual components and their interactions within the system.

Through its logging capabilities, the profiler captures and records the sequence of events and operations that take place during the execution of the system. This includes all communication exchanges between middlewares and virtual things, as well as the inner-computation steps performed within them. By meticulously logging this information, the profiler offers a granular level of insight into the system's behavior. Users can trace the flow of data and operations throughout the system from the initial input to the final output. By examining the logged information, they can identify specific steps or components that exhibit slower execution times or excessive resource consumption, thereby indicating potential bottlenecks within the system. The log profiler serves as a valuable tool in the optimization process, enabling users to make informed decisions based on a comprehensive analysis of the system's performance characteristics.

To ensure accurate measurement of communication overhead with the logged times-

tamp, it is crucial to synchronize the device time before evaluation. The evaluator provides accurate device timing for simulating environments with multiple devices. The two methods used to handle time differences are Network Time Protocol (NTP) [70] and Precision Time Protocol (PTP) [71]. NTP facilitates device time synchronization with a single time server over the air. Prior to each simulation, we undergo a time synchronization process using NTP. It continues sending synchronization requests until the time difference between each device and the NTP server is reduced to 0.1ms or less, at which point the experiment begins.

While this approach represents a modest improvement, it is important to note that the average time synchronization error of approximately 10ms may result in a reversal of the log order for certain communication processes, making it insufficient for detailed profiling purposes. As an alternative, the evaluator has the option to utilize the PTP method, which achieves clock accuracy in the sub-microsecond range and is well-suited for measuring systems within a local area network. During our experiments with the simulator, we discovered that although wireless synchronization using PTP still exhibits inaccuracies, wired connections yield reliable results without any log reversal in communication processes. Through these time synchronization techniques, the simulator effectively mitigates device time mismatches and measures communication overhead in detail. This fine-grained profiling capability allows users to more precisely optimize scheduling algorithms or make target system performance predictions.

5.5 Scheduling Framework

After developing the simulator, we discovered that simulating the system involves handling numerous hyperparameters. This complexity arises from the variety of scheduling policies that can be applied in the simulation and the need to evaluate each policy across different IoT environments. However, manually experimenting with all the policies and environments is a tedious and error-prone task. Therefore, we require a framework

that automates this repetitive process and manages everything from setup to statistical analysis in a cohesive manner.

Hence, we propose a scheduling framework that automates the evaluation of different scheduling policies using the IoT simulator as its core engine. The workflow of the framework, as depicted in Fig. 5.3, outlines the steps involved in exploring and assessing the performance of various scheduling policies. By watching the IoT simulation environment, each scheduling policy can be thoroughly evaluated. It is worth noting that the middleware is designed to facilitate easy replacement of scheduling policies, allowing for seamless automation of the process.

The framework systematically explores multiple candidate policies, compares their simulation results, and ranks them based on specific scheduling objectives such as energy consumption or average response time. Users have the flexibility to incorporate multiple objectives or vary the scheduling objectives within the framework. This empowers users to observe the impact of different scheduling policies on their specific IoT system. Moreover, the framework offers default algorithms that users can identify the most effective scheduling policies for their system's scale. Alternatively, users can develop and evaluate their own custom scheduling algorithms using the framework's capabilities.

By adopting the proposed scheduling framework, users can streamline the process of evaluating and selecting suitable scheduling policies for their IoT systems. The automation provided by the framework eliminates the need for manual configuration and analysis, enabling efficient decision-making and optimization of system performance.

5.6 Experiments

In this hierarchical IoT platform, one important concern is the potential increase in latency of super services as the number of communications between layers within the platform grows. To address this concern and evaluate the scalability of the proposed platform, a series of experiments are conducted. These experiments aim to measure the

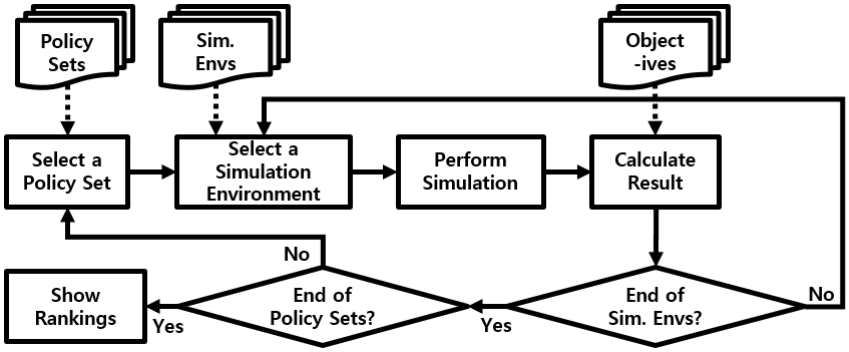


Figure 5.3: Workflow of the proposed scheduling framework

latency of super services while varying the system configuration.

Additionally, further experiments are performed to observe how different scheduling policies impact the overall performance of the system. By analyzing the effects of different scheduling policies, valuable insights can be gained regarding their influence on latency and system responsiveness. All of these experiments are carried out using the simulator described earlier, providing a controlled environment to measure and assess the performance of the hierarchical IoT platform. The simulator enables researchers to replicate various system configurations, simulate communication between layers, and gather quantitative data on latency and other performance metrics.

Through these experiments, a comprehensive understanding of the scalability and performance characteristics of the proposed hierarchical IoT platform can be obtained. The results of these experiments will provide valuable insights for optimizing the system's configuration, selecting appropriate scheduling policies, and ensuring efficient and responsive operation in real-world IoT deployments.

5.6.1 Experimental Setup

The hardware platforms used in the experiments are as follows.

- Server (Simulator and Virtual Things): Intel Core i9-12900K, 64GB RAM, and Ubuntu 22.04.1 LTS x86_64

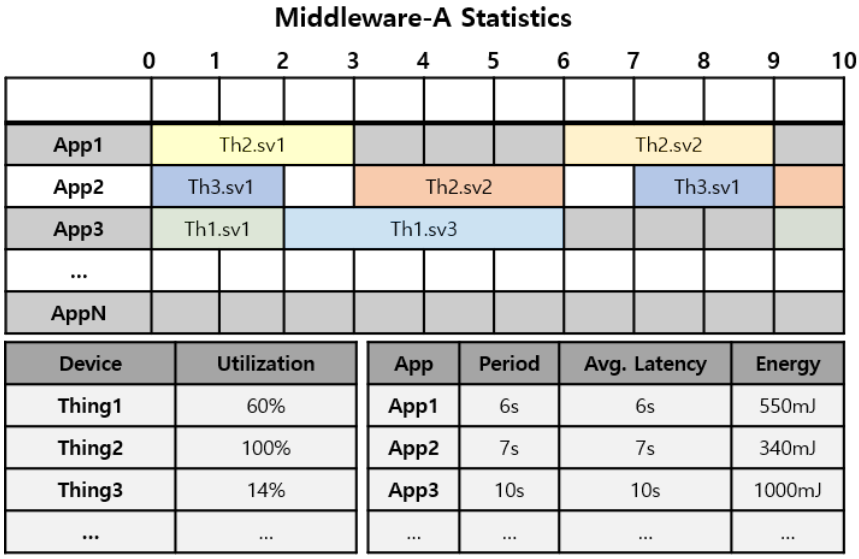


Figure 5.4: A simulation result example

- Raspberry Pi 4 (Middlewares): ARM Cortex A72, 8GB RAM, and Raspbian GNU/Linux 11 (bullseye) armv7l

Ten Raspberry Pi boards are used to run a middleware and an MQTT broker, one for each, constructing a tree architecture. They connect to each other based on the tree structure configuration file. A simulation is repeated 10 times to obtain the result by averaging the simulation outcomes. In the repetition, to mitigate the unexpected performance gap between the devices, we randomly pick and assign to the levels of the tree. For IoT devices, we generate virtual things that model hardware devices and execute lightweight processes to simulate them. Thus, the simulator and virtual things are run together on the same server.

To vary the system configuration, 20 normal services, and three super services are first generated. We let each thing pick three services randomly, allowing duplication. The execution time of each service is randomly set between 0.1 to 1.0 seconds. While all super services are defined at the top-level middleware, they can be requested either at the bottom level or the top level, depending on two use cases shown in Fig. 4.6.

The system configuration is defined by the following tuple: $Config = \{h, w, N_a, N_t, S_t, S_b, NS_s\}$ where h and w indicate the height and the width of the tree structure, N_a and N_t mean the number of IoT applications and the number of devices running at the bottom level, S_t and S_b mean the number of super services called at the top level and the bottom level, and NS_s indicates the number of services contained in a super service. Even though we may define more parameters to define a general system configuration, we assume that the same configuration parameters are applied to all applications or all middlewares, in order to avoid unnecessary complexity by restricting the configuration space.

5.6.2 Super Service Latency

A super service has NS_s services internally that will be mapped to virtual devices at the bottom level. Since a virtual device is run on the server and a middleware at the bottom level runs on a Raspberry Pi board, the communication delay between the two hardware platforms is directly measured. As explained in Section 4.4.1, we need to schedule a super service before execution. In our experiments, we assume that the scheduling of super services has already been performed, and we focus solely on measuring the execution delay. To obtain accurate results, each experiment is repeated 10 times, and the average execution delay is calculated. The net overhead incurred by the execution of super services is then computed by subtracting the service execution time from the total end-to-end execution time.

By conducting these measurements and analyzing the net overhead, we can gain insights into the performance of super services and their impact on the overall system. This evaluation helps us understand the efficiency of the proposed platform and provides valuable information for further optimization and improvement.

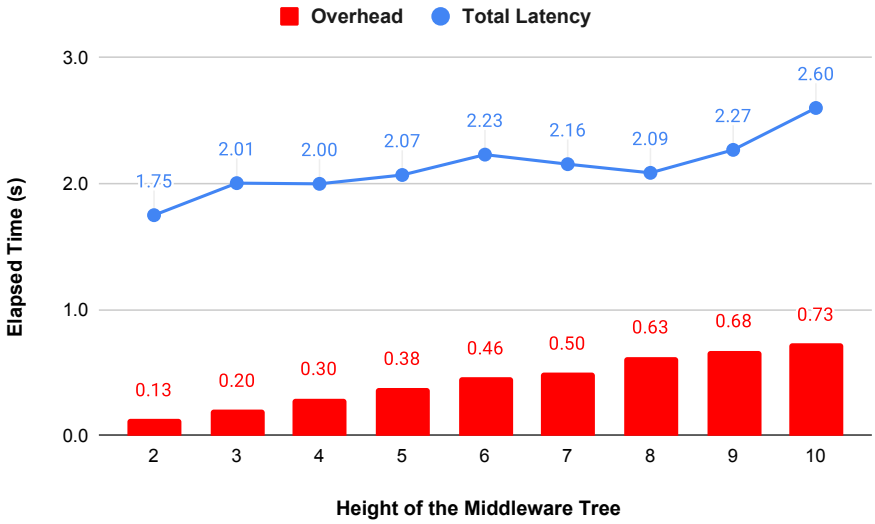


Figure 5.5: The average latency of a super service and the overhead as we vary the height of the middleware tree

5.6.2.1 Varying the height of the tree structure

At first, we measure the super service latency varying the height of the tree structure whose configuration can be represented by $Config = \{h, 1, 10, 10, 2, 2, 4\}$. To focus on scalability with respect to the height, we set the width to 1. We assume that 10 IoT applications are run at the bottom middleware. In addition, two super services that include four services each are requested at the bottom level and the top level, respectively.

Fig. 5.5 shows the super service latency (blue line) and the overhead involved in the super service execution (red bars) as the height of the tree varies from 2 to 10. Since the service execution time is randomly set, we focus on the overhead only. The overhead includes the scheduling overhead of the middleware and communication overhead between devices. From Fig. 5.5, it is observed that the overhead of super service execution is proportional to the height as expected: Even when the height is ten and execution of a super service incurs tens of inter-device communications, the total overhead is relatively low (0.73s) compared with the super service request interval that is usually much longer.

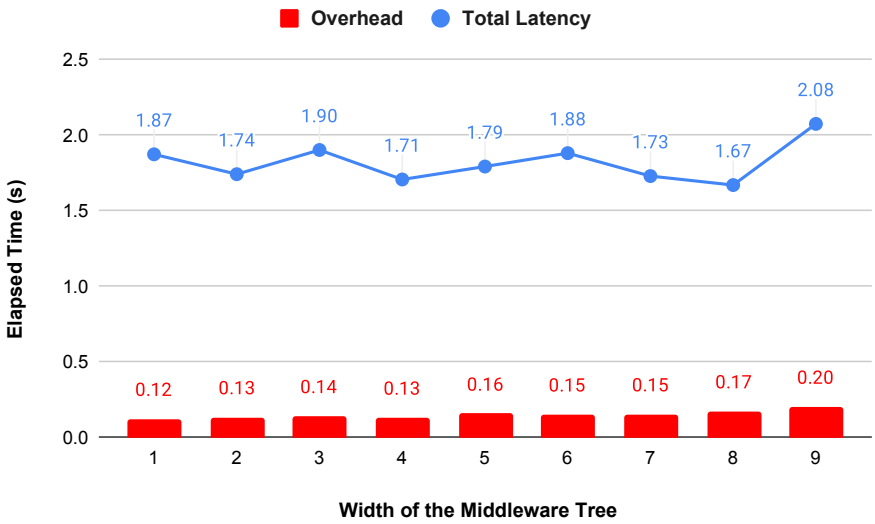


Figure 5.6: The average latency of a super service and the overhead as we vary the width of the middleware tree

It means that a user in a smart city-scale IoT system could get the result of a city-level public service with an overhead lower than 1 sec.

5.6.2.2 Varying the width of the tree structure

In this experiment, we vary the width of the tree structure while the height is fixed to 2: $Config = \{2, w, 10, 10, 2, 2, 4\}$. We measure the super service latency and compute the overhead, varying the width from 2 to 9. In contrast to the previous experiment, the workload of the top-level middleware increases as the number of super services and the number of services that need to be scheduled increase proportionally to the width. Thus we can check how the top-level middleware can support the increasing workload as the system complexity increases. Even with load balancing across middleware in the hierarchy, the more child nodes connected to a middleware, the more workload that middleware is faced with at any given time.

Fig. 5.6 shows that the latency and the overhead of a super service are slowly increasing as the width increases: the overhead is only 0.20s even when the top-level mid-

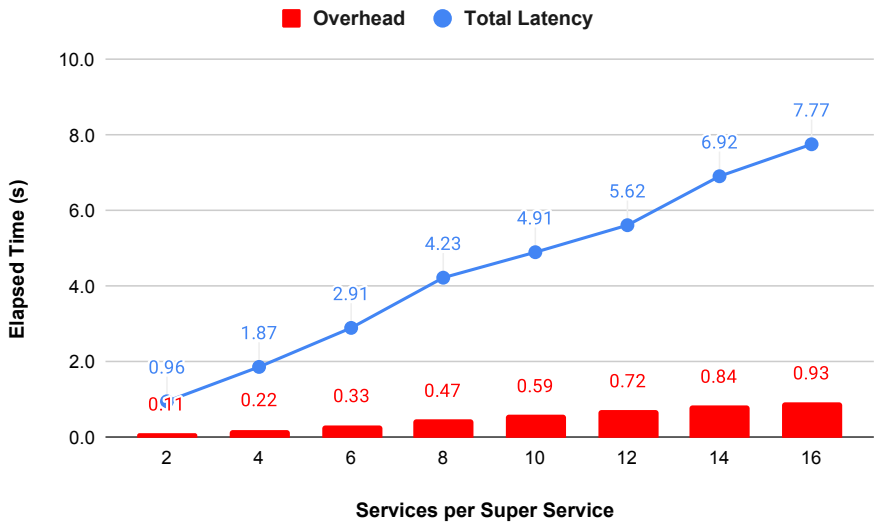


Figure 5.7: The average latency of a super service and the overhead as we vary the number of services per super service

dleware has nine child middlewares and the total number of super services is 20 (9 times 2 plus 2). It means that the computing overhead of the edge middleware involved in the execution of a super service is not a performance bottleneck of an upper middleware.

5.6.2.3 Varying the number of services per super service

Next, we perform a similar experiment, varying the number of services per super service while fixing the tree structure's height and width as $Config = \{3, 2, 10, 10, 2, 2, NS_s\}$. Note that an upper middleware has two child middlewares so that the number of bottom level middlewares is 4. The measurement results on the average latency and the overhead of super service execution are shown in Figure 5.7 while varying the number of services from 2 to 16. As the number of services increases, the latency and the overhead increase proportionally. The impact of communication overhead can be seen indirectly through this experiment, as every 2 increase in services within a super service increases the number of in-layer communications required to execute once by 8. Note that the involved overhead is still under 1 sec when the number of services per super service is as large as

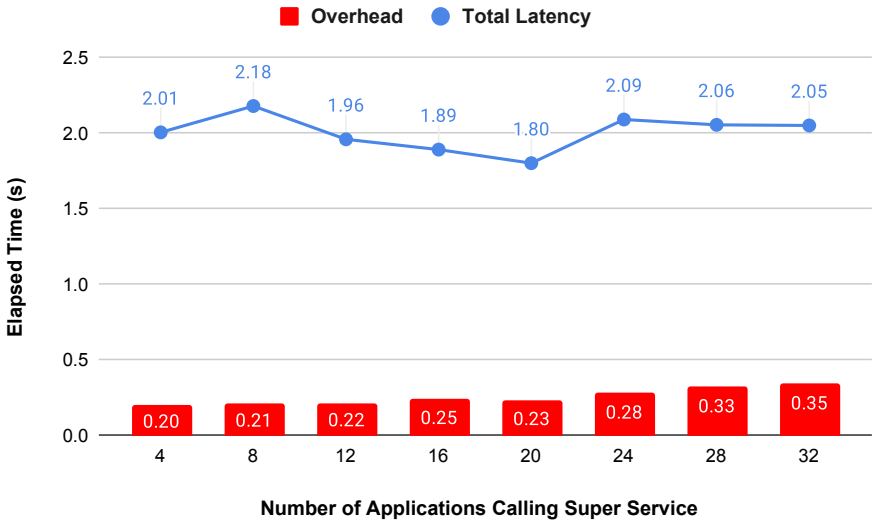


Figure 5.8: The average latency of a super service and the overhead as we vary the number of super service requests per middleware

16, where the communication occurs 130 times at maximum.

No matter how complex a super service is created, it's highly unlikely that users will create a super service within it that requires 16 steps of service execution, and even if it is more complex than that, according to this graph, it would only incur a few seconds of overhead. It implies that we can make a complex super service without worrying about the latency overhead.

5.6.2.4 Varying the number of super service requests per middleware

In this experiment, we increase the number of super services with the following configuration: $Config = \{3, 2, 10, 10, S_t, S_b, 4\}$. Compared with the earlier experiments in which only two applications in each middleware call a super service, we increase the number of applications that call a super service from 4 to 32 in each middleware at the top level middleware and the bottom level. As displayed in Fig. 5.8, the execution overhead is slowly increasing. While the overhead grows monotonically, the total latency fluctuates

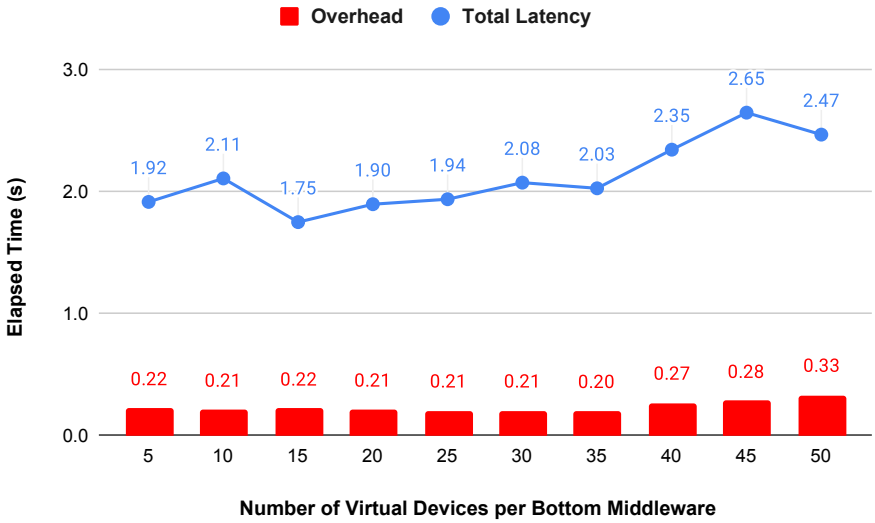


Figure 5.9: The average latency of a super service and the overhead as we vary the number of virtual devices per bottom middleware

slightly because the execution time assigned to the service is randomly generated within a set range when the environment is generated. The experiment results provided valuable insights, indicating that the overhead itself is not the primary performance bottleneck even when the number of super services in a tree reaches as high as 160. These findings highlight the system's ability to handle a substantial increase in the number of super services without a drastic degradation in performance.

5.6.2.5 Varying the number of virtual devices per bottom middleware

In this particular experiment, the scalability of the system is investigated in terms of the number of devices. To achieve this, the number of virtual devices per bottom-level middleware is increased from 5 to 50, denoted by the configuration $Config = 3, 2, 10, N_t, 2, 2, 4$, where N_t varies between 5 and 50. Fig. 5.9 illustrates the results of this experiment. It can be observed that there is no significant difference in the overhead until the total number of devices reaches 140 ($N_t = 35$). However, once the number of reg-

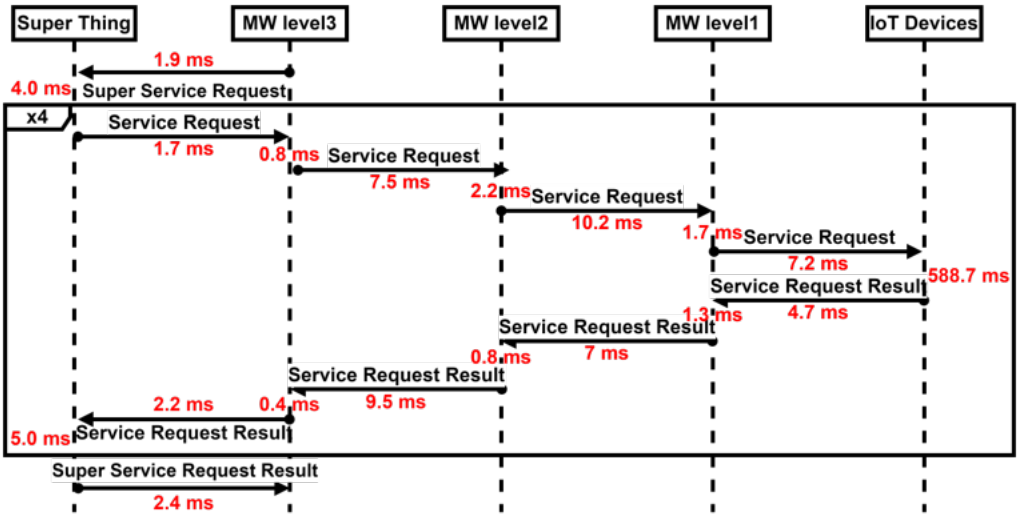


Figure 5.10: A sequence diagram of super service execution with the overhead information of each step

istered devices exceeds 40, the overhead starts to increase. This increase in overhead can be attributed to the internal workload within the middleware surpassing a certain threshold. Remarkably, even in an environment with 200 IoT end devices, the overhead remains relatively small, measuring only one-third of a second. This indicates that the system performs well and exhibits promising scalability potential for larger-scale deployments. The findings of this experiment suggest that the proposed system can handle larger systems efficiently, maintaining a manageable level of overhead even with a significant number of IoT end devices. This scalability is crucial for accommodating the ever-growing number of devices in IoT environments and ensuring the smooth operation of the system as it expands.

5.6.2.6 Breaking down the super service overhead

In this experiment, we break down the execution overhead of a super service with the following configuration: $Config = \{3, 2, 10, 10, 2, 2, 4\}$. Figure 5.10 shows the sequence diagram that details the overhead at each step in the execution of a super service.

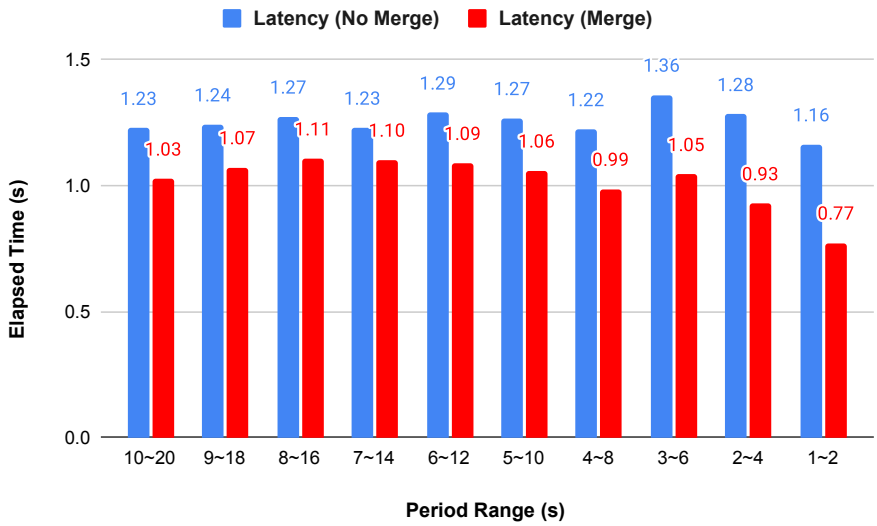


Figure 5.11: The average latency of services as we vary the range of application periods

The overhead values are averaged over 10 executions of the super services performed during each simulation. Since the super services are composed of four services, there are 16 communications between middlewares, 8 between a middleware and a device, and 10 between the top-level middleware and a super thing for a single execution. Communication between middlewares is performed by wireless communication between Rasberry Pi edge devices, while communication between a middleware and a device is between a Rasberry Pi and a simulation host. Communication between the middleware and a super thing is performed in the same edge device. The experimental result shows that the scheduling overhead within the edge middleware device is relatively low, on the order of 0.4-2.2 ms, compared with the communication overhead between edge devices.

5.6.3 Scheduling Policy

As explained in Section 4.4, we may change the scheduling policies that constitute the scheduling algorithm in the proposed IoT platform. To examine the effect of a scheduling policy on performance, we implemented two schedulers with two different

policies for service execution and compared them, varying the period of IoT applications. One is the baseline policy(”No Merge”) that serves the requests one by one in the FIFO order, and the other is to merge all requests for the same service in the run queue for performance optimization (”Merge”).

With the same configuration, $Config = \{3, 2, 10, 10, 2, 2, 4\}$, we vary the range of application periods that are randomly chosen within the range. Fig. 5.11 measures the average latency of two policies. Since we reject the service request if it violates the latency constraint, more service requests are rejected as the period reduces. It explains why the average latency is relatively stable over a wide range of periods. On the other hand, as the period range of the applications decreases, the gap between the two policies gets larger. As the period is reduced, the number of simultaneous requests on the same device increases, triggering more merging. It implies that there is much room for performance improvement in the proposed IoT platform. We leave it as future work to find better scheduling policies for performance improvement.

Chapter 6

Conclusion and future work

In this thesis, we present a novel and scalable hierarchical edge-based architecture for service-oriented IoT platforms. We derive design requirements and propose technical solutions to support the new middleware structure. The architecture includes a special type of service called a super service, which is defined in the higher-level middleware and enables communication and resource sharing between edges. The lower-level middleware exploits the capabilities of the extended IoT system through IoT applications that request and utilize these super services.

The main role of middleware in a service-oriented IoT platform is to schedule service requests to smart devices. We present how the scheduling can be performed in the hierarchical structure, considering the possibility of dynamic device status change and execution of super services. To enable users to customize and evaluate their own scheduling policies, a scheduling simulation framework is developed. This framework comprises an IoT environment generator and an IoT system simulator, providing a comprehensive toolset for studying and analyzing different scheduling algorithms.

Extensive experiments conducted using the simulation framework demonstrate the scalability of the proposed IoT platform. The experiments focus on measuring the latency of super services as the complexity of the system increases. The results confirm the platform's ability to handle larger-scale deployments while maintaining acceptable levels

of latency. In addition, a toy-level smart campus test bed is implemented to examine the viability of the proposed platform in real life.

Since device-to-device communication is not allowed in the proposed IoT platform for safety, all of the communications among IoT devices are through the edge middleware device. Hence the edge device may become the performance bottleneck if an IoT application requires heavy traffic between devices, such as live video streaming. To solve this problem, we leave it as a future work to allow device-to-device communication between two certified devices.

There are several other research topics for the proposed IoT architecture. One is to explore various scheduling policies to find a suitable scheduling algorithm for their system. We may want to consider the dynamic variation of service time, conditional execution of services, and so on in the scheduling policy. The following is a study to improve reliability. Robust connection between edges in a tree structure compared to peer-to-peer connections is a topic of future research. A backup middleware could be prepared for the case an edge is destroyed, or perhaps there may be a way to build a bypass connection in case the connection between edges is lost. And how to utilize the IoT big data collected at each level is another topic to study. Last but not least, we will deploy the proposed IoT platform to a real building in the future. By addressing these research topics and continuously refining the architecture, the proposed service-oriented IoT platform holds great promise for enabling scalable and efficient IoT deployments in various domains.

Bibliography

[1] Statista. Number of internet of things (iot) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030 (in billions), 2022.

[2] Google. Google home, https://developers.home.google.com/, 2023.

[3] Amazon. Alexa, https://alexa.amazon.com/, 2023.

[4] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context aware computing for the internet of things: A survey. IEEE communications surveys & tutorials, 16(1):414–454, 2013.

[5] David Gil, Antonio Ferrández, Higinio Mora-Mora, and Jesús Peral. Internet of things: A review of surveys based on context aware intelligent services. Sensors, 16(7):1069, 2016.

[6] Connectivity Standards Aliance. Matter, https://csa-iot.org/all-solutions/matter/, 2023.

[7] IoTAnalystics. Iot platform companies landscape 2021/2022: Market consolidation has started, 2021.

[8] Flavio Cirillo, Gürkan Solmaz, Everton Luís Berz, Martin Bauer, Bin Cheng, and Ernoe Kovacs. A standard-based open source iot platform: Fiware. IEEE Internet of Things Magazine, 2(3):12–18, 2019.

[9] Ricardo S Alonso, Inés Sittón-Candanedo, Óscar García, Javier Prieto, and Sara Rodríguez-González. An intelligent edge-iot platform for monitoring livestock and crops in a dairy farming scenario. Ad Hoc Networks, 98:102047, 2020.

[10] Partha Pratim Ray. A survey of iot cloud platforms. Future Computing and Informatics Journal, 1(1-2):35–46, 2016.

[11] openHAB. Remote openhab binding, https://www.openhab.org/addons/bindings/remoteopenhab, 2023.

[12] Custom components for Home Assistant. Remote home assistant, https://github.com/custom-components/remote_homeassistant, 2023.

[13] Fatih Bakir, Rich Wolski, Chandra Krintz, and Gowri Sankar Ramachandran. {Devices-as-Services}: Rethinking scalable service architectures for the internet of things. In 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19), 2019.

[14] Hyunjae Lee, Eunjin Jeong, Donghyun Kang, Jinmyeong Kim, and Soonhoi Ha. A novel service-oriented platform for the internet of things. In Proceedings of the Seventh International Conference on the Internet of Things, pages 1–8, 2017.

[15] Leonardo Babun, Kyle Denney, Z Berkay Celik, Patrick McDaniel, and A Selcuk Uluagac. A survey on iot platforms: Communication, security, and privacy perspectives. Computer Networks, 192:108040, 2021.

[16] Amazon. Aws iot core, https://aws.amazon.com/iot-core, 2023.

[17] Samsung. Smartthings developer documentation, https://developer.smartthings.com, 2023.

[18] Apple. Homekit developer documentation, https://developer.apple.com/homekit/, 2023.

[19] Mahadev Satyanarayanan. The emergence of edge computing. Computer, 50(1):30–39, 2017.

[20] Huihui Xue, Bi Huang, Mingming Qin, Hua Zhou, and Hongji Yang. Edge computing for internet of things: A survey. In 2020 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics), pages 755–760. IEEE, 2020.

[21] Morghan Hartmann, Umair Sajid Hashmi, and Ali Imran. Edge computing in smart health care systems: Review, challenges, and research directions. Transactions on Emerging Telecommunications Technologies, 33(3):e3710, 2022.

[22] Florian Heimgaertner, Stefan Hettich, Oliver Kohlbacher, and Michael Menth. Scaling home automation to public buildings: A distributed multiuser setup for openhab 2. In 2017 Global Internet of Things Summit (GIoTS), pages 1–6. IEEE, 2017.

[23] Home Assistant. Home assistant github, https://github.com/home-assistant, 2023.

[24] Pasquale Pace, Gianluca Aloi, Raffaele Gravina, Giuseppe Caliciuri, Giancarlo Fortino, and Antonio Liotta. An edge-based architecture to support efficient applications for healthcare industry 4.0. IEEE Transactions on Industrial Informatics, 15(1):481–489, 2018.

[25] Abel O Gomez Rivera, Deepak K Tosh, and Laurent Njilla. Scalable blockchain implementation for edge-based internet of things platform. In MILCOM 2019-2019 IEEE Military Communications Conference (MILCOM), pages 1–6. IEEE, 2019.

[26] Nebil Ben Mabrouk, Sandrine Beauche, Elena Kuznetsova, Nikolaos Georgantas, and Valérie Issarny. Qos-aware service composition in dynamic service oriented environments. In Middleware 2009: ACM/IFIP/USENIX, 10th International Middleware Conference, Urbana, IL, USA, November 30–December 4, 2009. Proceedings 10, pages 123–142. Springer, 2009.

[27] Husnu S Narman, Md Shohrab Hossain, Mohammed Atiquzzaman, and Haiying Shen. Scheduling internet of things applications in cloud computing. Annals of Telecommunications, 72:79–93, 2017.

[28] Ling Li, Shancang Li, and Shanshan Zhao. Qos-aware scheduling of services-oriented internet of things. IEEE Transactions on Industrial Informatics, 10(2):1497–1505, 2014.

[29] Sayantani Basu, Marimuthu Karuppiah, K Selvakumar, Kuan-Ching Li, SK Hafizul Islam, Mohammad Mehedi Hassan, and Md Zakirul Alam Bhuiyan. An intelligent/cognitive model of task scheduling for iot applications in cloud computing environment. Future Generation Computer Systems, 88:254–261, 2018.

[30] Kostas Kolomvatsos and Christos Anagnostopoulos. Multi-criteria optimal task allocation at the edge. Future Generation Computer Systems, 93:358–372, 2019.

[31] Shangguang Wang, Yan Guo, Ning Zhang, Peng Yang, Ao Zhou, and Xuemin Shen. Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach. IEEE Transactions on Mobile Computing, 20(3):939–951, 2019.

[32] Mohammad Goudarzi, Marimuthu Palaniswami, and Rajkumar Buyya. A distributed application placement and migration management techniques for edge and fog computing environments. In 2021 16th Conference on Computer Science and Intelligence Systems (FedCSIS), pages 37–56. IEEE, 2021.

[33] Mohammad Goudarzi, Marimuthu Palaniswami, and Rajkumar Buyya. Scheduling iot applications in edge and fog computing environments: a taxonomy and future directions. ACM Computing Surveys, 55(7):1–41, 2022.

[34] Munish Bhatia, Sandeep K Sood, and Simranpreet Kaur. Quantum-based predictive fog scheduler for iot applications. Computers in Industry, 111:51–67, 2019.

[35] Thomas Pusztai, Fabiana Rossi, and Schahram Dustdar. Pogonip: Scheduling asynchronous applications on the edge. In 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), pages 660–670, 2021.

[36] JongBeom Lim. Latency-aware task scheduling for iot applications based on artificial intelligence with partitioning in small-scale fog computing environments. Sensors, 22(19):7326, 2022.

[37] Massimiliano De Benedetti, Fabrizio Messina, Giuseppe Pappalardo, and Corrado Santoro. Jarvsis: a distributed scheduler for iot applications. Cluster Computing, 20(2):1775–1790, 2017.

[38] Djabir Abdeldjalil Chekired, Lyes Khoukhi, and Hussein T Mouftah. Industrial iot data scheduling based on hierarchical fog computing: A key for enabling smart factory. IEEE Transactions on Industrial Informatics, 14(10):4590–4602, 2018.

[39] Ramesh Yerraballi and Ravi Mukkamalla. Scalability in real-time systems with end-to-end requirements. Journal of Systems Architecture, 42(6-7):409–429, 1996.

[40] Qi Zhu, Yang Yang, Marco Natale, Eelco Scholte, and Alberto Sangiovanni-Vincentelli. Optimizing the software architecture for extensibility in hard real-time distributed systems. IEEE Transactions on Industrial Informatics, 6(4):621–636, 2010.

[41] Iain Bate and Paul Emberson. Incorporating scenarios and heuristics to improve flexibility in real-time embedded systems. In 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06), pages 221–230. IEEE, 2006.

[42] Jangryul Kim, Kangkyu Park, Hoeseok Yang, and Soonhoi Ha. Adaptive runtime scheduling of dependent services for service-oriented iot systems. Design Automation for Embedded Systems, 25(4):265–295, 2021.

[43] Parvaneh Asghari, Amir Masoud Rahmani, and Hamid Haj Seyyed Javadi. Service composition approaches in iot: A systematic review. Journal of Network and Computer Applications, 120:61–77, 2018.

[44] Damian Arellanes and Kung-Kiu Lau. Evaluating iot service composition mechanisms for the scalability of iot systems. Future Generation Computer Systems, 108:827–848, 2020.

[45] Giuseppe Tricomi, Carlo Scaffidi, Giovanni Merlino, Francesco Longo, Salvatore Distefano, and Antonio Puliafito. From vertical to horizontal buildings through iot and software defined approaches. In 2021 IEEE International Conference on Smart Computing (SMARTCOMP), pages 365–370. IEEE, 2021.

[46] Mohammad Abdur Razzaque, Marija Milojevic-Jevric, Andrei Palade, and Siobhán Clarke. Middleware for internet of things: a survey. IEEE Internet of things journal, 3(1):70–95, 2015.

[47] Thiago Teixeira, Sara Hachem, Valérie Issarny, and Nikolaos Georgantas. Service oriented middleware for the internet of things: A perspective. In Towards a Service-Based Internet: 4th European Conference, ServiceWave 2011, Poznan, Poland, October 26-28, 2011. Proceedings 4, pages 220–229. Springer, 2011.

[48] An Ngoc Lam. Sai: a service oriented autonomic iot platform. Master's thesis, 2017.

[49] Gareth George, Fatih Bakir, Rich Wolski, and Chandra Krintz. Nanolambda: Implementing functions as a service at all resource scales for the internet of things. In 2020 IEEE/ACM Symposium on Edge Computing (SEC), pages 220–231. IEEE, 2020.

[50] Praveen Pawar and Aditya Trivedi. Device-to-device communication based iot system: benefits and challenges. IETE Technical Review, 36(4):362–374, 2019.

[51] Allseen Alliance. Alljoyn® framework, https://allseenalliance.org/framework/, 2023.

[52] Open Connectivity Foundation. Iotivity, https://www.iotivity.org/, 2023.

[53] ptc. Build a better iiot solution with thingworx, https://www.ptc.com/en/products/thingworx, 2023.

[54] Sumi Helal. Programming pervasive spaces. IEEE Pervasive Computing, 4(1):84–87, 2005.

[55] Soma Bandyopadhyay, Munmun Sengupta, Souvik Maiti, and Subhajit Dutta. Role of middleware for internet of things: A study. International Journal of Computer Science and Engineering Survey, 2(3):94–105, 2011.

[56] MQTT.org. Mqtt, https://mqtt.org/, 2023.

[57] Eclipse Foundation. Mosquitto, https://mosquitto.org/, 2023.

[58] A.C. Weaver. Secure sockets layer. Computer, 39(4):88–90, 2006.

[59] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. Mqtt-s — a publish/subscribe protocol for wireless sensor networks. In 2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08), pages 791–798, 2008.

[60] Eclipse Foundation. Eclipse paho mqtt-sn c/c++ client for embedded platforms, https://github.com/eclipse/paho.mqtt-sn.embedded-c, 2023.

[61] Connectivity Standards Alliance. Zigbee, https://csa-iot.org/all-solutions/zigbee/, 2023.

[62] Paolo Baronti, Prashant Pillai, Vince WC Chook, Stefano Chessa, Alberto Gotta, and Y Fun Hu. Wireless sensor networks: A survey on the state of the art and the 802.15. 4 and zigbee standards. Computer communications, 30(7):1655–1695, 2007.

[63] Arduino Team. Ardunio, 2023.

[64] IBM Cloud. Watson iot platform, https://cloud.ibm.com/docs/iot?topic=iot-getting-started, 2023.

[65] IFTTT Inc. Ifttt, https://ifttt.com/, 2023.

[66] The MathWorks Inc. Thingspeak, https://thingspeak.com/, 2023.

[67] ThingsBoard Inc. Thingsboard https://thingsboard.io/, 2023.

[68] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In 2000 IEEE International Symposium on Circuits and Systems (ISCAS), volume 4, pages 101–104. IEEE, 2000.

[69] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis–the symta/s approach. IEE Proceedings-Computers and Digital Techniques, 152(2):148–166, 2005.

[70] Network Time Foundation. Network time protocol (ntp) project, 2023.

[71] Kang Lee, John C Eidson, Hans Weibel, and Dirk Mohl. Ieee 1588-standard for a precision clock synchronization protocol for networked measurement and control systems. In Conference on IEEE, volume 1588, page 2, 2005.

요 약

사물인터넷(IoT)은 일상 생활에서 더 보편화되고 있으며, 많은 상용화된 IoT 플랫폼이 사용되고 있다. IoT 솔루션을 구축하는데 가장 핵심이 되는 IoT 플랫폼은, 설계를 어떻게 하느냐가 IoT 시스템의 특성을 좌우하기 때문에 다양한 디자인의 플랫폼이 연구되고 있다. 그 중, 엣지 기반 플랫폼은 디바이스와 가까운 곳에 미들웨어가 위치하여 반응속도와 프라이버시 보호의 이점이 있고, 서비스 지향 IoT는 사용자가 디바이스를 직접 지정하지 않고 IoT와 소통할 수 있게 해준다는 강점이 있다. 엣지 기반 플랫폼의 약점인 확장성을 극복할 수 있도록 하는 다양한 연구가 진행되고 있지만 이들은 모두 수평적 구조 즉, 단일 계층을 통한 엣지간 연결만을 고려하고 있다.

본 논문에서는 엣지 기반 IoT와 서비스 지향 IoT의 장점들을 유지하면서도 트리 구조를 따라 유연하게 확장될 수 있도록 하는 엣지 미들웨어의 계층적 구조를 제안한다. 이 때, 이 새로운 구조를 실현하기 위한 설계 요구사항들과 테크니컬 챌린지들을 도출하고, 그에 대한 솔루션을 제시한다. 계층 구성, 자원 공유, 스케줄링 등의 다섯 가지 메인 요구사항들은 크게 두 가지의 새로운 기법을 통해 해결된다. 첫째, 프라이버시를 보호하며 엣지 간 통신 및 자원 공유를 가능하게 하기 위해 슈퍼 서비스라는 새로운 유형의 서비스를 도입한다. 이는 계층에서 서비스 요청 중재자의 역할을 하며 각 미들웨어간의 서비스 요청을 대신 한다. 둘째, 서비스 지향 IoT의 유연한 디바이스와 서비스간의 매핑을 가능하게 하기 위해 슈퍼 서비스가 추가됨으로써 더욱 복잡해진 스케줄링 문제를 푸는 분산 알고리즘을 고안한다. 이는 슈퍼 서비스를 실시간으로 대응할 수 있게 해줌과 동시에 스케줄링 부하를 여러 엣지에 분산시킴으로써 한 엣지로의 연산 과부하도 막아준다.

IoT 시스템에서는 수많은 서로 다른 디바이스가 존재하며 예측 불가한 동적인 이벤트들을 자주 일어나기 때문에 정확한 모델링 기반 시뮬레이션은 거의 불가능하다. 따라서, 설정에 따라 실제 디바이스 위에 계층 구조 IoT 환경을 생성하고, 설정에 따라 임의 생성된 이벤트를 발생시켜 IoT 미들웨어와 디바이스간의 실제 교류를 관측하는 하드웨어 인 더 루프 시뮬레이터를 구현한다. 이 시뮬레이터를 기반으로 하여, 다양한 스케줄링 알고리즘의 성능을 여러 IoT 환경에서 평가할 수 있게 해주는 프레임워크를

제공한다. 사용자는 이를 통해 자신이 목표로 하는 시스템의 규모와 특성에 따라 IoT 환경을 미리 만들어볼 수 있을 뿐더러, 최적의 스케줄링 알고리즘을 찾을 수 있다. 이 프레임워크는 향후 추가연구를 위해 깃헙에 공개하였다.

마지막으로, 스마트 캠퍼스 테스트 베드 구현과 시뮬레이션 프레임워크를 이용한 다양한 실험을 통해 제안한 플랫폼의 실행 가능성을 검증한다. 서울대학교 내에 두 건물과 다섯 개의 방에 라즈베리 파이를 설치하여 계층 구조를 구축하고, 서로 다른 엣지를 연결하는 다양한 IoT 응용들을 실행하여 실제 공간에서도 문제없이 작동함을 확인한다. 슈퍼 서비스의 성능을 확인하기 위해 트리의 높이, 너비, 응용 갯수 등을 조정하며 부하가 높아짐에 따른 반응속도를 측정하여 큰 규모의 계층 구조에서도 슈퍼 서비스의 성능이 충분하다는 것을 입증한다. 또, 서로 다른 스케줄링 정책을 동일한 환경에 적용했을 때 결과가 달라지는 것을 확인하며 스케줄링 알고리즘 개발을 돕는 스케줄링 프레임워크의 필요성을 다시 한번 보인다.

주요어 : 사물인터넷 플랫폼, 엣지 기반, 서비스 기반, 계층적 구조
학번 : 2015-21232