FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

DOT - Digital Orchestration of Things

Carlos Alexandre Teixeira do Amaral Resende

Doctoral Program in Electrical and Computer Engineering

Supervisor: Prof. Dr. Luis Miguel Pinho de Almeida

Co-supervisor: Dr. Waldir Aranha Moreira Júnior

May 15, 2026

Resumo

Estudos estatísticos recentes apontam para um crescimento contínuo na disseminação da Internet of Things, estimando um total de 50 mil milhões de ligações IoT em 2035. Esta disseminação é acompanhada por tecnologias, como o Fog e Edge Computing, nas quais as aplicações IoT seguem uma arquitectura orientada a serviços, composta por múltiplos componentes distribuídos pelas camadas que compõem o continuum IoT: Cloud, Fog, Edge e Dispositivos IoT. A gestão destas aplicações é um desafio relevante, tradicionalmente endereçado nas camadas de Cloud, Fog e Edge com tecnologias de orquestração da Cloud. Contudo, estas tecnologias de orquestração não consideram a camada de Dispositivos IoT como fornecedora de recursos computacionais que participam no processo de instanciação dos componentes da aplicação. Consideram-na apenas como fornecedora de serviços pré-existentes, utilizados no processo de encadeamento destes componentes. Esta limitação provoca que a instanciação e gestão dos componentes da camada de Dispositivos IoT seja feita em ferramentas de software dedicadas, dificultando a gestão eficiente da aplicação IoT como um todo.

A utilização de tecnologias de orquestração da Cloud para a gestão integrada das aplicações IoT enfrenta três problemas principais: 1) a heterogeneidade nos modelos de dados e de interacção utilizados pelos dispositivos IoT, que a ferramenta de orquestração necessita de endereçar para identificar correctamente cada dispositivo e qual o seu estado; 2) a heterogeneidade dos ambientes de execução dos dispositivos IoT, os quais a ferramenta de orquestração tem que suportar no processo de instanciação dos componentes da aplicação; 3) a diversidade de características de qualidade das aplicações IoT, que incluem, entre outras, características de qualidade dos dados, dispositivos e actuação, para além da qualidade de serviço e de experiencia de utilização, geralmente suportadas pelas ferramentas de orquestração da Cloud. Estes três problemas ainda não foram resolvidos pela comunidade. As propostas para endereçar a heterogeneidade baseiam-se em soluções proprietárias, o que resulta numa dependência desses fornecedores. Adicionalmente, as propostas para a diversidade de características de qualidade focam-se em casos de uso específicos, limitando a aplicação da ferramenta de orquestração a esses casos de uso, o que condiciona a sua disseminação.

Nesta Tese, endereçamos estes três problemas propondo o *Digital Orchestration of Things*, que é composto por três contribuições principais: 1) NextGenGW, que endereça a heterogeneidade de modelos de dados e interacção através do IETF *Semantic Definition Format*; 2) FITA, que integra os dispositivos IoT nas Kubernetes, endereçando a heterogeneidade de ambientes de execução e permitindo a instanciação de componentes das aplicações nesses ambientes, o que possibilita a orquestração do continuum IoT; 3) QOIA, um escalonador e gestor do ciclo de vida dos componentes da aplicação, que permite às Kubernetes orquestrar esses componentes considerando requisitos de qualidade configuráveis. A avaliação mostra a eficácia de cada contribuição e a sua aplicabilidade em casos de uso constituídos por um grande número de dispositivos heterogéneos.

Abstract

Recent statistical studies indicate that the Internet of Things vision of ubiquitous digitalisation is becoming a reality, with IoT connections expected to reach 50 billion by 2035. This widespread digitalisation is accompanied by technological developments such as Fog and Edge computing, where IoT applications are composed of multiple components distributed across the Cloud, Fog, Edge and IoT devices layers of the IoT continuum following a service-oriented approach. Managing such applications is a relevant challenge, which Cloud orchestration technologies have successfully addressed in the Cloud, Fog and Edge layers. However, Cloud orchestration currently considers the IoT devices layer just as a provider of services for service function chaining, and not as a pool of computing resources for application component deployment. Such a limitation relegates the management of components hosted by IoT devices to dedicated software tools, hindering the efficient management of the IoT application.

Three main problems hamper the realisation of IoT continuum orchestration: 1) the data and interaction model heterogeneity of IoT devices, which the orchestration framework should handle to know what a device is and its status; 2) the runtime heterogeneity of IoT devices, which the orchestration framework should support for application component deployment; and 3) the quality of IoT applications, which, among others, consider the data, device and actuation aspects of the application in addition to the computing and network aspects normally supported by Cloud orchestration. The community has not fully addressed these issues, as existing approaches rely on proprietary solutions to manage heterogeneity, leading to vendor lock-in. Additionally, it follows a use-case-centric approach regarding the quality of IoT applications, tying the orchestration to that specific use case, which hampers its dissemination.

In this Thesis, we address these issues by proposing the Digital Orchestration of Things, which is composed of three main contributions: 1) NextGenGW, which addresses data and interaction model heterogeneity using IETF Semantic Definition Format bound to MQTT; 2) FITA, which integrates IoT devices in Kubernetes and addresses runtime heterogeneity, enabling IoT continuum orchestration with application component deployment; and 3) QOIA, which is a configurable scheduler and runtime manager that enables Kubernetes to orchestrate considering IoT applications configurable quality requirements. Evaluation shows the effectiveness of each contribution and its applicability to use cases composed of a high number of heterogeneous devices.

Acknowledgements

First, I would like to thank my supervisors, Prof. Dr. Luís Almeida and Dr. Waldir Junior, for their valuable guidance and for sharing their knowledge.

I would also like to thank Filipe Sousa, my team lead at Fraunhofer Portugal AICOS, for having encouraged me to apply for the PhD. Without it, this document would probably not exist.

For all the fruitful discussions and companionship, I would like to thank João Oliveira, who began his own journey at the same time as I did. I would also like to thank José Costa for his contributions and insightful dialogues.

To my family and to my partner and best friend, I would like to thank them for their emotional support and patience during the most demanding stages of this journey.

Carlos Resende

Contents

List of Figures

List of Tables

Abbreviations

6LoWPAN IPv6 over Low -Power Wireless Personal Area Networks.

6lo IPv6 over Networks of Resource-constrained Nodes.

AAS Asset Administration Shell.

API Application Programming Interface.

AutomationML Automation Markup Language.

BLE Bluetooth Low Energy.

CAN Controller Area Network.

CBOR Concise Binary Object Representation.

CMIP Common Management Information Model.

CMOT Common Management Information Model Over TCP/IP.

CNCF Cloud Native Computing Foundation.

CoAP Constrained Application Protocol.

CoMI CoAP Management Interface.

CoRAL Constrained RESTful Application Language.

CoRE Constrained RESTful Environments.

CPU Central Processing Unit.

CRD Custom Resource Definitions.

DIS Draft International Standard.

DM Device Management.

DOLCE Descriptive Ontology for Linguistic and Cognitive Engineering.

DOT Digital Orchestration of Things.

DSL Domain-Specific Language.

ETSI European Telecommunications Standards Institute.

EXI Efficient XML Interchange.

FITA Far-edge IoT device mAnagement.

GPL General Purpose Language.

HTTP Hypertext Transfer Protocol.

IaaS Infrastructure as a Service.

IEC International Electrotechnical Commission.

IETF Internet Engineering Task Force.

IIoT Industrial Internet of Things.

IoT Internet of Things.

IP Internet Protocol.

ISO International Organization for Standardization.

ITU-T International Telecommunication Union - Telecommunication Standardization Sector.

JSON JavaScript Object Notation.

K8S Kubernetes.

LoRa Long Range.

lpwan IPv6 over Low Power Wide-Area Networks.

LwM2M Lightweight Machine to Machine.

MANO Management and Orchestration.

MEC Multi-access Edge Computing.

MQTT Message Queuing Telemetry Transport.

NAT Network Address Translation.

NETCONF NETwork CONFiguration Protocol.

NFV Network Functions Virtualization.

NGSI-LD Next Generation Service Interfaces-Linked Data.

OASIS Organization for the Advancement of Structured Information Standard.

OCF Open Connectivity Foundation.

OCI Open Container Initiative.

OGC Open Geospatial Consortium.

OMA Open Mobile Alliance.

OneDM One Data Model.

OPC-UA Open Platform Communications - Unified Architecture.

PaaS Platform as a Service.

PoC Proof of Concept.

QoA Quality of Actuation.

QoC Quality of Context.

QoD Quality of Data.

QoDe Quality of Device.

QoE Quality of Experience.

QoI Quality of Information.

QOIA Quality-aware Orchestration for IoT Applications.

QoIoT Quality of Internet of Things.

QoO Quality of Output.

QoS Quality of Service.

QoSe&P Quality of Security and Privacy.

QoT Quality of Things.

RDF Resource Description Framework.

REST Representational State Transfer.

RESTCONF Representational State Transfer Configuration Protocol.

RFC Request for Comments.

ROLL Routing Over Low power and Lossy networks.

RQ Research Question.

RTU Remote Terminal Unit.

SAREF Smart Appliances REFerence ontology.

SDF Semantic Definition Format.

SDN Software Defined Networks.

SenML Sensor Measurement List.

SensorML Sensor Model Language.

SLA Service Level Agreement.

SNMP Simple Network Management Protocol.

SOSA Sensor, Observation, Sample, and Actuator.

SSN Semantic Sensor Network Ontology.

STA SensorThings API.

SUMO Suggested Upper Merged Ontology.

T2TRG Thing-to-Thing Research Group.

TCP Transmission Control Protocol.

TD Thing Description.

TLS Transport Layer Security.

TOSCA Topology and Orchestration Specification for Cloud Application.

UFO Unified Foundational Ontology.

UI User Interface.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

VoI Value of Information.

W3C World Wide Web Consortium.

WAMP Web Application Messaging Protocol.

Wasm WebAssembly.

WoT Web of Things.

XML Extensible Markup Language.

XMPP Extensible Messaging and Presence Protocol.

YAML Yet Another Markup Language.

YANG Yet Another Next Generation.

Chapter 1

Introduction

IoT Analytics estimates that in 2024 there were 18.5 billion Internet of Things (IoT) connections[1], which represents a growth of 12% compared to 2023. The 2025 estimate, considering the first half of the year, was for an expected growth of 14%, reaching 21.1 billion Internet of Things (IoT) connections by the end of the year. The forecast is that by 2035, the number of IoT connections will grow up to 55 billion.

This data shows that the IoT vision of ubiquitous digitalisation is becoming a reality. The concept of IoT encompasses a set of devices that sense the environment around us, (pre-)process such data and/or transmit it to a device richer in computing resources for further processing and storage, and actuate according to the result of the processing stage(s), working autonomously to improve the efficiency of tasks currently performed manually [32]. The IoT applications providing such automation are composed of multiple services (also referred to as application components in this Thesis) spread across the devices that compose the IoT installation and cover the Cloud, Fog, Edge and IoT device layers of the IoT stack (see Figure 1.1). These application components are interconnected in a service-oriented manner to guarantee the fulfilment of the IoT application purpose, composing what is known as the IoT computing continuum [52].

In other words, modern IoT applications do not follow a Cloud-only approach in which gateways connect IoT devices (the elements of an IoT installation that sense and/or act on their environment, which are referred to as "Things" in the title of this Thesis) to the Cloud, where data is stored and processed. A Cloud-only approach is not the most appropriate strategy for addressing the current vision of full-fledged IoT deployments, especially with the forecasted growth in the number of IoT connections. The high bandwidth usage, communication latency, and the lack of response time guarantees are some reasons why Cloud-only approaches cannot meet the requirements of current IoT deployments. Edge and Fog computing, which aim to have data processed (and possibly stored) close to its origin instead of being forwarded to the Cloud, are being used to fill in the gap between IoT devices and the Cloud [113, 42, 114].

With the evolution of this approach, the concept of Edge computing has expanded and currently considers the migration of computing tasks not only to small data centres or networking de-

[1] https://iot-analytics.com/number-connected-iot-devices/

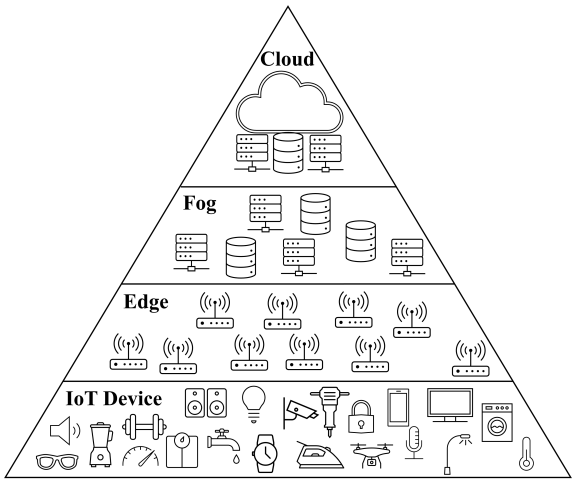


Figure 1.1: IoT four-layer stack architecture.

vices near data sources, which compose the Edge and Fog layers of Figure 1.1, but to the devices that compose the IoT Device layer. The ongoing optimisation efforts to bring AI into microcontrollers, such as Google LiteRT for Microcontrollers[2] and the Edge Impulse platform[3], allied with the growing processing capabilities of new microcontroller architectures, allows even the resource constrained IoT devices to perform on-site computing in addition to sensing and actuation [55].

Nevertheless, despite this evolution in IoT applications and IoT device dissemination, the IoT devices are still, traditionally, delivered to customers with pre-installed and/or pre-configured software, and updating them normally is a manual process that requires physical access to the device. This procedure poses a deployment and management problem because updating IoT devices manually and locally is too challenging and complex, considering the envisioned dense IoT device networks [22, 70]. A set of management platforms addresses this issue, allowing for remote management of IoT devices [41, 101]. However, management with these platforms is still performed manually and disassociated from the other elements of the IoT stack, i.e., multiple platforms need to be used to manage the application components and devices located on the IoT device layer and the application components and devices located in Cloud, Fog, and Edge layers.

In the scope of "traditional" Edge computing, i.e., the one that does not consider offloading computation to IoT devices, a set of existing data centre technologies is being explored to support the autonomous management of application components spread between the Cloud and Edge, namely their scalability, flexibility, and life cycle management [84, 113, 42]. The concepts of Platform as a Service (PaaS) and Infrastructure as a Service (IaaS), typical of data centre technologies, are explored in Edge and Fog installations. They are being explored to address application packaging, orchestration, resource sharing, and isolation. Orchestration is a vital approach within this scope, allowing providers to provision, configure, and manage their applications in a more automated manner.

[2] https://ai.google.dev/edge/litert/microcontrollers/overview
[3] https://edgeimpulse.com/product

1.1 Motivation

Current efforts on extending the operation of Cloud orchestration frameworks to the Fog and Edge layers do not yet fully address the IoT Devices layer. Such limitation hampers the realisation of the IoT computing continuum, whose IoT devices might host computing application components in addition to sensing and actuation. Among the different challenges faced when enabling IoT devices in Cloud orchestration frameworks, we highlight their heterogeneity in terms of interaction and data models, as well as runtime environments, and the diversity of resources that differentiates them from the devices of the other layers of the IoT stack. Examples of such resources are the sensors and actuators, which are currently not properly considered by the orchestration solutions despite being crucial for IoT applications to work correctly [76, 108].

Regarding heterogeneity, one of the issues is the number of manufacturers in the IoT computing continuum from the IoT device to the Cloud. In this landscape, standard technologies are mandatory to avoid vendor lock-in. Some standardisation bodies are targeting the IoT domain, particularly its heterogeneity in data representation, communication, encoding, and management [74, 35, 102]. However, such efforts are mainly heterogeneous themselves, normally focusing on certain application domains or use cases, with most domain-independent and interoperability efforts being very recent. Considering this, the community has not yet provided an interoperability solution that effectively addresses this issue in a standard way. Focusing on the heterogeneity regarding runtime environments, the current Cloud orchestration framework handles such heterogeneity in the Cloud, Fog and Edge through container virtualisation, which is standardised by the Open Container Initiative (OCI)[4]. However, not all IoT devices have enough resources to handle OCI runtimes, which makes runtime environment heterogeneity an issue for the integration of IoT devices in Cloud orchestration [76].

When the resources of IoT devices, intrinsic to IoT deployments, are added to this problem, an extra layer of complexity emerges. The status of the resources of such IoT devices and their characteristics are related to the quality of IoT applications, introducing new challenges regarding topology definition, data representation, provisioning and life cycle management. The community has already identified that traditional Quality of Service (QoS) and Quality of Experience (QoE) are not sufficient to model the quality of IoT applications, and focused on identifying and defining which are their quality characteristics, considering they might not have a human as a consumer, run on heterogeneous devices and be composed of multiple components featuring sensing, computing, communication, and actuation aspects, each with its own quality metrics [32, 62, 99]. For this reason, new terms, such as Quality of Things (QoT), Quality of Internet of Things (QoIoT) or Quality Enabled IoT Applications, were proposed to define the quality aspects of IoT applications [32]. Such qualities should now be considered during provisioning, either in what respects the assignment of the application components to IoT devices or the selection of already running application components, to guarantee the fulfilment of the IoT application quality requirements.

[4] https://opencontainers.org/

Monitoring the metrics that compose such qualities and acting accordingly is also fundamental to guarantee that the IoT application meets the expected quality level during its life cycle.

In summary, the dissemination of IoT digitalisation should be accompanied by new tools and technologies to ease the provisioning and life cycle management of IoT devices because non-automated approaches result in a less efficient solution, with high installation and life cycle management overhead. Orchestration is a core concept in this scope, which could provide automated provisioning and life cycle management. However, the mature Cloud-based orchestration technologies do not yet cover the full IoT stack. Community efforts have extended Cloud orchestration to the Edge by addressing some of the issues on this layer. Nevertheless, IoT devices bring a set of new challenges, such as their heterogeneity and the quality requirements of their applications.

To address such challenges, we propose the development of the Digital Orchestration of Things (DOT), which is an orchestration framework for the digitalisation and corresponding automation of the provisioning, from the perspective of application component placement, and life cycle management of IoT applications by dealing with IoT devices heterogeneity and IoT application quality requirements.

1.2 Thesis Statement and Research Questions

Considering the context and motivation expressed before, we state our Thesis as follows:

"Improving existing Cloud orchestration tools, such as Kubernetes, to consider IoT devices heterogeneity and IoT applications quality requirements, allows such devices to be considered legitimate resources for application component placement and life cycle management, enabling IoT continuum orchestration."

We consider Kubernetes (K8S) Cloud orchestration framework[5] for our Thesis, because it is a mature and open-source orchestration system whose adoption rate is constantly increasing[6]. It also contains a working group focused on exploring Edge orchestration[7]. These characteristics make it an interesting tool to build Cloud to IoT Device continuum orchestration. Additionally, K8S provides a set of extension mechanisms that allow adding new functionalities through a plugin or addon-based approach, eliminating the need to modify its core components or functionalities. This plugin and addon-based approach maximises non-intrusiveness and preserves existing K8S usage patterns, allowing system operators to use IoT functionalities as optional addons or plugins without disturbing their Cloud and Edge installations.

To validate our Thesis, we examined such K8S extension mechanisms seeking answers to the following research questions:

[5] https://kubernetes.io/
[6] https://www.cncf.io/reports/cncf-annual-survey-2023/
[7] https://tag-runtime.cncf.io/wgs/iot/

RQ1: *Can standards-based approaches be used to address IoT device heterogeneity and avoid vendor lock-in?*

IoT devices are highly heterogeneous in their interaction and data models (both syntactically and semantically), which poses an interoperability problem. We should use standards to address such a problem without facing the risk of vendor lock-in. This interoperability problem leads to two sub-questions:

- RQ1.1: Which are the most appropriate standards to address IoT device heterogeneity regarding interaction and data models?
- RQ1.2: Are there community proposals addressing interoperability that use such standards?

Regarding RQ1.1, we explore current standard proposals, considering their modelling objectives and use cases. For the scope of IoT orchestration, the standard should handle the multiple IoT entities and interactions. Additionally, ideally, they should not be focused on a specific use case so as not to limit the applicability of the orchestration system. RQ1.2 explores whether the community already considers such standards in the proposed interoperability solutions, which we could use or adapt for the scope of IoT orchestration. The answer to RQ1.2 did not retrieve a viable solution, so we explore the viability of developing such an interoperability solution.

RQ2: *Can Cloud orchestration systems handle the heterogeneity of IoT devices runtime environments?*

IoT devices may operate using virtualisation technologies, operating systems, or bare metal architectures. K8S, the Cloud orchestration system we identified as baseline, is tailored to handle virtualisation technologies based on the OCI specification. However, only a subset of the IoT devices has enough resources to run such technologies. IoT devices based on microcontrollers normally run real-time operating systems or bare metal architectures, some providing runtime environments for application component deployment, such as scripting interpreters, virtual machines, and dynamic loadable components. In RQ2, we explore K8S and whether it can be extended to handle such heterogeneity in runtime environments. We use the outputs of RQ1 to feed the work on this RQ.

RQ3: *Can Cloud orchestration systems be extended to consider the quality characteristics of IoT applications?*

IoT applications quality depends on multiple parameters due to their autonomous, heterogeneous and distributed nature, covering multiple aspects, such as sensing, computing, communication, and actuation, each with its own quality requirements. One of the main goals of orchestration is to guarantee that the various components that compose an application are provisioned according to their quality requirements. After provisioning, the orchestration system might autonomously adapt the IoT installation if the IoT application components are no longer meeting their quality requirements. Considering this, we break RQ3 in two sub-questions:

- RQ3.1: Which are the quality characteristics and metrics that define quality for IoT applications?
- RQ3.2: How are such quality characteristics and metrics handled by the existing application provisioning solutions?

RQ3.1 relates to the identification of which quality characteristics should be considered when extending Cloud orchestration systems to support IoT devices and IoT applications. Additionally, answering RQ3.1 indicates if the quality of IoT applications is defined by a closed set of characteristics and metrics, or if the orchestration solution should be adaptable to accommodate qualities and metrics that change according to the use case or applications they are related to. RQ3.2 explores how quality has been addressed by existing IoT provisioning solutions, if and how such solutions can be used in the scope of IoT orchestration, what their limitations are, and how we overcame such limitations.

1.3 Contributions and Publications

The RQs in Section 1.2 defined the path towards achieving the contributions of our Thesis, which were already shared with the research community in a set of scientific publications. Figure 1.2 shows this relationship between Thesis, RQs, contributions and the publications that support them. The contributions are the following:

C1.1: *Interoperability framework for IoT devices interaction and data models*
and
C1.2: *Evaluation baseline for the performance of IoT gateways targeting interoperability*

The work on RQ1 produced contributions C1.1 and C1.2. C1.1 results from the answers to RQ1.1 and RQ1.2, which identified Web of Things (WoT) Thing Description (TD) and Internet Engineering Task Force (IETF) Semantic Definition Format (SDF) as the most appropriate standards to address IoT device heterogeneity regarding interaction and data models. These answers also showed that the community proposals addressing interoperability do not use such standards. C1.1 proposes NextGenGW, which uses IETF SDF bound with MQTT to address IoT device heterogeneity regarding interaction and data models. While evaluating NextGenGW, we faced constraints in comparing its performance against the other proposals in the literature because each proposal used a specific set of evaluation metrics, making them hardly comparable. Considering this, C1.2 proposes a performance evaluation baseline that can be used by new interoperability proposals, aiming to ease their quantitative comparison.

Publications supporting the contributions:

- Carlos Resende, Waldir Moreira, and Luís Almeida. NextGenGW: a Software-based Architecture Targeting IoT Interoperability. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4, 2022 *(Best WiP for Emerging Technologies)*

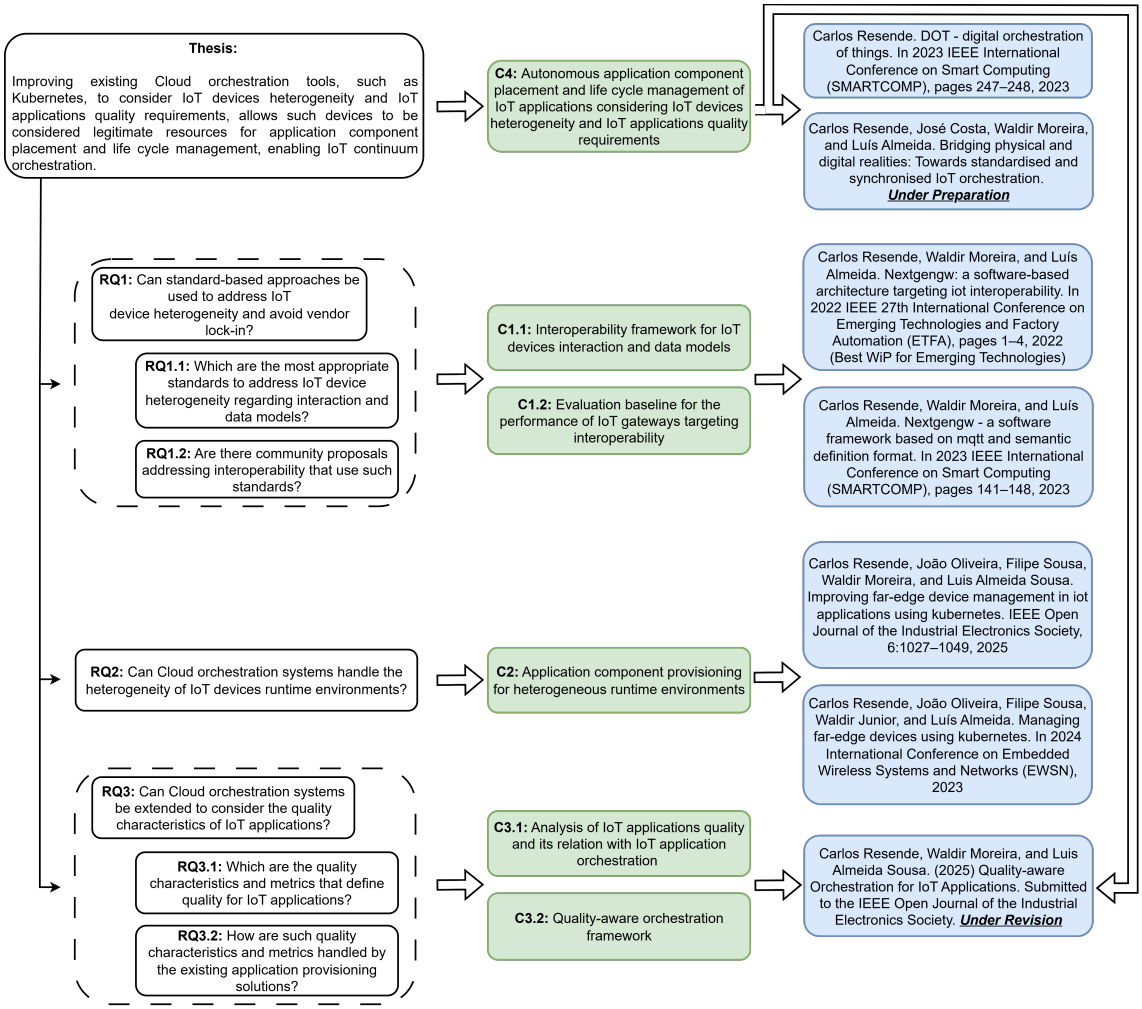


Figure 1.2: Relationship between DOT's Thesis, Research Questions, Contributions and Publications (Thesis and Research Questions in white, Contributions in green and Publications in blue).

- Carlos Resende, Waldir Moreira, and Luís Almeida. NextGenGW - a Software Framework Based on MQTT and Semantic Definition Format. In *2023 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 141–148, 2023

C2: *Application component provisioning for heterogeneous runtime environments*

Contribution C2 emerges from RQ2. While working on RQ2, we found that not only are the OCI standard runtimes non-compliant with resource-constrained IoT devices, but the K8S Kubelet is also not compliant with them. K8S Kubelet is the agent that runs on the K8S compliant devices and, among other things, is responsible for deploying application components on the OCI runtimes. This means that working on the Kubelet to allow it to handle the heterogeneity of runtime environments provided by IoT devices was not a solution. Considering such limitations, we propose the Far-edge IoT device mAnagement (FITA) in C2, which is a mechanism based on K8S virtual kubelet[8] and contribution C1.1, i.e., NextGenGW, which abstracts the IoT device heterogeneous runtime environments, allowing their integration in the K8S cluster. This integration is achieved without modifying the K8S control plane, which treats IoT devices in the same way as the devices of the other layers of the IoT stack, while allowing the deployment of application components on them independently of their runtime environment. This contribution is available open-source at: https://github.com/fraunhoferportugal/fita

Publications supporting the contribution:

- Carlos Resende, João Oliveira, Filipe Sousa, Waldir Moreira, and Luís Almeida Sousa. Improving Far-Edge Device Management in IoT Applications Using Kubernetes. *IEEE Open Journal of the Industrial Electronics Society*, 6:1027–1049, 2025
- Carlos Resende, João Oliveira, Filipe Sousa, Waldir Junior, and Luís Almeida. Managing Far-Edge Devices using Kubernetes. In *2024 International Conference on Embedded Wireless Systems and Networks (EWSN)*, 2024

C3.1: *Analysis of IoT applications quality and its relation with IoT application orchestration* and

C3.2: *Quality-aware orchestration framework*

Contributions C3.1 and C3.2 result from the work on RQ3. The answer to RQ3.1 revealed that the quality characteristics of IoT applications, as well as their metrics, are highly dependent on the application. In C3.1, we argue that it is possible to define a macro set of quality characteristics that hold several quality metrics, but the definition of the characteristics of interest and their concrete metrics is application-dependent. Additionally, the answer to RQ3.2 showed that current provisioning solutions do not handle such dynamism, despite, in our opinion, the relevance of handling it in the scope of application orchestration to avoid narrowing the operation scope to certain use cases or applications. Considering this, C3.2 proposes the Quality-aware Orchestration for IoT

[8] https://virtual-kubelet.io/

Applications (QOIA), which handles IoT application quality requirements during provisioning (application component placement) and life cycle monitoring, adopting a configurable approach that adapts to the quality goals of each application.

Publication supporting the contributions:

- *Carlos Resende, Waldir Moreira, and Luis Almeida Sousa. (2025) Quality-aware Orchestration for IoT Applications. Submitted to the IEEE Open Journal of the Industrial Electronics Society.* ***Under Revision****".*

C4: *Autonomous application component placement and life cycle management of IoT applications considering IoT devices heterogeneity and IoT applications quality requirements*

Contribution C4 holds DOT, which is the solution that validates our Thesis and is composed of contributions C1.1, C2 and C3.2. These contributions, correctly integrated, build the core infrastructure of DOT, contributing to solving the problem of IoT orchestration and validating our Thesis.

Contribution C4 goes beyond the contributions that compose it and adds a set of relevant insights about IoT continuum orchestration, towards further enhancements concerning IoT devices integration in Cloud orchestration systems.

Publications supporting the contribution:

- Carlos Resende. DOT - Digital Orchestration of Things. In *2023 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 247–248, 2023
- *Carlos Resende, José Costa, Waldir Moreira, and Luís Almeida. Bridging Physical and Digital Realities: Towards Standardised and Synchronised IoT Orchestration.* ***Under Preparation***
- *Carlos Resende, Waldir Moreira, and Luis Almeida Sousa. (2025) Quality-aware Orchestration for IoT Applications. Submitted to the IEEE Open Journal of the Industrial Electronics Society.* ***Under Revision****.*

1.4 Document Structure

The remainder of this document is organised as follows:

- Chapter 2 reviews the state of the art on IoT orchestration solutions, the standardisation efforts addressing IoT interoperability and the community work on defining quality for IoT applications and developing provisioning platforms that handle it.
- Chapter 3 presents NextGenGW, our contribution to address IoT heterogeneity regarding interaction and data models.

- Chapter 4 presents FITA, our proposal to integrate IoT devices in the K8S cluster, and deal with their heterogeneous runtime environments.
- Chapter 5 presents QOIA, which is our proposed configurable scheduler and life cycle monitoring solution that is not tied to a specific set of quality characteristics, metrics or models.
- Chapter 6 concludes the document with an overview of DOT and how it validates our Thesis, as well as the identification of a set of future research directions on IoT continuum orchestration.

Finally, we note that we refer to URLs in footnotes that are numbered sequentially per chapter. We leave for the References section just the references to published literature and standardisation documents.

Chapter 2

State of the Art

Our Thesis proposes a solution to the problem of IoT orchestration, addressing the challenges related to the heterogeneity of IoT devices regarding interaction and data models, as well as runtime environments, and the quality requirements of IoT applications. To build such a novel solution, a set of technological areas was surveyed, namely: 1) the IoT orchestration itself, its architecture and properties, and how the community is addressing the IoT continuum orchestration (Section 2.1); 2) the set of standard technologies that can be used to interact with IoT devices to address their interoperability and avoid vendor lock-in (Section 2.2); and 3) the Quality characteristics intrinsic to IoT devices and IoT applications, exploring how they should be considered in the scope of IoT orchestration for it to be effective (Section 2.3).

2.1 IoT Orchestration Frameworks

A framework that supports the orchestration of IoT applications needs to guarantee a set of features that facilitate the creation of orchestration plans, the deployment of such orchestration plans, the monitoring of the IoT installation to guarantee it copes with the quality requirements defined in the orchestration plan, and the actuation over the IoT installation to adapt it when it deviates from those requirements, or when a new orchestration plan is available. The heterogeneity intrinsic to IoT devices and to the quality requirements of IoT applications add more complexity to the development of such frameworks [108, 114, 76].

This section covers the academic and community efforts on the development of solutions for IoT orchestration. To analyse them, we use the taxonomy for orchestration of IoT systems proposed by Nguyen et al. [76]. Section 2.1.1 overviews the taxonomy, and Section 2.1.2 presents the literature review. Section 2.1.3 closes the orchestration topic with the description of the IoT orchestration framework proposed with DOT and how we position it in the scope of IoT orchestration.

2.1.1 Feature Taxonomy for IoT Orchestration Frameworks

Four major groups of features compose the taxonomy proposed by Nguyen et al. [76]: deployment support, orchestration support, design support, and advanced support. This section briefly overviews them.

Deployment Support

An IoT orchestration framework should guarantee full support for deploying IoT application components following the service-oriented paradigm. In this scope, one can consider that the various IoT devices are executing a service-oriented runtime, which allows the deployment or update of IoT application components. Considering this, we can classify an IoT framework according to its:

- Deployment support — can be declarative or imperative. In the declarative approach, the user specifies the desired deployment, and the scheduler autonomously computes the best way to reach it (e.g., the user specifies the required application components, including their hardware, runtime and quality requirements, and then, a scheduling algorithm associates the application components with the resources that will host them). In the imperative approach, the user completely defines the deployment, and the deployment engine only needs to follow the specified recipe.
- Target support — specifies the infrastructure supported by the framework: Cloud, Edge, Fog, and IoT devices.
- Network — identifies the network aspects supported by the framework, such as custom addressing, virtual networks, Software Defined Networks (SDN), and communication protocols, including Wi-Fi, Bluetooth, and other IoT standards.
- Cloud provisioning — indicates if Cloud resources can also be provisioned and orchestrated, as well as the support for Cloud virtualisation services, such as the IaaS, Database as a Service, PaaS, etc.
- Bootstrap — specifies the software that needs to be installed in the IoT devices to allow the orchestration framework to manage them.

Orchestration Support

After deployment, the orchestration framework should be able to guarantee the connection between the various components that compose the application. In this sense, we can classify the orchestration framework according to its:

- Communication support — refers to the ability of the orchestration framework to accommodate or manage the types of communication models (such as message passing, message queues, publish/subscribe or invocation of remote methods) that the application components might be using.

- Integration support — refers to the capabilities of the orchestration framework to enable components to be connected and made able to communicate. It includes infrastructure-level setup, such as networking and port exposure.

Design Support

The IoT orchestration framework needs to be fed with an orchestration plan to perform the provisioning and management of the IoT application. IoT orchestration frameworks can be classified according to the support they provide for the design of such an orchestration plan, namely:

- Specification language — the user can define the orchestration plan according to a General Purpose Language (GPL) or a Domain-Specific Language (DSL).
- Specification capabilities — indicate which details of the IoT system the language can specify. This is particularly important because, for example, the correct definition of an application component, including its hardware and quality requirements, allows its correct deployment, configuration, and management.

Advanced Support

Nguyen et al. [76] include all the features that go beyond the simple definition of an IoT application, its deployment and interactions, in the advanced support group. Features such as monitoring the status of the IoT system, security characteristics, and multi-tenancy support are considered advanced features of an IoT orchestration solution. In this scope, Nguyen et al. divide the advanced support into:

- Trustworthiness — indicates if the orchestration framework considers features such as security, reliability, and resilience.
- Adaptation — indicates if the orchestration system can do runtime adaptations. Such adaptation can be materialised by simple parametric adaptation, with the parameters being defined at design time, or compositional adaptation, where new software models are deployed.
- Monitoring — to cope with quality requirements, the IoT orchestration frameworks need to keep track of the IoT system status. In this scope, the support for monitoring solutions is an important feature.
- Shared access to resources — supporting multiple tenant scenarios demands precise control over access to shared resources. In addition to the previously referred security aspects, it must be guaranteed that the tenants do not conflict with each other by, for example, performing actions that indirectly or directly impact the environment where the other tenants are operating.

Still in this scope, Velasquez et al. [108] present a set of requirements and corresponding challenges to consider when developing Fog orchestration frameworks. We can map such requirements and challenges to the taxonomy proposed by Nguyen et al. [76], complementing it.

Resource management, performance, and security management are the requirements identified by Velasquez et al. [108]. Each of these requirements contains a set of challenges, namely:

- **Resource management** contains the scheduling and path computation challenges. Scheduling relates to associating the requirements of the application components with the available resources. Path computation relates to the connectivity between application components. These challenges can be associated with Nguyen et al. [76] deployment support feature. Still, regarding resource management, Velasquez et al. [108] identify the resource discovery and allocation, as well as system interoperability challenges, which we map to the advanced support feature of the taxonomy proposed by Nguyen et al. [76].
- **Performance** contains the latency, resilience, prediction, and optimisation challenges. This is a requirement and set of challenges associated with the support for quality guarantees, thus linked with the advanced support feature of taxonomy proposed by Nguyen et al. [76].
- **Security management** contains the challenges related to security, privacy, and authentication, thus linked to the advanced support feature of the taxonomy proposed by Nguyen et al. [76], more precisely, the trustworthiness aspect.

Velasquez et al. [108] requirements and challenges enhance Nguyen et al. [76] analysis on IoT orchestration frameworks. They mainly contribute to the advanced support feature by adding a set of new aspects, such as resource discovery and interoperability. Additionally, they expand the analysis on the aspects of trustworthiness and monitoring.

2.1.2 Literature Review on IoT Orchestration Frameworks

The literature review on IoT orchestration frameworks that we present here is composed of the analysis of eight academic proposals, three surveys and four industrial initiatives.

GENESIS [31] is a framework for the declarative orchestration of IoT devices. Despite the authors' statement indicating that the framework supports the provisioning of IoT devices, Edge and Fog nodes, and Cloud components, the framework focuses on addressing the orchestration of IoT devices, with its use cases and requirements. Arduino and Docker are the supported runtimes. The authors state that support for other runtimes can be added, but without details on how to do it. The authors propose a gateway that utilises a Node-RED application to deploy application components on devices without an Internet connection, i.e., devices whose updates are performed via a gateway that is connected to the Internet. GENESIS framework uses a DSL created by the authors to define the orchestration plan. They state that the DSL is flexible to include more features than those already supported and reported in the publication. However, the currently supported feature set is limited, since the platform does not consider QoS, security, and other advanced features for the orchestration of IoT applications. Additionally, the authors do not evaluate the proposed solution, so its caveats and tradeoffs are not fully explored.

FITOR [25] is a three-layer descriptive orchestration framework focused on the Fog Environment. Figure 2.1 presents the three-layer architecture, which is composed of: Fog Node, that

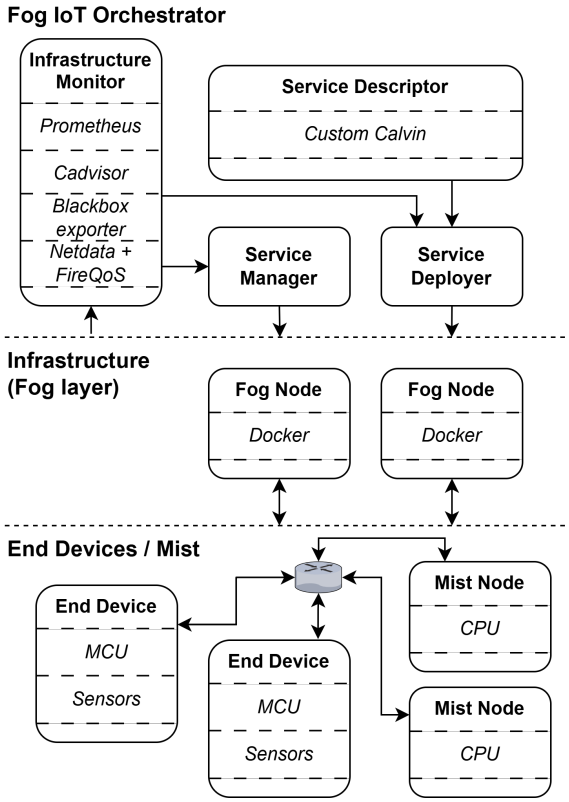


Figure 2.1: FITOR architecture [25].

provides nearby computational and storage resources; Mist Nodes, specialised nodes that are even closer to the IoT devices and provide low computational resources; and End Devices, which are sensors and actuators that sense and act over the environment (i.e., IoT devices in our nomenclature). Despite this layered architecture considering Fog nodes, Mist nodes, and IoT devices, the framework only supports the provisioning of containerised nodes (more precisely Docker containers), which, according to the authors, are just represented by the Fog Nodes. Five key components compose the FITOR architecture:

- The Service Descriptor that describes the application, namely its components and requirements. The authors state it includes features such as QoS and network-related requirements.
- The Service Deployer, which comprises the orchestration algorithm.
- The Service Manager that manages the deployed system, considering the monitored status (from the application components, nodes, and communication links).
- The Infrastructure Monitor that monitors the system, providing information about the resources available in the nodes, such as CPU, RAM, and disk, and in the network, such as latency and bandwidth.
- The Fog node is the Docker-based device of the Fog layer.

FITOR makes use of a customised version of Calvin[1] for application component specification, so it makes use of a proprietary DSL. According to the authors, the extensions introduced to Calvin added QoS related features to the application component specification, such as RAM, CPU, bandwidth usage, and network latency. Prometheus[2] is used for system monitoring, along with Cadvisor[3], blackbox exporter[4], Netdata[5] and FireQoS[6]. Along with the framework, the authors propose a provisioning algorithm that calculates the optimal provisioning strategy for the minimum amount of resource usage, considering metrics such as residual processing power, residual memory, and geographic location. Although theoretical support exists for QoS features, the authors do not validate it.

Alam et al. [2] propose a modular and scalable architecture based on Docker virtualisation to orchestrate microservices in IoT. The solution proposed by the authors envisions managing and orchestrating the resources and application components of the Cloud to Edge continuum (which, in the authors' perspective, includes the IoT devices). For this, the authors assume that all the devices along this continuum, including the IoT devices, will support Docker runtime.

The authors propose the architecture in Figure 2.2, which is composed of three components: (1) Edge, where the Smart IoT Devices (i.e., the IoT devices in our nomenclature) are located. This devices run Docker Worker[7] and directly sense and interact with the environment; (2) Fog, composed of the gateways that run Docker Worker and offer network, computational and storage resources close to the Edge; and (3) Cloud composed of the enterprise systems that run Docker Swarm[8] and offer high computational and storage resources, while managing the containers on the other layers. A messaging hub connects the devices and application components in the proposed architecture. The message hub is based on the SDN approach to distinguish the data and control traffic in different channels. The authors do not provide details about the functioning of each channel, so it is unclear which communication models are supported. Despite the validation in an application scenario, the proposed solution is in a very embryonic stage. The implementation details are unavailable or not validated for many of the presented (envisioned) features. The authors also do not address important aspects of IoT orchestration, namely, they do not specify: how is the orchestration plan defined, neither what is the used specification language; if QoS requirements are supported, and how is the system adapted according to them; or how are the resources and application components added to the system.

Brito et al. [23] start by analysing application component orchestration solutions for the Cloud, network, and other settings to understand the respective orchestration scopes, and how they match the Fog requirements. To accomplish this, the authors analyse ETSI NFV MANO (European

[1] https://github.com/EricssonResearch/calvin-base
[2] https://prometheus.io/
[3] https://github.com/google/cadvisor
[4] https://github.com/prometheus/blackbox_exporter
[5] https://www.netdata.cloud/
[6] https://github.com/firehol/firehol/wiki/FireQOS
[7] https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/
[8] https://docs.docker.com/engine/swarm/swarm-tutorial/

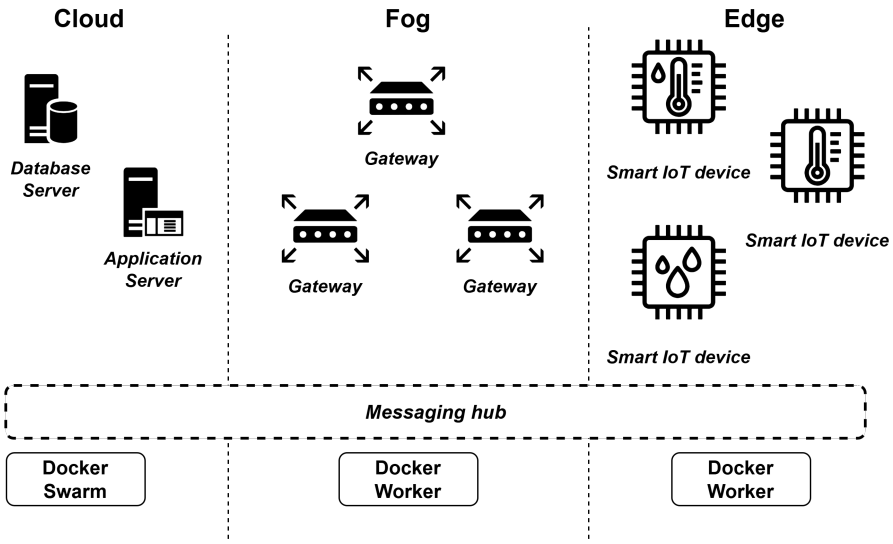


Figure 2.2: Alam et al. proposed architecture [2].

Telecommunications Standards Institute Network Functions Virtualization Management and Orchestration) Reference Architecture[9], container orchestration, ETSI Multi-access Edge Computing (MEC) Reference Architecture[10], OpenFog Reference Architecture [19] and TOSCA [60]. The authors conclude that none of the studied architectures and solutions fully cover the requirements for orchestrating Fog deployments. They just briefly mention orchestration, with no concrete approach on how to reach it (this is the case of OpenFog Reference Architecture and ETSI MEC Reference Architecture) or the proposed orchestration scheme does not cover the Fog requirements (this is the case of ETSI NFV MANO Reference Architecture, container orchestrations and TOSCA).

To fill this void, Brito et al. propose the architecture for Fog orchestration presented in Figure 2.3. Two essential components are part of this architecture: the Fog Orchestrator Agent, which must be running in every Fog Node; and the Fog Orchestrator, responsible for managing and controlling the Fog Orchestrator Agents. The Fog Orchestrator Agent is responsible for managing the resources available in the nodes and controlling their containers according to the instructions received from the Fog Orchestrator. It is a kind of middleware that abstracts the virtualisation technology supported by the node, assesses the resources available in the node, and advertises them to the Fog Orchestrator while monitoring them at runtime. The Fog Orchestrator keeps a database of the available resources and calculates the orchestration plan based on such resources and the orchestration template. It is also responsible for deploying the orchestration plan by interfacing with the Fog Agents.

The authors identify some open questions not yet addressed in their work, namely the communication infrastructure between Fog Orchestrator and Fog Orchestrator Agent, along with the monitoring components and virtualisation technologies. The authors identify the importance of

[9] https://www.etsi.org/technologies/open-source-mano
[10] https://www.etsi.org/technologies/multi-access-edge-computing

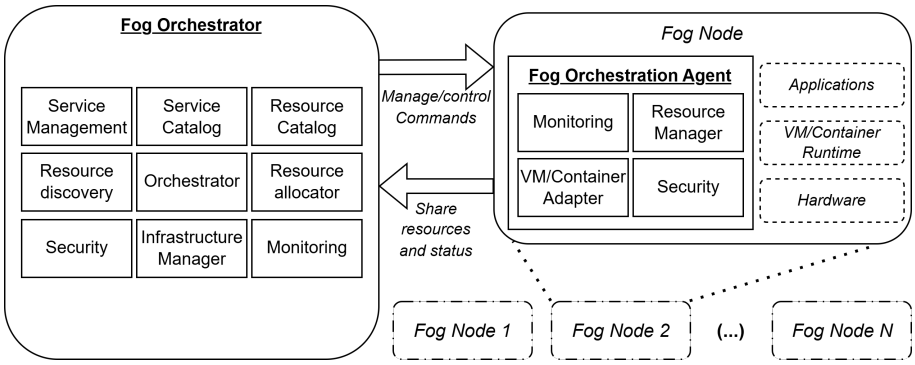


Figure 2.3: Brito et al. proposed architecture [23].

using standard communication protocols between these elements to address their heterogeneity. The authors also fail to address the mechanisms for discovering resources and application components. TOSCA 1.0 is the model language used by the authors for creating the orchestration plan, so they based the proposed solution on a declarative DSL. The authors built a prototype of the proposed architecture, but the prototype validation and characterisation (assessing the impact of the For Orchestrator and Fog Orchestrator Agent on resource usage) are missing.

Jiang et al. [49] propose a high-level definition of an orchestration framework for the Fog environment. The proposed architecture, presented in Figure 2.4, is composed of three layers:

- The Physical resource layer contains the sensors and actuators that sense and act in the environment, i.e., the IoT devices.
- The control layer manages the devices in the Physical resource layer. The authors propose the creation of clusters of devices in the Physical Resources Layer based on their ownership and operation. The objective is to distribute the control responsibilities between multiple controllers.
- The service layer abstracts the heterogeneity and details of the Fog infrastructure to the end-users.

The layers establish vertical communication links between each other, and horizontal communication links to the devices of the same layer.

The authors identify the need for a self-measurement module in the Control layer to monitor its status and guarantee QoS requirements. In addition to the QoS requirements, the need to support locality and performance requirements is also identified by the authors. Since the authors do not present implementation details but a theoretical architecture, it is impossible to characterise the proposed solution according to the taxonomy presented in Section 2.1.1.

Yangui et al. in [113] propose the architecture in Figure 2.5, which supports and automates the provisioning of applications in a hybrid Cloud/Fog environment. Four layers compose the architecture:

- Application development layer - includes an IDE that allows application development, testing, and deployment.

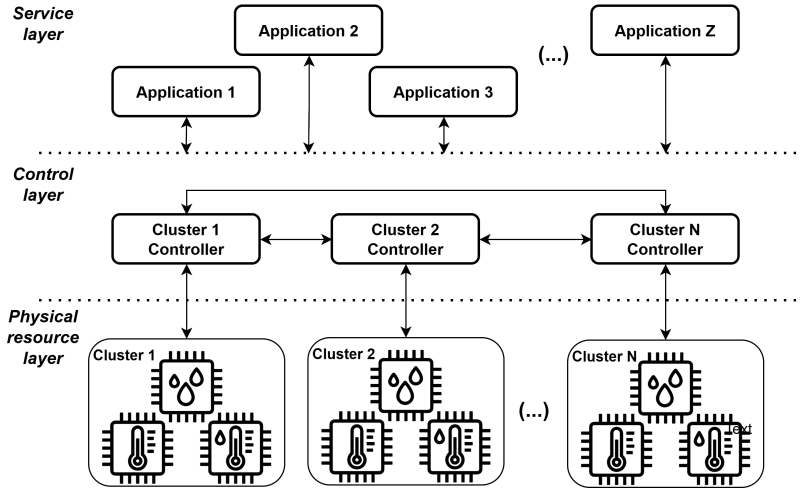


Figure 2.4: Jiang et al. proposed architecture [49].

- Application deployment layer - it is composed of: the deployer that discovers and instantiates the Fog resources able to host the application components. It also deploys such application components over the defined resources, in a dedicated service container; the controller, which, with the deployer, supervises the configuration of the instantiated hosting resources; the Fog resources repository, which stores the available hosting resources in the Fog domain.
- Application hosting and execution service - comprises the orchestrator that guarantees the execution flow, and coordinates the message exchange between the containers.
- Application management - interacts with the other three modules, guaranteeing Service Level Agreement (SLA) and QoS requirements through application component migration or elasticity.

A set of three interfaces connects the various elements of the architecture:

- Int.A — a signalling, control and data interface that allows each application component to interact with each other when executing. It follows the Representational State Transfer (REST) paradigm by supporting the Web Application Messaging Protocol (WAMP) standard communication protocol.
- Int.B — an operation interface that updates the platform with the available capabilities in the Fog.
- Int.C — a management interface that monitors the status of the Fog resources. It feeds the migration engine with that information, which reallocates application components if the situation demands it.

The authors extended CloudFoundry to prototype the proposed solution and perform end-to-end data delay tests, whose results were compared with those of a Cloud-only deployment. The result analysis showed that the proposed solution had lower delays than the Cloud-based one.

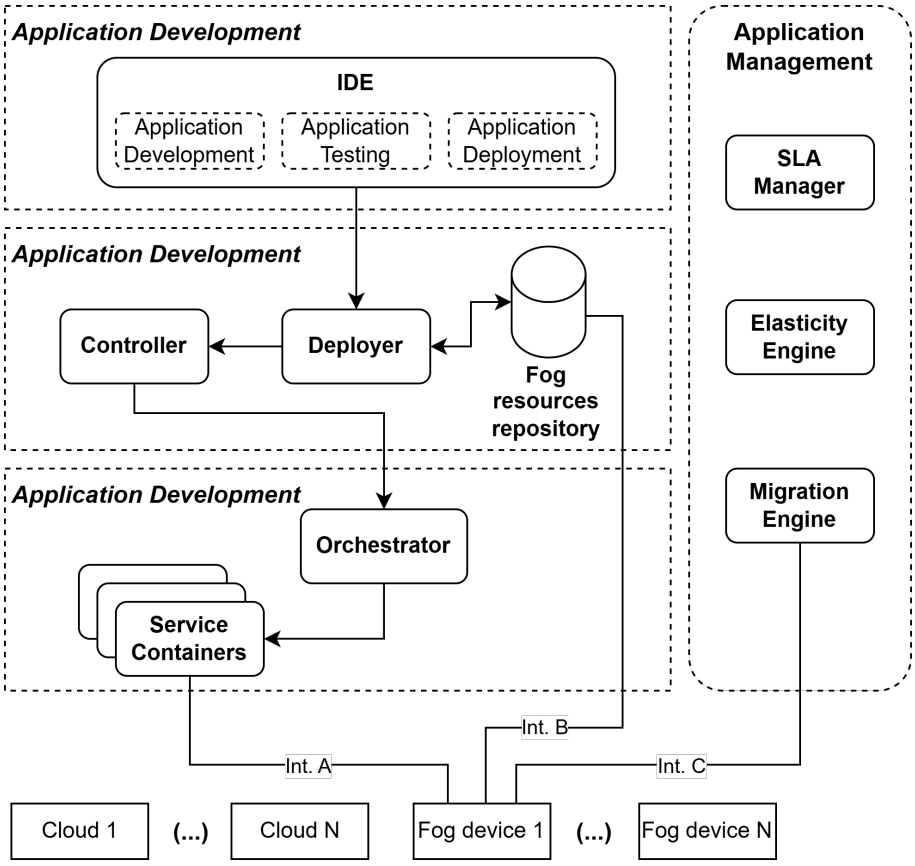


Figure 2.5: Yangui et al. proposed architecture [113].

Application development and orchestration plan specification is done using an IDE, which considers the capabilities and properties of the envisioned deployment node. The authors built such an IDE around the JSCL4Eclipse technology, so they used a GPL to create a declarative orchestration plan. Given the use of CloudFoundry, Docker is the supported runtime environment.

Pahl et al. [84] present a conceptual solution for managing clusters and containers in Edge, Fog and Cloud environments. The authors state that the architecture of such deployments would be composed of three layers: the smart things network, the lowest layer that contains the sensors and actuators; the field area network, the middle layer that contains the Internet Protocol (IP) core infrastructure; and the top layer that contains the Cloud resources. The orchestration mechanism that supports such deployments should provide localisation-aware provisioning, QoS, multi-tenancy and security features. Another crucial aspect of the orchestration solution is the creation of the orchestration plan, which should be done according to a commonly accepted standard. They propose TOSCA for this, so they follow a declarative and DSL approach. For bootstrap, the authors focus on using lightweight virtualisation, namely containers. The authors also present a practical evaluation of the proposed solution, but the results and the implementation details are scarce. The contributions of this paper are mainly theoretical. It analyses the solution requirements, the main architectural components and their functionalities, as well as the technologies available at the time of publication and their limitations.

Tsagkaropoulos et al. [106] propose a set of extensions to TOSCA that could feed the orchestration of Cloud and Fog deployments, thus their solution is based on a declarative DSL. They address the runtime reconfiguration issue by proposing the creation of two topologies: (1) the type-level, which defines the orchestration plan; and (2) the instance-level, which defines the instantiation of the system, i.e., the actual deployment of the type-level plan calculated by the orchestration algorithm. Docker is the runtime environment selected by the authors, so the solution is tied to the provisioning of nodes running the Docker engine. The authors identify the need for monitoring mechanisms, although they do not specify their implementation/integration (the authors briefly mention Prometheus as a possible monitoring tool). In fact, the authors do not focus on developing an orchestration framework. They focus on extending TOSCA for Fog and Function as a Service deployments. In their work with TOSCA, the authors theorise about an orchestration algorithm, which can optimise the deployment based on requirements, such as location (Cloud or Fog), cost, CPU, among others.

In addition to academic proposals, there are two projects from the Cloud Native Computing Foundation (CNCF)[11] - Akri[12] and KubeEdge[13] - that target the integration of IoT devices into K8S. Being K8S-based projects, they use a declarative deployment, and the Kubelet agent is the bootstrap software that needs to be running on the Cloud and Edge nodes. The integration of IoT devices is done without requiring any specific bootstrap software to run on them. However, these projects do not aim to deploy application components on IoT devices. Akri supports a set of protocols (uDev, OPC-UA, ONVIF, Modbus TCP and extensions to add support for custom protocols) and has the objective of automating the discovery of IoT devices and adding them to the K8S cluster. Once in the cluster, K8S can consider them in the function-chaining process by linking IoT device resources, such as sensors, cameras, etc., to the application components that need them. KubeEdge creates digital twins of IoT devices with the objective of managing them to guarantee that the twin and the device are synchronised with a specific desired state. This management strategy means that, in addition to allowing the function chaining of the resources of IoT devices with the application components that consume them, KubeEdge allows for manipulating their operation through parametric configuration. KubeEdge also considers the deployment of application components that interact with the IoT device in the Edge node that hosts the IoT device digital twin and bridges it with the K8S cluster.

Another relevant project, hosted by the Eclipse Foundation, is Arrowhead[14], which aims at IoT automation and digitalisation through a System of Systems approach in which the IoT elements are abstracted as services. However, it does not aim for application component deployment/placement, which is the focus of our Thesis. It assumes the services are already available and the orchestration purpose is to chain them according to the application objectives and QoS requirements. The service specification and the connection between provider and consumer are defined using a DSL. Interoperability is addressed through translators and adapters for OPC-UA, Modbus TCP and Web

[11] https://www.cncf.io/
[12] https://docs.akri.sh
[13] https://kubeedge.io/
[14] https://projects.eclipse.org/projects/iot.arrowhead

of Things. Runtime management is planned for future releases and aims to update the service chains at runtime. Security is guaranteed through authentication and authorisation mechanisms.

Another relevant proposal is The Linux Foundation project Margo[15], an industrial initiative to address Edge interoperability in industrial automation. It is still under specification, and the current version envisions an interoperable approach for application specification and application component deployment on Edge devices on factory floors. It follows a descriptive approach based on K8S DSL with security considerations. Runtime adaptation is supported by K8S reconciliation pattern, and the envisioned telemetry system. In its current version, Margo orchestration is focused on Edge devices and their compute resources, disregarding the sensing and actuation characteristics of IoT and IIoT devices.

Table 2.1 presents an overview of the studied orchestration frameworks considering the taxonomy presented in Section 2.1.1. The overview only considers the four major groups of features because many of the orchestration frameworks do not target the aspects considered inside each group or just briefly mention them.

Considering the overview, it is possible to conclude that interoperability and using standards to address it, both for the topology definition and data communication and representation, are not entirely addressed by the studied solutions. Some frameworks use TOSCA for the topology definition, but by itself, TOSCA does not address IoT deployments, so the language does not easily model IoT devices and their heterogeneity and quality aspects. Tsagkaropoulos et al. [106] propose some extensions to the standard to address Fog deployments, but such extensions do not support the requirements of Edge and IoT devices. In 2018, OASIS created a TOSCA Ad Hoc workgroup to discuss its applicability in IoT, Edge and Fog[16]. However, finding any conclusions from such a workgroup was impossible. Additionally, the current status of TOSCA and its industry adoption are unclear. A new version of TOSCA, version 2.0 [59], has been under preparation since its latest release in 2020, but such a release is not yet stable. Margo is defining an open standard, targeting application and deployment specifications for industrial Edge devices. Its current version does not support IoT devices and IoT applications quality requirements. Regarding the use of standards for communication and data representation, Brito et al. [23] briefly mention the need to use standard communication protocols but do not specify a solution, nor the standards that could be considered. Yangui et al. [113] propose the use of REST-based communications between the different modules of the orchestration framework and WAMP for the communication with external components. Relying on a single communication protocol with external components might not be feasible, since the heterogeneous IoT devices might not support or use it for communication. The orchestration framework should be able to deal with such heterogeneity. KubeEdge addresses this issue using mappers to convert the IoT device communication protocol to MQTT. However, they do not follow a standards-based strategy to avoid vendor lock-in. Arrowhead uses translators for standard protocols, such as Modbus TCP, OPC-UA and Web of Things, to allow services to

[15] https://margo.org/
[16] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca

connect with each other. However, its focus is not on application component placement, so its topology definition is not relevant for our scope.

Still, regarding interoperability, only GENESIS [31] is not dependent on the device runtime environment, while the other solutions rely on a particular virtualisation technology for application component provisioning and management. The support for heterogeneous runtime environments and non-virtualised devices is crucial, since it adds to the orchestration framework the capability to manage IoT devices that do not support virtualisation.

With respect to monitoring and runtime management aspects, an advanced feature of the IoT orchestration framework, we can conclude that it is not yet fully addressed. Yangui et al. [113] state that the proposed solution supports QoS and monitoring, but lacks their implementation details. Pahl et al. [84] identify the need for these features without proposing a solution for them. Tsagkaropoulos et al. [106] identify infrastructure and application component monitoring as fundamental features of the orchestration framework, but also do not provide a solution (the authors identify Prometheus as a possible approach). The authors who propose concrete solutions to address the monitoring and runtime management issue focus their approaches on Fog or Edge devices, and their QoS requirements, disregarding the IoT devices and their intrinsic quality characteristics. FITOR implements infrastructure and application component monitoring using Prometheus, but the solution only targets Fog nodes. Margo specification considers the use of Open Telemetry for application and Edge device monitoring, hence it also does not target IoT devices. Arrowhead includes QoS monitoring and management modules, which add QoS support to the framework. However, management is focused on service function chaining.

The surveys performed by Yousefpour et al. [114], Nguyen et al. [76] and Velasquez et al. [108] provide similar conclusions.

Yousefpour et al. [114] perform a comprehensive survey on various areas of IoT, focusing on Fog and Edge computing. The studied orchestration frameworks rely on virtualised devices, and one of the conclusions of the survey is that there is a lack of literature studies on monitoring Fog resources.

Nguyen et al. [76] perform a systematic review of deployment and orchestration frameworks. They conclude from the literature review that very few studies address the orchestration of IoT devices, and most of them are linked to a particular runtime environment or bootstrap engine, such as Docker or Node-RED.

Velasquez et al. [108] perform a state-of-the-art review of Fog orchestration for the IoT and concluded that they are still very theoretical. Because of this, there is a lack of detail regarding the implementation of the adaptability modules or how the solutions address the interoperability aspect. The studied solutions also do not support the provisioning and life cycle management of IoT devices, and most of them depend on virtualisation technologies.

Table 2.1: Overview of studied IoT orchestration frameworks

Framework	Deployment Support	Orchestration Support	Design Support	Advanced Support
GENESIS [31] (2019)	Declarative, targeting IoT devices. It currently supports Arduino and Docker runtime environments, but is theoretically capable of provisioning devices with other runtimes using ThingML and Node-RED.	The proposed DSL allows the configuration of low-level integration parameters, such as the definition of communication interfaces and IPs. No information about the communication support.	Proprietary DSL with limited capabilities but expandable.	Adaptation through the use of the Models@Run-time pattern and Node-RED.
FITOR [25] (2019)	Declarative, focused on Fog nodes built with Docker.	Not specified.	DSL based in Calvin, with extensions to support RAM, CPU, and network bandwidth and latency.	QoS is supported by monitoring network, CPU, and memory resources.
Alam et al.[2] (2018)	All the devices of the Edge to Cloud continuum, as long as they are based on Docker. Service function chaining done with SDN.	Data and control channels. No details about the supported communication models.	Not defined	No
Brito et al. [23] (2017)	Declarative, focused on Fog nodes running virtualisation/containerization technologies.	Identified the need to have a strong communication channel, but it does not specify a solution	Based on TOSCA, so a DSL without support for IoT requirements.	No

Continued on next page

Table 2.1 – *Continued from the previous page*

Framework	Deployment Support	Orchestration Support	Design Support	Advanced Support
Jiang et al. [49] (2017)	Not specified.	Not specified.	Not specified.	Not specified.
Yangui et al. [113] (2016)	Declarative. Targets Fog nodes based on Docker	Communication support tightened to WAMP, so limited to WAMP publish-subscribe and WAMP invocation of remote methods. No details about the integration support.	IDE based on the JSCL4Eclipse GPL	Listed the QoS and SLA features but does not provide implementation details
Pahl et al. [84] (2016)	Declarative, focused on Edge and Fog nodes with lightweight virtualisation capabilities	Not specified	Identifies the need for an accepted topology, and proposes the use of TOSCA DSL	Identifies the need for localisation awareness, QoS, multi-tenancy and security features, but no implementation is proposed
Tsagkaropoulos et al. [106] (2021)	Declarative. Targets Fog and Cloud nodes based on the Docker runtime environment.	Not specified	DSL. Propose extensions to TOSCA to support Fog nodes, FaaS use cases and run-time topology updates.	Localisation aware provisioning, FaaS use cases

Continued on next page

Table 2.1 – *Continued from the previous page*

Framework	Deployment Support	Orchestration Support	Design Support	Advanced Support
Akri[12]	Declarative. Application component deployment in K8S Kubelet-compliant nodes and function chaining in IoT devices.	Cloud and Edge does not impose any restriction regarding the communication between application components. Regarding integration support, it supports low-level configurations, such as port exposure, routing and network policies.	DSL based on YAML with limited support to IoT devices	K8S default advanced features, such as security, monitoring and deployment adaptation. A dynamic community of contributors proposes extra features. Automated discovery of IoT devices.
KubeEdge[13]				K8S default advanced features, such as security, monitoring and deployment adaptation. K8S dynamic community of contributors proposes extra features. Parameter configuration of IoT devices.

Continued on next page

Table 2.1 – *Continued from the previous page*

Framework	Deployment Support	Orchestration Support	Design Support	Advanced Support
Arrowhead[14]	Not Applicable	The DSL documentation appears to support multiple communication modules. Infrastructure layer setup, including service discovery, networking and resource reservation considering QoS requirements	DSL to specify services, provider endpoints and consumer requirements	Security through authentication and authorisation. Runtime adaptation planned for future releases. Interoperability through translators. QoS support.
Margo[15]	Declarative. Application component deployment in Margo-compliant Edge devices.	The current specification does not appear to impose any restriction regarding the communication between application components. Regarding integration support, by being based on K8S, it would support low-level configurations, such as port exposure, routing and network policies.	DSL based on YAML focused on the industrial Edge	Specification requires compliant components to consider security, monitoring and deployment characteristics. Interoperability regarding application specification, deployment and telemetry.

Continued on next page

Table 2.1 – *Continued from the previous page*

Framework	Deployment Support	Orchestration Support	Design Support	Advanced Support
DOT	Declarative. K8S supports Cloud, Fog and Edge. DOT adds support to IoT devices. Abstracts the runtime environment of the nodes and their bootstrap software. It needs a translator that integrates the IoT device in DOT.	Cloud and Edge do not impose any restrictions on the communication between application components. IoT devices work on publish-subscribe. In terms of integration support, it supports low-level configurations, such as port exposure, routing and network policies.	K8S DSL adapted to support IoT devices.	K8S default advanced features, such as security, monitoring and deployment adaptation. K8S dynamic community of contributors proposes extra features. IoT quality-aware provisioning and runtime adaptation. Addresses vendor lock-in through the use of standards.

2.1.3 Positioning DOT in the Scope of IoT Orchestration

There are multiple challenges in the integration of IoT devices in Cloud orchestration, such as their discoverability and heterogeneity, as well as their support for service-oriented runtimes that allow workload deployment in addition to full software and firmware updates. Additionally, the orchestration solution needs to handle the definition of IoT specific characteristics using standard topologies specification languages, considering the quality and security requirements of IoT devices and applications, along with monitoring and the creation of fault tolerance mechanisms, which manage the IoT installation through parametric adaptation and the deployment of new software models. In the scope of this Thesis, we focus on IoT device heterogeneity regarding interaction and data models, as well as runtime environments, and IoT applications quality. From our perspective, these are foundational challenges that are not yet resolved. By addressing heterogeneity, we are not bounding our solution to a specific set of IoT devices and runtime environments. Additionally, handling IoT applications quality truly enables orchestration in the scope of IoT by allowing the scheduler to reason about the intrinsic qualities of IoT applications instead of taking into consideration just the ones related to the Cloud, Fog or Edge, which existing Cloud orchestration systems already handle. Considering this, we argue that addressing these challenges allows building a complete Proof of Concept (PoC) of an IoT orchestration framework that the community could start exploring in multiple use cases, promoting the identification of new challenges and limitations, and helping to prioritise the challenges left open.

DOT is the PoC that addresses the chosen challenges. To develop it, we defined RQ1 to RQ3, whose answers define how we address these challenges. Following this, our approach for developing DOT was to add support for the orchestration of IoT devices to a mature Cloud orchestration system, because it would enable an integrated orchestration of the IoT continuum, since the community is already working on supporting Fog and Edge in these platforms. We have selected Kubernetes (K8S) for this purpose, because, as explained in Section 1.2, it is a mature, open-source and modular Cloud orchestration system that enables us to cover the IoT continuum following a non-intrusive approach. With this strategy, users can continue using K8S as they currently do, and, if they have the intent of using K8S in the scope of IoT, they just need to install the set of DOT's addons and plugins.

Considering this, we have included DOT in Table 2.1 to position it against the current orchestration solutions. DOT's ***deployment support*** features a declarative orchestration approach, based on K8S DSL, which tackles the IoT continuum (Cloud, Edge and IoT devices) and is independent of the bootstrap engine and runtime environment. K8S DSL is the tool used for DOT's ***design support***. We adapted it to support IoT devices, allowing their representation in the K8S cluster and the definition of their application components requirements and optimisation objectives. The standards-based approach followed by DOT to address heterogeneity is part of its ***advanced support***, which makes its ***orchestration support*** coupled with the publish-subscribe model on the communication with IoT devices. This communication model is inherited from the solution for interoperability discussed in Section 2.2.4 and presented in Chapter 3. DOT takes advantage of

K8S and its virtual network interfaces for the communication between application components, allowing the low-level configuration of such communication, including port exposure, routing, network policies and K8S services, which abstract application component network configurations and provide a stable network endpoint to access application components independently of network updates or application component migration between devices. DOT's ***advanced support*** is complemented by its configurable quality-aware scheduler and life cycle monitoring.

Section 6.1 presents the details of DOT and how it emerges from the answers to RQ1 through RQ3, addressing IoT continuum orchestration and validating our Thesis. Section 4 details our approach to answer RQ2, while RQ1 and RQ3 are further explored in the remainder of this State of the Art (Sections 2.2 and 2.3, respectively).

Section 6.2 further explores the challenges identified in IoT orchestration that were left open, as well as their relation with the insights obtained while working on RQ1 through RQ3. The objective is to provide our view about the relevant open challenges and possible next steps for peers working on the topic of IoT orchestration, independently of whether such work is being done with DOT or not.

2.2 Interoperability in IoT

In this section, we provide an overview of the existing standardisation efforts in the scope of IoT, which gives a high-level understanding of the dynamism of standard approaches in various IoT aspects (Section 2.2.1). We then focus our review on the standard models used to represent the characteristics of IoT devices and their sensing and actuating capabilities (Section 2.2.2), which is one of the interoperability issues addressed in this Thesis. Considering this, we then review the literature for existing interoperability proposals that could deal with the heterogeneity in IoT data models (Section 2.2.3). The section closes with an analysis of the interoperability problem from the perspective of data models heterogeneity, and how they should be considered in the scope of IoT devices orchestration (Section 2.2.4).

2.2.1 IoT standardisation efforts

Morabito and Jiménez [74] overview the Internet Engineering Task Force (IETF) standardisation efforts in the IoT domain. Some key takeaways of this review are:

- The IETF effort on bringing IP communication to non-IP communication protocols, even in scenarios composed of very resource-constrained devices. IPv6 over Low -Power Wireless Personal Area Networks (6LoWPAN)[17], IPv6 over Networks of Resource-constrained Nodes (6lo)[18] and IPv6 over Low Power Wide-Area Networks (lpwan)[19] are some efforts

[17] https://datatracker.ietf.org/wg/6lowpan/about/
[18] https://datatracker.ietf.org/wg/6lo/about/
[19] https://datatracker.ietf.org/wg/lpwan/about/

in this scope. Routing Over Low power and Lossy networks (ROLL)[20] is another relevant working group in this topic.

- IETF is also interested in supporting resource-oriented applications. Constrained RESTful Environments (CoRE)[21] and Concise Binary Object Representation (CBOR)[22] are two important working groups in this scope. CoRE aims at defining mechanisms and frameworks for resource-oriented applications, being Constrained Application Protocol (CoAP) [100] — a RESTful-based communication protocol for resource-constrained devices — one of the major takeaways of this group. CBOR enables the use of JSON-based features in resource-constrained devices. It can be used with CoAP to encode a JSON payload in binary format.

- IETF also explores topics such as resource discovery and management, mainly around the CoRE working group. Examples of such efforts are: CoRE Resource Directory [4], which addresses the issue of resource discovery; Constrained RESTful Application Language (CoRAL) [40], which is currently archived but aimed to describe the connections between resources and possible operations on them; and CoAP Management Interface (CoMI) [107], which aims to provide a network management interface for constrained devices.

- Sensor Measurement List (SenML) [47] is another effort in the IETF CoRE working group. SenML defines a format to represent simple sensor data in JSON, CBOR, Extensible Markup Language (XML) and Efficient XML Interchange (EXI).

- Thing-to-Thing Research Group (T2TRG)[23] is another relevant IETF effort. T2TRG is a research-oriented group focused on guiding the design of IoT systems that follow the REST architectural style. They address topics such as management and operation of constrained-node networks, security, life cycle management and semantic interoperability.

Sinche et al. [102] provide a survey on the IoT management platforms and frameworks focused on solutions targeting IoT devices. Two interesting contributions are the analysis of the management architectures and IoT management protocols proposed by standardisation organisations.

In what respects the management architectures, the authors overview the International Telecommunication Union - Telecommunication Standardization Sector (ITU-T)'s IoT Device Management Framework [44], OneM2M Management Architecture [81], FIWARE Platform for IoT Management[24], and Open Connectivity Foundation (OCF) Management Architecture for IoT [82] (which is implemented in the IoTivity framework[25]). They conclude that all the architectures follow a client-server approach. Some of them (ITU-T's IoT Device Management Framework and OneM2M Management Architecture) include a gateway entity responsible for bridging the management task with the devices not compliant with the defined communication procedures. The

[20] https://datatracker.ietf.org/wg/roll/about/
[21] https://datatracker.ietf.org/wg/core/about/
[22] https://datatracker.ietf.org/wg/cbor/about/
[23] https://datatracker.ietf.org/rg/t2trg/about/
[24] https://www.fiware.org/
[25] https://iotivity.org/

proposed architectures are a high-level definition of an IoT management system, hence they do not directly address the scalability, interoperability, security, and reliability issues. Such issues are partially addressed by the standard IoT management protocols that can be used to build the proposed architectures.

Still on this topic, the IETF's Request for Comments (RFC) 7547 (Management of Networks with Constrained Devices: Problem Statement and Requirements) [27] is also analysed due to its focus on the management problem of networks of resource-constrained devices. This RFC presents a problem statement and a list of requirements for several use cases around this topic.

Regarding IoT management protocols, the authors overview: the International Organization for Standardization (ISO) Common Management Information Model (CMIP) and Common Management Information Model Over TCP/IP (CMOT) protocols [109]; the IETF Simple Network Management Protocol (SNMP) [30], NETwork CONFiguration Protocol (NETCONF) [26], Representational State Transfer Configuration Protocol (RESTCONF) [10] and CoMI [107]; and the Open Mobile Alliance (OMA) Device Management (DM) protocol [68] and Lightweight Machine to Machine (LwM2M) protocol [69]. The conclusion was that only LwM2M and CoMI are targeted for IoT devices. CoMI is still an IETF draft, while LwM2M was already released and is supported by some open source implementations. Additionally, LwM2M allows managing IoT devices and exchanging actuation commands and sensing data. Moreover, practically all the protocols use Yet Another Next Generation (YANG) as the specification language and already include security features. The IoT management standards support several transport protocols, namely: Hypertext Transfer Protocol (HTTP), Transmission Control Protocol (TCP), Transport Layer Security (TLS) or CoAP, among others.

Our overview of IoT standardisation efforts highlights the high number of different approaches to this topic. Full-stack standardisation efforts that take a holistic view and define the models for data and device representation, and the protocols for communication and management use existing communication, representation and management standards in their definition. LwM2M, IoTivity, and OneM2M fall into this category and bundle existing standards according to the objectives of their target applications, which can be linked to the type of device (resource-constrained or not) and/or the application domain (industrial automation, smart homes, etc.). LwM2M and IoTivity, for example, use several outputs from the IETF CORE working group because they aim to support resource-constrained devices.

Management in the scope of the reviewed work relates to device and network configuration, not to orchestration and application component deployment. Considering this, and although the integration of such management standards might be beneficial for DOT (is one of the items identified for future work in Chapter 6.2.1), our focus in the scope of this Thesis is on device and data representation, which are foundational interoperability aspects that need to be addressed for IoT orchestration and application component deployment. The orchestration framework needs to know what a device is to consider it during scheduling, i.e., when it is deciding, considering the characteristics of the devices, which device to bind with an application component. Section 2.2.2 focuses on the modelling standards for IoT.

2.2.2 IoT Modelling Standards

This section overviews the standardisation effort for modelling data and information in a way that could enhance human and machine reasoning. Before briefly overviewing the standardisation efforts, we would like to clarify three key concepts on standard data/information modelling:

- Syntax: Refers to the structure and rules in which the information is represented. It defines formatting, allowed characters, notations and structure (e.g. XML, JSON, YANG).
- Semantic: Defines the meaning of the elements and structure defined by the syntax, allowing the interpretation of the data and information coded by the formal syntax definition.
- Ontology: Provides a vocabulary and structure to represent knowledge. It defines concepts (classes of elements) and their relationships. For example, it can define the vocabulary and structure to describe the concepts of sensor, measurement, and measurement procedure, as well as the relationship between them: a sensor uses a measurement procedure to acquire a measurement.

In the remainder of this section, we provide an overview of the standardisation efforts for defining ontologies and data models with semantic concerns.

Ana Silva and Pascal Hirmer [35] survey existing models for IoT. The key models identified in the survey are:

- Semantic Sensor Network Ontology (SSN) [5], which is a World Wide Web Consortium (W3C) standard ontology for describing: sensors, including their observed properties and observation procedures, and observation results; actuators, including the actuation procedure, the actuated properties and actuation results; samples, including the sampling procedure, the sampled features and results. It builds on top of W3C Sensor, Observation, Sample, and Actuator (SOSA) ontology, which defines the base entities, relations, and activities associated with sensing, sampling, and actuation.
- oneM2M base [80] is the ontology defined by the oneM2M standard association and the one used in its specification. It allows the definition of devices, their network aspects, components (such as sensors, actuators, etc.) and functions.
- IoT-Lite [9] is a lightweight ontology to represent the IoT devices and its services. It is an instantiation of the SSN ontology, extending it with information about the class of IoT device, which can be a sensor, actuator or tag (currently only sensor is defined), and the coverage of its services in a two-dimensional space. IoT-Lite was submitted in 2015 to W3C for approval and the creation of a working group to continue working on it (it was developed by the FIWARE[26] and FIESTA-IoT[27] European projects, not by W3C). However, no working group seems to have been created until now.

[26] https://www.fiware.org/
[27] https://cordis.europa.eu/project/id/643943

- Open Platform Communications - Unified Architecture (OPC-UA) Information Model [37] is the data representation used by the OPC-UA specification. It allows the specification of industrial assets, which can be aggregated hierarchically to compose a system. It also models the asset measurement results and the operations it provides.
- SenML [47] is an IETF standard, which was already addressed in Section 2.2.1.
- Sensor Model Language (SensorML) [14] is a standard from the Open Geospatial Consortium (OGC) that defines measurements and the transformation of observations, by defining their processes and components.

Michael Jacoby and Thomas Usländer [45] focus their study on Industry 4.0, and the digital twin and IoT standards that can support it. The key objects of their analysis are the following standards:

- Asset Administration Shell (AAS)[28] from Platform Industrie 4.0. It is used to digitally represent an asset, which can be organised hierarchically to represent a simple component, a machine or a plant. The standard is composed of five parts, of which part one focuses on the model and how to serialise the content of an AAS using XML, JSON, Resource Description Framework (RDF), OPC-UA information model, and Automation Markup Language (AutomationML), which models engineering information, such as plant hierarchy structure and components.
- Next Generation Service Interfaces-Linked Data (NGSI-LD) [28] is an ETSI standard composed of an API and data model. The data model represents context information by using the concept of entities that have properties and relationships with each other.
- The SensorThings API (STA) [66, 65] is an OGC standard made to represent IoT sensing devices, the observed properties, and the observation procedures. It also defines the API to interact with its data model.
- The Web of Things (WoT) Thing Description (TD) [50] is a W3C standard that allows the description of IoT resources as Things, which comprise properties, actions, and events. To improve interoperability, WoT TD does not define an API to access the resource. It specifies instead a meta-model to describe the supported APIs (protocol, payload, security, etc.).

In addition to the standards identified in these studies, we also highlight: (1) Smart Appliances REFerence ontology (SAREF) developed by ETSI. It comprises a core ontology, which models devices, their sensors and actuators, and corresponding observations and actuations. It also models the tasks and functions of devices, as well as the commodities they produce, consume, and/or store. The core ontology is then extended for multiple domains, such as energy, environment, building, smart cities, industry and manufacturing, smart agriculture and food chain, automotive, eHealth, wearables, water, smart lifts and smart grid; (2) LwM2M data model developed by OMA

[28] https://industrialdigitaltwin.io/aas-specifications/index/home/index.html

is focused on modelling IoT devices through its objects, and the resources that compose such objects, as well as the operations that can be performed on them; (3) IoTivity data model developed by OCF follows an approach similar to LwM2M, it specifies IoT devices, its resources (can be mapped to LwM2M objects), the proprieties that compose such resources, and their operations. It focuses on providing concrete device models for smart building automation, smart home and healthcare; (4) ISO/Draft International Standard 23726-3 is intended to represent industrial data and information. It follows an ontology-based perspective, aiming to provide rich information and context for all phases of the life cycle of industrial assets and processes; (5) IETF Semantic Definition Format (SDF) models IoT devices, along with their objects and interactions, including the data types of the information shared in those interactions [56]. It is the definition format adopted by OneDM[29], which is a community of industry groups that aims to develop and curate single, agreed-upon data models to represent the different classes of IoT devices and their objects.

Table 2.2 summarises the ontologies (W3C SSN and SOSA, oneM2M base, IoT-Lite, ETSI SAREF, ISO/DIS 23726-3) and data/information models (OPC-UA, IETF SenML, OGC SensorML, AAS, AutomationML, NGSI-LD, OGC STA, OMA LwM2M, OCF IoTivity, IETF SDF and W3C WoT TD) identified in this section. We use it to ease our analysis of the existing modelling standards.

[29] https://onedm.org/

Table 2.2: (I)IoT modelling standards

Standard	Type	Modelling Objective	Domain or Application
SSN and SOSA	Ontology	Sensors, observation procedures, observed properties, and results. Actuator, actuation procedure, actuated properties and results. Sample, sample procedure, sampled properties and result. The device deployment context.	No specific domain or application.
oneM2M base	Ontology	Device and its components (sensors and actuators)	No specific domain or application.
IoT-Lite	Ontology (instantiation of SSN)	Class of IoT device, which can be sensor, actuator or tag (currently only sensor is defined), and the coverage of its services in a two-dimensional space	Devices types
SAREF	Ontology	Devices, their sensors and actuators, and corresponding observations and actuations. The tasks of devices and their functions, and the commodities they produce, consume and/or store.	Energy, environment, building, smart cities, industry and manufacturing, smart agriculture and food chain, automotive, eHealth, wearables, water, smart lifts and smart grid.
ISO/DIS 23726-3	Ontology	Industrial data and information, modelling all phases of the life cycle of industrial assets and processes	Industry
OPC-UA	Data model	Industrial assets. Hierarchical aggregation to compose systems. Asset operations and measurement data.	Industry
AAS	Data model	Industrial asset. Hierarchical organisation allows representing simple components up to machines or a plant.	Industry
AutomationML	Data model	Engineering information, such as plant hierarchy structure and components.	Industry

Continued on next page

Table 2.2 – *Continued from the previous page*

Standard	Type	Modelling Objective	Domain or Application
NGSI-LD	Data model	Context information.	Smart cities, smart industries, and digital twins.
SenML	Data model	Represent sensor measurements	No specific domain or application.
SensorML	Data model	Measurements and the transformation of observations.	No specific domain or application.
STA	Data model	Sensing devices, the observed properties, and the observation procedures	No specific domain or application.
LwM2M	Data model	Devices through their objects, the resources that compose the objects, as well as the operations they provide.	Constrained devices
IoTivity	Data model	Devices, their resources, the properties that compose the resources, as well as their operations	Smart building automation, smart home and healthcare
SDF	Data model	Devices and their objects, as well as the interactions provided by the objects and devices. The data types of the information shared in the interactions.	No specific domain or application.
WoT TD	Data model	Resources, which comprise properties, actions, and events. Specifies a meta-model to describe the APIs to access such resources.	No specific domain or application.

Regarding ontologies, they are focused on modelling concrete entities, i.e, they are not upper ontologies, such as Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE) [11], DOLCE Ultra Lite [11], Unified Foundational Ontology (UFO)[30] or Suggested Upper Merged Ontology (SUMO)[31], which are foundational ontologies that define general concepts on top of which other ontologies can be defined, providing an interoperability foundation between those. Except W3C SSN and SOSA, which were initially created with alignment with DOLCE Ultra Lite, none of the other ontologies used an upper ontology in their definition. Even W3C SSN and SOSA, with their evolution, dropped the direct alignment with DOLCE Ultra Lite, making it optional. Despite this, there is an interoperability effort through the proposal of mappings between different ontologies, such as OneM2M and SAREF [29], OGC and W3C[32], and W3C SSN/SOSA and SAREF [46].

In what respects data models, there are several standards focused on the industry domain, modelling different aspects of the industrial processes: OPC-UA; AAS; AutomationML. Additionally, several standardisation organisations propose models related to sensors, sensor measurements and observation procedures: IETF SenML, OGC SensorML and OGC STA. The modelling of devices, their properties and available interactions is proposed by: LwM2M, which focuses on constrained devices; IoTivity that focuses on smart building automation, smart home and healthcare; and SDF and WoT TD, which target interoperability and domain independence.

Considering this heterogeneity in data modelling, the envisioned IoT orchestration framework should be able to deal with it. One approach to address this interoperability issue is to have an intermediary entity, such as a gateway or middleware, that bridges the IoT orchestration framework with the IoT data model heterogeneity. This gateway or middleware should be standards-based to avoid vendor lock-in. Ideally, it should use a domain-independent standard, such as SDF or WoT TD, to avoid restricting its application to the application domain of the standard. Following this, we explore in Section 2.2.3 the literature on IoT middlewares and gateways addressing the issue of heterogeneous data model interoperability.

2.2.3 IoT Middleware/Gateways addressing IoT Interoperability

IoT middlewares and IoT gateways are the two main solutions proposed in the literature to address IoT interoperability. Their differences are normally associated with their role in the IoT stack [7]. The IoT middleware is normally associated with the Cloud and the data processing and management tasks traditionally performed at this layer. The IoT gateway is normally associated with the Edge and the traditional roles of protocol translation and management of data flows. However, with the advent of Edge computing, some tasks performed in the Cloud can be offloaded to the gateway at the Edge. With this, the separation between IoT gateway and IoT middleware becomes more tenuous, with both starting to share some functional properties, such as data management and resource discovery, management and provisioning [8, 20].

[30] https://ontouml.org/
[31] http://www.ontologyportal.org/
[32] https://www.w3.org/groups/wg/sdw/

Independently of where interoperability is addressed in the IoT stack, syntactic and semantic interoperability are needed. In this section, we provide an overview of the literature proposals for IoT gateways and middlewares that address such interoperability issues.

Aloi et al. [3] propose a smartphone-based gateway adaptable to different communication protocols and data models. These authors mention the support for five protocols (ZigBee, SPINE, Wi-Fi, Bluetooth and 3G/LTE) and state that they designed their software architecture to add support to others seamlessly. However, there is no information on how to do it, nor about the API and data model used to interface the IoT applications with the devices.

Diana Yacchirema, Manuel Esteve and Carlos Palau [112] propose a gateway to solve the IoT interoperability in pervasive environments through standards. However, the proposed solution is only compatible with IoT devices that use MQTT or CoAP as communication protocols, which limits its application scope. Despite translating the IoT devices data model to a common data model, theoretically a standard one, the authors do not specify which standard was used. They just specify that the syntax is based on JSON. There is also no information on the API that the gateway exposes to communicate with the IoT devices.

The industrial gateway proposed by Viacheslav Kulik and Ruslan Kirichek [58] translates data from CoAP, MQTT, ModBus Remote Terminal Unit, ModBus TCP/IP, and Controller Area Network (CAN) packets into a common and proprietary packet format. Since the translation scope is limited to the packet structure, the semantic information of the packet payload remains heterogeneous, so it does not fully address the syntactic and semantic interoperability issue.

MSOAH-IoT [71] is a REST-based middleware capable of interfacing with ZigBee, IPv6 over Low -Power Wireless Personal Area Networks (6LoWPAN), Bluetooth Low Energy (BLE), and Wi-Fi devices, and extensible to other communication protocols. It translates each device-specific service and functionality to the proposed REST API. However, there are no details about the translation procedure, so there is no information on how to add new communication protocols. There is also no information on how the proposed solution manages the different data models used by the IoT devices.

The gateway proposed by Nugur et al. [77] is adaptable to different IoT devices communication protocols and payloads. Although modular, the proposed solution does not follow a standards-based approach, and its API based on web sockets seems to be solely focused on the smart building application domain.

In.IoT [21] is a microservice-based IoT middleware that, among other goals, addresses interoperability, but it does not mention the use of any standards-based approach to do so. XWARE [95] follows the same non-standard approach, proposing a middleware for pervasive environments that addresses, among other aspects, communication and data heterogeneity.

In fact, the use of non-standard approaches is common to all the solutions analysed above and is in line with Beniwal and Singhrova's conclusions in their survey on IoT gateways [8]. We only found three solutions in the literature that follow a standards-based approach.

Pratikkumar Desai, Amit Sheth and Pramod Anantharam [24] propose a gateway solution that interacts with IoT devices through CoAP, MQTT and Extensible Messaging and Presence Protocol

(XMPP). It then adds semantic annotation to their data using OGC Observation and Measurement, OGC SensorML, OGC Sensor Observation Service and W3C SSN ontology. The downside of this approach is the lack of information on how to add support for more protocols, and the focus of the used annotation standards on the monitoring and actuation process, and sensor and actuator characterisation, disregarding the characterisation of the device itself [36].

The IoT middleware proposed by Cheng et al. [111] is based on knowledge graphs and uses the OneM2M ontologies to define its elements and SPARQL [38] as the protocol and query language to access its data. A downside of this approach is that subscription to changes is not directly available in SPARQL. Subscribing to changes in the properties and events of IoT devices is an important feature for IoT systems due to their intrinsic dynamism, which cannot rely on query-based approaches that might bring more complexity and overhead.

Yonghua Li, Xuxin Huang and Siye Wang [64] propose an OCF-based gateway that can interface with non-OCF devices, and exposes them as OCF elements, so it performs protocol and data model translation. This is a very interesting approach, whose main downside is relying just on the OCF semantic representation of data that only covers the smart building automation, smart home and healthcare domains [34].

The FIWARE open source framework[33] is also an interesting community-based interoperability proposal. The framework development was supported by a set of European projects, and currently, a community of research and industrial partners contribute to it. It follows a standards-based approach by adopting the ETSI NGSI-LD data model and API. Its architecture is highly scalable due to the use of a broker that implements the ETSI NGSI-LD API and bridges third-party devices and applications to ETSI NGSI-LD data. FIWARE provides a catalogue of different connectors (enablers in FIWARE nomenclature) developed by the community that bridge the FIWARE broker with other standard protocols and data models. Despite FIWARE's interesting concept and strong community of contributors, the focus of ETSI NGSI-LD API and data model is managing data with context information, rather than the device itself, so its ability to represent the device and its interactions is limited.

Despite following a standards-based approach, the analysed solutions fail to address some important characteristics of IoT systems, such as the capability to subscribe to changes and events on IoT devices [111], as well as the capability to represent the IoT device characteristics [24] and the focus on a subset of application domains [64].

2.2.4 Positioning Interaction and Data Model Interoperability in the Scope of IoT Orchestration

There are multiple standardisation efforts in the scope of IoT, which cover topics such as device communication, network and device management, as well as data and information modelling. Each standardisation organisation normally targets a specific application domain or use case, which makes the different efforts heterogeneous and, normally, complementary.

[33] https://www.fiware.org/about-us/

Focusing on data and information modelling, there are standards focused on the syntactic and semantic aspects, as well as ontologies. In the scope of IoT orchestration, both ontologies and data models with synthetic and semantic richness are relevant. The latter can be used to build orchestration frameworks that reason over device properties, configurations and status, i.e., the orchestration framework could use standards that model devices, ideally domain-independent ones such as SDF and WoT TD. The former can be built on top of that to provide awareness about classes of devices, their procedures and relationships, as well as the specific processes in which they are involved. Considering this, from our perspective, IoT orchestration frameworks should start by targeting the syntactic and semantic data models, handling their heterogeneity. This is the approach we follow with DOT to validate our Thesis. An ontology-aware orchestration is a second step to be addressed as future work, which could be built on top of the outputs of this thesis by using its mechanisms and fine-tuning the scheduler, allowing it to reason about the domain knowledge and processes provided by the ontologies (see Section 6.2.1).

Considering this, the state-of-the-art on IoT gateways and middlewares targeting data model syntactic and semantic interoperability shows that despite the several proposals available in the literature, none completely solves it. The non-standards-based proposals do not address syntactic and semantic interoperability because they create a new proprietary abstraction which might bring vendor lock-in issues, restrict them to specific application domains, or lack support for important IoT functionalities, such as the representation of actuators and subscription to changes, among others. On the other hand, the standards-based proposals avoid vendor lock-in issues but still face the problem of targeting specific application domains and not complying with important IoT functionalities. Such non-compliance comes from the chosen standard and its data and interaction model. FIWARE is an interesting open-source interoperability proposal with an active community of contributors, whose concept aligns with this Thesis's standards-based and domain-independence aspects, but whose focus on interoperability does not meet our needs. FIWARE uses the ETSI NGSI-LD API and data model, which is focused on handling data with context information and has limited capability of representing the devices and their interactions. In this regard, WoT TD and IETF SDF are two relevant standards, which target domain independence and compatibility with the other modelling standards. For example, the ETSI NGSI-LD API and data model are not capable of modelling the devices' actions and events (interpreted according to the IETF SDF and WoT TD meanings), which are important aspects for the management of IoT devices. ETSI NGSI-LD modelling is limited to devices' properties, which do not provide information about which interactions with the device result in a physical effect (action), not just state changes, or which device changes should be kept until consumed (events), i.e., device changes whose history and order should be preserved. Considering this, the community is missing a FIWARE-like framework focused on the interoperable representation of devices and their interactions, which uses IETF SDF or WoT TD as its representation model.

WoT TD exposes each device, its properties and interactions in a structured way. When used by the framework, it would allow the orchestration framework to discover what the available devices are and what protocols, data and interaction models they use. SDF focuses on creating a

common semantic foundation for data models, which enables a consistent interpretation of data and their interactions across ecosystems, i.e, SDF presents itself as complementary to the other modelling standards, accommodating all their particularities in its definition format. This means that it is possible to build SDF translators to WoT TD [93]. When used by the framework, SDF would allow the orchestration system to use a single definition format, which would be translated by the framework to the definition format of the devices. If paired with OneDM efforts, the framework would expose each IoT device and its objects using its respective standardised data models. The SDF based approach is closer to the FIWARE approach to interoperability, which also standardises data models for context information in the Smart Data Models repository[34].

In Section 3, we propose NextGenGW, a standards-based software framework that addresses the data model syntactic and semantic interoperability problem, aiming for domain independence. To the best of our knowledge, NextGenGW is the first proposal to use IETF SDF for homogenising devices data and interaction models. We have selected IETF SDF over WoT TD because of its homogenisation in a single semantic definition format, and the data model standardisation effort for IoT devices and objects provided by OneDM. This approach enables clients at the ecosystem level (such as the orchestration framework) to use a single API for communication and data interpretation. With WoT TD, the clients would need to communicate and interpret data according to each device specification, which is exposed following WoT TD definition.

2.3 Quality in IoT Applications

For the orchestration of IoT applications to be effective, the orchestration framework needs to consider its quality requirements and optimisation objectives. Provisioning IoT applications disregarding such quality characteristics might result in inefficient or inadequate deployments. Considering this, the current section reviews the community efforts to define quality for IoT applications (Section 2.3.1) and its proposals for quality-aware IoT application provisioning (Section 2.3.2). The section closes with an analysis of the results of such reviews, discussing how, from our perspective, IoT applications quality should be considered in the scope of IoT orchestration (Section 2.3.3). For the sake of clarity, in this thesis, we use *quality characteristic* to refer to the high-level organisation of qualities according to a certain context or rationale, and *quality metrics*, or just *metrics*, to refer to the measurable parameters that define qualities and feed the quality models.

2.3.1 IoT Quality Characteristics

The community has been intensively discussing the quality metrics that affect IoT applications because traditional QoS and QoE approaches are insufficient in this scope. IoT applications might not have a human as a consumer, might run on heterogeneous devices and be composed of multiple services featuring sensing, computing, communication, and actuation aspects, each with its quality

[34] https://www.fiware.org/smart-data-models/

metrics [32, 62, 99]. Table 2.3 summarises the different organisational perspectives for defining IoT application quality identified in the state-of-the-art, which we discuss in this section.

Jay Kiruthika and Souheil Khaddaj [54] propose a hierarchical quality model to consider when selecting IoT-based services. The hierarchical organisation comprises five top-quality characteristics: reliability, interoperability, performance, scalability and security. Each top-level characteristic has its sub-characteristics: reliability is composed of fault tolerance, availability and recoverability; interoperability is composed of portability and adaptability; performance is composed of time behaviour and resource utilisation; scalability is composed of management and communication; and security is composed of privacy, consistency and stability. In some of these sub-characteristics, the authors identify another hierarchical level where they provide concrete metrics: up-time to measure availability; coupling factor to measure portability; response time to measure time behaviour; CPU usage to measure resource utilisation; number of resources to measure management; and data exchange to measure communication.

G. White et al. [110] also propose a hierarchical organisation of the quality characteristics to measure the performance of IoT applications. However, they use the International Organization for Standardization/IEC 25010 [43] quality model to identify them. The identified quality characteristics are: (1) functional suitability, with sub-characteristics completeness, correctness and appropriateness; (2) Performance efficiency, with sub-characteristics time behaviour, resource utilisation and capacity; (3) Compatibility, with sub-characteristics co-existence and interoperability; (4) Usability, with sub-characteristics accessibility and user error protection; (5) Reliability, with sub-characteristics availability, fault tolerance and recoverability; (6) Maintainability, with sub-characteristics modifiability and modularity; (6) Security, with sub-characteristics encryption and signing communications. The authors do not identify concrete metrics to measure the identified characteristics.

Despite identifying the need to examine IoT applications from a broader perspective to assess their quality fully, the proposed hierarchical models are still too abstract, lacking the relation of each quality feature with the various components of the IoT application (e.g., sensing, communication, actuation, and (pre-)processing) and the identification of concrete metrics that compose the quality characteristics.

In this sense, Li et al. [62] and Badawy et al. [6] link quality metrics to a three-layer IoT architecture comprising sensing, networking and application. Li et al. considered accuracy, energy consumption, lifetime, and cost as metrics for the sensing layer; bandwidth, capacity, power efficiency and throughput for the networking layer; and service performance cost, performance time and reliability for the application layer. Badawy et al. consider lifetime, power consumption, resource costs, and redundancy for the sensing layer; communication costs, traffic costs, communication overhead, latency and delay time, network bandwidth and throughput for the network layer; and reliability, scalability, availability, end-to-end latency, throughput and performance cost for the application layer.

Manisha Singh and Gaurav Baranwal [103] follow a similar approach, but their architecture comprises Compute, Network, and Things. However, Compute maps to Application and Things

to Sensing of the Li et al. and Badawy et al. architecture. Considering this, the identified quality metrics for the network layer are jitter, bandwidth, throughput, efficiency, network connection time, monetary cost, availability, security and privacy, interoperability, service level agreement, monitoring and reliability. At the Things (or sensing) layer, they identify device physical weight, interoperability, flexibility, availability, reliability, overall accuracy, long-term stability, response time, range, sensitivity, precision, security, over-the-air update, power consumption, drift and mobility support. At the Computing (or application) layer, the authors identify scalability, dynamic availability, reliability, pricing, response time, capacity, security and privacy, customer support facility, user feedback and reviews.

Despite being interesting and more concrete, the proposed IoT stacks do not reflect modern IoT applications in which the Things (or the sensing layer) might be (pre-)processing data. This means we should consider part of the responsibilities and quality metrics of the application layer at the sensing/Things layer. Considering this, the proposed organisation does not capture the complete set of quality metrics present at the Sensing/Things layer.

Addressing this issue, a set of proposals in the literature identifies quality characteristics according to the different entities that compose an IoT application, such as Device, Data, Information (the result of data processing), Communication, etc. In such proposals, the authors associate each quality characteristic with a set of metrics that define it.

Qamar et al. [86] focus on Things and their sensing, actuating and communicating capabilities by trying to capture their non-functional properties. They propose a definition of Quality of Things (QoT) composed of frequency, accuracy and resource consumption for the sensing capability; actuation validity and resource consumption for the actuation capability; and reception rate, delivery rate and resource usage for the communication capability.

Mocnej et al. [72] consider a wider set of characteristics to assess the quality of decentralised IoT applications. They identify five quality characteristics: (1) Quality of Device (QoDe), which relates to the capabilities and proprieties of the devices in the IoT platform; (2) Quality of Service (QoS), which is associated with network performance; (3) Quality of Information (QoI) and Context, which relates with the quality of the information generated from the processed data and the context that comes with it; (4) Value of Information (VoI) that relates with the utility of the information for a specific purpose or use case; and (5) Quality of Output (QoO), which is similar to QoI, but it looks at the produced information at a higher aggregation level.

Fizza et al. [32] focus on identifying the quality characteristics of autonomic IoT applications and reach a set of seven characteristics, some of which overlap with the ones identified by Mocnej et al. For some of the proposed characteristics the authors identify a set of quality metrics that compose them: (1) Quality of Data (QoD), which relates to the data produced by sensors and comprises the metrics data uncertainty (accuracy, timeliness and completeness) and suitability (how useful the data is for an IoT application); (2) QoDe, which refers to the quality of the IoT sensors and actuators, considering metrics such as energy consumption, mobility and environmental factors (harsh environments will impact device quality); (3) QoS, which refers to the network capacity to ensure reliable data communication between IoT devices, considering the metrics path length,

delay, packet loss and bandwidth; (4) QoI that relates to the quality of the information produced after applying data analytics to the sensor data, and contains the metrics reliability (relevance, timeliness and usefulness) and suitability (completeness and accuracy); (5) Quality of Context (QoC), which refers to parameters that characterise the context relevant to the IoT data; (6) Quality of Actuation (QoA), which relates to the correctness of the actuation decision; (7) Quality of Security and Privacy (QoSe&P), which measures the security and privacy of IoT applications.

The entity-based quality characteristics proposed by the community seem broad enough to cover all the quality aspects of IoT applications, despite the lack of consensus between authors in the set of considered quality characteristics and metrics.

With these broader views on the quality of IoT applications, it is also relevant to know how IoT application provisioning solutions handle them. Section 2.3.2 is dedicated to this aspect.

Table 2.3: Community perspectives on IoT quality characteristics.

Organization	Publications	Quality Characteristics	Quality Sub-Characteristics	Quality Metrics
Hierarchical	Jay Kiruthika and Souheil Khaddaj [54]	Reliability	fault tolerance	N.A.
			availability	Up-time
			recoverability	N.A.
		Interoperability	portability	Coupling factor
			adaptability	N.A.
		Performance	Time behaviour	response time
			Resource utilisation	CPU usage
		Scalability	Management	Number of resources
			Communication	Data exchange
		Security	Privacy	N.A.
			Consistency	
			Stability	
	G. White et al. [110]	Functional Suitability	Completeness	N.A.
			Correctness	
			Appropriateness	
		Performance Efficiency	Time behaviour	
			Resource utilisation	
			Capacity	
		Compatibility	Co-existence	
			Interoperability	
		Usability	Accessibility	
			User error protection	

Continued on next page

Table 2.3 – *Continued from the previous page*

Organization	Publications	Quality Characteristics	Quality Sub-Characteristics	Quality Metrics
Hierarchical	G. White et al. [110]	Reliability	Availability	N.A.
			Fault tolerance	
			Recoverability	
		Maintainability	Modifiability	
			Modularity	
		Security	Encryption	
			Signing communications	
Architecture Layers	Li et al. [62]	Sensing	N.A.	Accuracy Energy consumption Lifetime Cost
		Networking		Bandwidth Capacity Power efficiency Throughput
		Application		Service performance cost Performance time Reliability
	Badawy et al. [6]	Sensing	N.A.	Lifetime Power consumption Resource costs Redundancy

Continued on next page

Table 2.3 – *Continued from the previous page*

Organization	Publications	Quality Characteristics	Quality Sub-Characteristics	Quality Metrics
Architecture Layers	Badawy et al. [6]	Networking	N.A.	Communication costs Traffic costs Communication overhead Latency Delay time Network bandwidth Throughput
		Application		Reliability Scalability Availability End-to-end latency Throughput Performance cost
	Manisha Singh and Gaurav Baranwal [103]	Things	N.A.	Weight Interoperability Flexibility Availability Reliability Overall accuracy Long-term stability Response time Range Sensitivity

Continued on next page

Table 2.3 – *Continued from the previous page*

Organization	Publications	Quality Characteristics	Quality Sub-Characteristics	Quality Metrics
Architecture Layers	Manisha Singh and Gaurav Baranwal [103]	Things	N.A.	Precision Security Over-the-air update Power consumption Drift Mobility support
		Network		Jitter Bandwidth Throughput Efficiency Network connection time Monetary cost Availability Security Privacy Interoperability Service level agreement Monitoring Reliability
		Compute		Scalability Dynamic availability Reliability Pricing

Continued on next page

Table 2.3 – *Continued from the previous page*

Organization	Publications	Quality Characteristics	Quality Sub-Characteristics	Quality Metrics
Architecture Layers	Manisha Singh and Gaurav Baranwal [103]	Compute	N.A.	Response time Capacity Security Privacy Customer support facility User feedback Reviews
Application Entities	Qamar et al. [86]	QoT	Sensing	Frequency
				Accuracy
				Resource consumption
			Actuation	Actuation validity
				Resource consumption
			Communication	Reception rate
				Delivery rate
				Resource usage
	Mocnej et al. [72]	QoDe	N.A.	N.A.
		QoS		
		QoI		
		VoI		
		QoO		

Continued on next page

Table 2.3 – *Continued from the previous page*

Organization	Publications	Quality Characteristics	Quality Sub-Characteristics	Quality Metrics
Application Entities	Fizza et al. [32]	QoD	N.A.	Accuracy
				Timeliness
				Completeness
				Suitability
		QoDe		Energy consumption
				Mobility
				Environmental factors
		QoS		Path length
				Delay
				Packet loss
				Bandwidth
		QoI		Relevance
				Timeliness
				Usefulness
				Completeness
				Accuracy
		QoC		N.A.
		QoA		
		QoSe&P		

2.3.2 Quality-aware IoT Application Provisioning

This section provides an overview of community proposals for provisioning solutions that consider IoT quality characteristics and metrics.

Mocnej et al. [72] propose a greedy algorithm for service selection with energy efficiency optimisation, which considers QoS, QoDe, QoI, QoO, and VoI. The solution focuses on the particular use case where a gateway connects to the IoT network, whose devices offer the services to select from. The proposed solution is tailored to this set of quality characteristics and to an infrastructure setup composed of an IoT network bridged with the cloud by a gateway. The authors do not provide any mechanism to adapt the proposed algorithm to other quality characteristics and models.

Badawy et al. [6] propose a QoS service selection framework that maximises the composite service quality of IoT applications by balancing service reliability and computational time. The framework considers the quality metrics availability, load of services, and response time. It is not possible to adapt the framework to consider other balancing objectives for service selection, nor to use different quality metrics for the supported balancing objective.

Li et al. [62] scheduling model targets the selection of services for composite IoT applications. The proposed scheduler is tied to the metrics sensor precision, energy consumption, lifetime and cost.

Brogi et al. [15] focus on deploying Fog applications and propose a cost model and an application component placement multi-objective optimisation strategy. The authors tailored the solution to the metrics QoS, CPU and memory requests, and monthly costs. They do not explore the possibility of adapting the proposed strategy to consider other quality metrics, nor even adapting the optimisation strategy and goals with the already supported metrics.

Yousefpour et al. [114] propose FogPlan, a QoS-aware framework that optimises component placement. FogPlan placement decision is tied to the analysis of resource use, bandwidth, and service delay.

Santos et al. [97] propose Diktyo, a K8S QoS provisioning solution for Fog deployments that considers Round Trip Time and bandwidth as quality metrics. They propose an extension to the K8S default scheduler that improves its filtering and scoring capabilities to consider such metrics.

Considering the analysed quality-aware application provisioning proposals, we can divide them into service/application component selection [72][6][62] and application component placement [15][114][97]. Independently of the main purpose, selection or placement, they are fine-tuned to a particular use case, i.e., they only support a small set of specific quality characteristics and metrics, and are tied to a single scheduling strategy and quality model. Adapting the proposed schedulers to other applications, or even the same application but considering other scheduling objectives and quality metrics, is not possible.

Focusing on the particular scope of K8S, the orchestration framework we considered for our Thesis, there are some recent surveys dedicated to the community-proposed K8S schedulers. Carmen Carrión [17] reviews the proposed K8S scheduling techniques and proposes a new taxonomy

to categorise them according to the following domains: (1) application, which refers to the kind of managed applications; (2) scheduling, i.e., the type of algorithms it uses and the supported metrics; (3) cluster, which relates to the cluster characteristics supported by the scheduler; (4) infrastructure, referring to the type of managed infrastructure; (5) and performance, i.e., how is the scheduler performance evaluated. Zeineb Rejiba and Javad Chamanara [87] provide a classification for the community-proposed K8S schedulers, considering the scheduler objectives and the handled workloads and environment types. The authors also provide an overview of the main approaches used to address each scheduling goal. Senjab et al. [98] review and characterise the community-proposed K8S scheduling algorithms, grouping them in generic scheduling, multi-objective optimisation-based scheduling, AI-focused scheduling, and autoscaling-enabled scheduling. Analysing these surveys, we can conclude that in the scope of K8S, the community also considers a specific set of quality metrics related to the target application objectives or application domain. Carmen Carrión [17] highlights this point while stating that the default K8S scheduler is not designed for a particular application.

2.3.3 Positioning IoT Quality-aware provisioning in the Scope of IoT Orchestration

As highlighted in Section 2.3.1, the community is proposing multiple perspectives to improve the characterisation of IoT applications quality, since the traditional quality perspectives are not sufficient to categorise all the quality aspects of such applications. In this scope, we consider that entity-based perspectives are the most relevant ones because they cover multiple aspects of IoT, independently of the hierarchical or layered architecture organisation. The hierarchical organisation is too abstract, and the layered architecture might become outdated with the technological evolution (for example, sensing/actuation and processing layers are currently not clearly segmented in different devices or application components). Fizza et al. [32] proposal is currently the one that provides a set of quality characteristics, which seems broad enough to cover all the quality aspects of IoT applications (we can add QoE to the list of proposed quality characteristics if the IoT application is not fully autonomous). However, the identified metrics might not truly represent the quality of IoT applications across the diversity of IoT applications use cases. The quality metrics of the IoT application depend on the use case and might be a subset of the ones identified in the literature or be outside that set. In fact, Fizza et al. propose a framework to monitor the quality of autonomic IoT applications and validate it in a food paste processing setup [33]. In this concrete scenario, the authors only use a subset of the quality characteristics and metrics identified in their previous work [32]. This is in line with the conclusion of Aggeliki Sgora and Periklis Chatzimisios in their literature review [99] to collect the quality metrics used in the IoT domain, which states that the application domain influences the set of considered metrics. In fact, Moulla et al. [75] conducted a systematic literature review to identify IoT quality metrics with no restrictions on application domains and use cases and reached a total of 158 metrics.

Thus, an IoT orchestration framework that effectively provisions and manages its applications based on their quality should be modular enough to handle the dynamics of quality characteristics and metrics relevant to each application. In section 2.3.2, we explored existing quality-aware

provisioning solutions considering this perspective and concluded that the community proposals considering a broader view on IoT application quality are tied to a certain set of quality characteristics and metrics relevant to their specific use case. Although this approach provides fine-tuned scheduling, which might be adequate for some use cases, it requires the infrastructure or the application provider to develop an orchestration algorithm or scheduler for each running application. We argue that having a generic and modular solution that frees the infrastructure and application providers from such responsibility is more scalable and promotes IoT dissemination. For the infrastructure provider, creating a scheduler for each running application is a cumbersome job due to the dynamism of IoT applications. On the other hand, the application providers might not have the knowledge to build their own scheduler or may not need such fine-tuned scheduling. In the scope of K8S, we argue that with the integration of IoT devices in its orchestration procedures, the K8S scheduler should be aware of IoT application quality and follow the same approach as the current K8S default scheduler, i.e., be generic and adaptable to different use cases.

Section 5 presents the Quality-aware Orchestration for IoT Applications (QOIA), our proposal for a configurable IoT Quality-aware orchestration that handles application component placement and life cycle management using K8S, and is not tied to a particular set of quality metrics or models.

2.4 Summary

In this chapter, we reviewed the state of the art on: 1) IoT orchestration; 2) IoT interoperability; and IoT applications quality.

Regarding IoT orchestration, we have studied the taxonomy proposed by Nguyen et al. [76] for IoT orchestration frameworks and integrated it with the conclusions by Velasquez et al. [108] on the requirements and challenges for developing Fog orchestration frameworks. We have used the resulting taxonomy to analyse the current IoT orchestration frameworks proposed by the community, concluding that none solves the IoT continuum orchestration issue, because three fundamental issues are not yet solved, namely: 1) the IoT device heterogeneity regarding interaction and data models; 2) the IoT device heterogeneous runtime environments; 3) and the deployment and recovery of IoT application components, considering their quality requirements. These are the issues that map to the research questions we use to validate our Thesis.

In what respects IoT interoperability, we reviewed the standardisation efforts in this scope and then focused on the standard data and information modelling efforts. We concluded that these efforts are focused on building data models and ontologies, and that, for the scope of this Thesis, our interest is in standard data models. The proposed standard data models are heterogeneous themselves, with the majority focusing on specific use cases. IETF SDF and WoT TD stand out due to their use case agnostic nature and compatibility with the other standards. However, despite their relevance, none of the interoperability gateways and middlewares proposed by the community use them in their proposals. The majority of these gateways and middlewares are not standards-based, and when they are, they are tied to a specific use case that is inherited from

the used standard. For the scope of IoT orchestration, we argue that the interoperability gateway should be use-case agnostic, and we identified IETF SDF as the most adequate solution due to its homogenization in a single data and interaction model.

Concerning IoT applications quality, we identified that the definition of quality for IoT applications is a topic actively discussed in the literature, because the traditional QoS and QoE are not enough to model it. The community examines this issue from various perspectives, proposing multiple quality characteristics and metrics that are often associated with a specific application or application domain. Considering this, we concluded that no closed set of characteristics and metrics defines the quality of IoT applications, and that IoT provisioning solutions should be adaptable to different quality characteristics, metrics, and models to allow their use with various applications and provisioning goals. However, after examining the provisioning solutions proposed by the community (both for application component placement and service selection), we concluded that they are fine-tuned to a specific application and provisioning goal and not adaptable to others. We argued that this is not scalable and hampers IoT dissemination, and identified the need for a configurable K8S scheduler that adapts to the different quality requirements of IoT applications.

Chapter 3

NextGenGW: Towards IoT Interoperability

Section 2.2 explored IoT interoperability, reviewing the existing standardisation efforts to address it. Among the different topics addressed by standardisation organisations (device communication, network and device management, as well as data and information modelling), our focus within the scope of IoT orchestration is on data and information modelling. In this respect, the standards focus both on data models with syntactic and semantic richness (ex., IoTivity [82], LwM2M [69], WoT TD [50], SDF [56]), as well as ontologies (ex.,SAREF, SSN and SOSA [5]).

Data models, ideally domain-independent ones, could be used to build orchestration frameworks that reason over device properties, configurations and status. Ontologies could be used on top of that to provide domain awareness about the classes, procedures, and relationships of the devices, as well as the specific processes in which they are involved. In the scope of this Thesis, we consider the usage of syntactic and semantic data models to tackle domain-independent orchestration. The study and usage of ontologies for fine-tuned domain-specific IoT orchestration is another research direction beyond the scope of this Thesis, which we briefly explore in Section 6.2.

The State-of-the-Art on IoT gateways and middlewares targeting data model syntactic and semantic interoperability showed that none of the community proposals completely solves it. Non-standards-based proposals [3][112][58][71][77][21][95] do not effectively address IoT interoperability due to their reliance on proprietary abstractions, which can introduce vendor lock-in, limit applicability to specific domains, or lack critical IoT features, such as actuator representation and change subscription support. In contrast, standards-based proposals [24][111][64] mitigate vendor lock-in but still struggle with domain specificity and incomplete support for essential IoT functionalities. These limitations stem from the underlying data and interaction model used.

As we referred in the previous chapter, two of the most prominent, domain-independent, standard data models are the WoT TD and IETF SDF. However, the community has not yet provided a solution for interoperability based on them. WoT TD exposes each device, its properties and interactions in a structured way. SDF focuses on establishing a common semantic foundation for

data models, enabling consistent interpretation of data across ecosystems. SDF aims to be complementary to the other data modelling standards, enabling, through the use of translators, the accommodation of their particularities, including the ones from WoT TD [93].

In this chapter, we propose NextGenGW [89], a standards-based software framework that addresses the data model syntactic and semantic interoperability problem, aiming for domain independence. NextGenGW uses IETF SDF for homogenising devices interaction and data heterogeneity, being, to the best of our knowledge, the first proposal to employ such an approach. We have selected IETF SDF over WoT TD because of its homogenisation in a single semantic definition format, and the data model standardisation effort for IoT devices and objects provided by OneDM. Section 3.1 briefly describes IETF SDF, providing the basis for understanding NextGenGW. Section 3.2 details NextGenGW architecture and implementation details. Section 3.3 presents its evaluation. Section 3.4 summarises this chapter.

3.1 Background Technology

IETF SDF is a semantic definition format focused on modelling the physical objects that are available for interaction over a network (Things in the IETF SDF terminology). In this section, we provide an overview of version 11 of the SDF specification [56], the one we used in NextGenGW.

The definition format follows the hierarchical organisation presented in Figure 3.1.

sdfThing is the top-level element of the hierarchy. It is the element that represents the Thing. It can contain both *sdfObjects* and other *sdfThings*, which means we can reuse *sdfThing* definitions to create representations of Things that include other Things defined in another SDF file. We can use the Raspberry Pi ecosystem as an example of *sdfThings* that include other *sdfThings*. The Raspberry Pi itself is an *sdfThing*, but its extension boards might also be sdfThings themselves, so when we connect them, the resulting Thing is composed of two *sdfThings*, the Raspberry Pi and the extension board. In the end, at the cost of losing definition reutilisation, one can always decompose the included *sdfThings* into their *sdfObjects*, reaching an *sdfThing* that is only composed of *sdfObjects*.

sdfObject is composed of a group of zero or more *sdfProperties*, *sdfActions*, *sdfEvents* and *sdfData*. *sdfObject* is the key reusable element of the SDF model, so it should be kept narrow in scope to allow its broad reuse and interoperability. For example, if we want to model an *sdfThing* that is composed of a button and a light sensor, we should model the button as an *sdfObject* and the light sensor as another *sdfObject*. Modelling them together in a single, and more complex, *sdfObject* hampers its reusability.

sdfProperty models elements of state within *sdfObject* instances, which can be monitored or updated through read, write, and observe operations. For example, the *sdfProperty* "Sensor_Value" of the *sdfObject* "temperature" models the temperature measured by a temperature sensor, while the *sdfProperty* "Measurement_Quality_Indicator" models the quality level of that measured value.

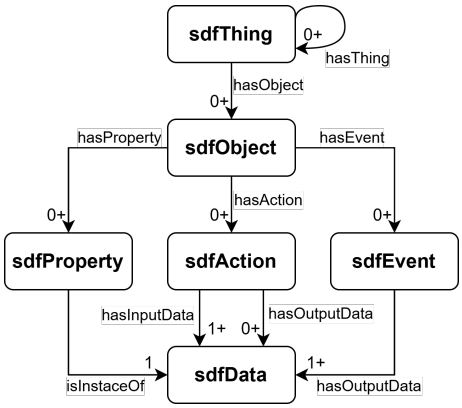


Figure 3.1: SDF elements and their relation.

sdfAction models commands and methods that can be sent to the *sdfObjects*. The *sdfAction* semantics differ from the semantics of a read, write and observe on an *sdfProperty*. When an *sdfAction* is triggered, such commands or methods should have more effect than just reading, updating or observing the state of a Thing. The commands or method should be translated to a "physical effect". Taking the *sdfObject* "temperature" as an example, its *sdfActions* "Reset_Min_and_Max_Measured_Values" and "recalibrate", should clean the register where the measured minimum and maximum values are stored, and recalibrate the sensor, respectively. An *sdfAction* has optional input and output data that allow passing arguments to the command or method and to receive outputs from such a call.

sdfEvent also differs semantically from the *sdfProperty* and its associated operations. An *sdfEvent* models asynchronous occurrences that may be communicated proactively, i.e., without being explicitly requested by a read or observe request. Additionally, *sdfEvent* should guarantee a certain ordering, consistency, and reliability, which also distinguishes it from *sdfProperty*. That is, while a property state change might be superseded by another state change, events might need to be preserved even if other events follow. The current NextGenGW version does not support this SDF element because it is not mandatory for the validation of our Thesis.

sdfData models reusable data types definition to be used in *sdfProperty*, *sdfActions* and *sdfEvent* to specify their data types. Following the example of the *sdfObject* "temperature", its *sdfProperty* "Sensor_Value" could be associated with an *sdfData* "temperatureData" composed of {"type": "number", "unit": "Cel"}, which specifies that the value is a number in degrees Celsius. The current NextGenGW version also does not support this SDF element because it is not mandatory for the validation of our Thesis.

The SDF specification is still composed of other details, which we have excluded from this section for simplicity and because they are not relevant for understanding NextGenGW.

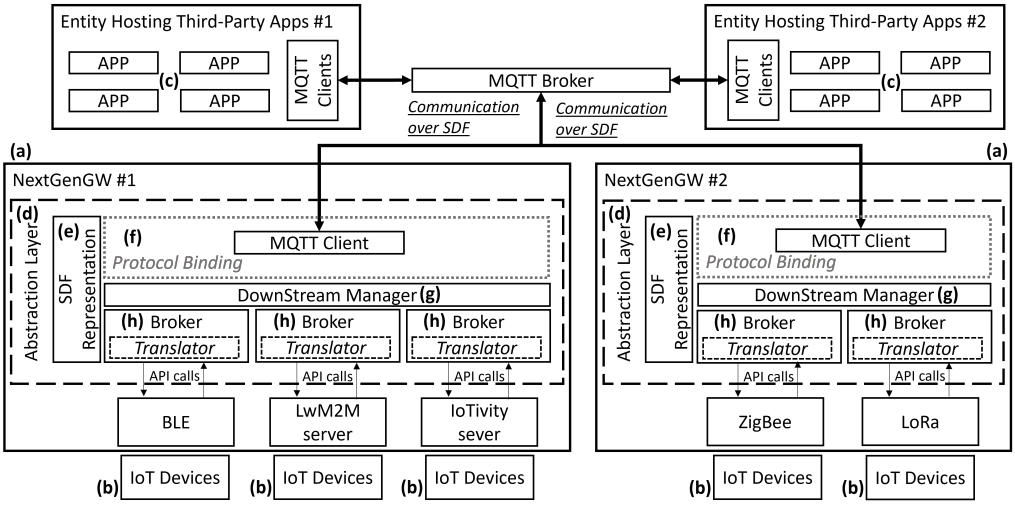


Figure 3.2: NextGenGW high-level architecture.

3.2 NextGenGW

NextGenGW addresses the heterogeneity in interaction and data models of IoT devices, abstracting and harmonising them with IETF SDF. With this approach, we model IoT devices, their data, and available operations, thereby addressing both syntactic and semantic interoperability.

Figure 3.2 presents the NextGenGW architecture (cf. a) and how it relates to the IoT devices (cf. b) and the entities hosting the third-party applications that need to communicate with such devices (cf. c). An installation can host multiple instances of the NextGenGW (cf. NextGenGW #1 and NextGenGW #2), which can be running in any layer of the IoT stack (as per Figure 1.1), in virtualised or non-virtualised environments. The decision on where and how to instantiate NextGenGW is installation-dependent, and we do not assume any specific scenario. With the objective of being as versatile as possible, we implemented NextGenGW in C++, aiming for portability and performance. Each instance of the NextGenGW provides an "Abstraction Layer" (cf. d) through which the IoT devices and the third-party applications interact. This "Abstraction Layer" is composed of the "SDF Representation" (cf. e), "Protocol Binding" (cf. f), "DownStream Manager" (cf. g), and "Broker" (cf. h) modules, which we detail in Sections 3.2.1 to 3.2.4, respectively. Section 3.2.5 further explains NextGenGW workflow.

3.2.1 SDF Representation

The "SDF Representation" module provides the internal structures to represent SDF Things, Objects, Properties and Actions, which we use for exchanging information between the different modules of the NextGenGW. Figure 3.3 presents the OnedmSdf Class, which is the C++ class that instantiates this module. It is composed of a set of C++ structs for each SDF element currently supported by NextGenGW, namely: sdfThing (cf. a), sdfAction (cf. b), sdfProperty (cf. c) and sdfObject (cf. d). The sdfPropertyInstance (cf. e) and sdfObjectInstance (cf. f) hold the

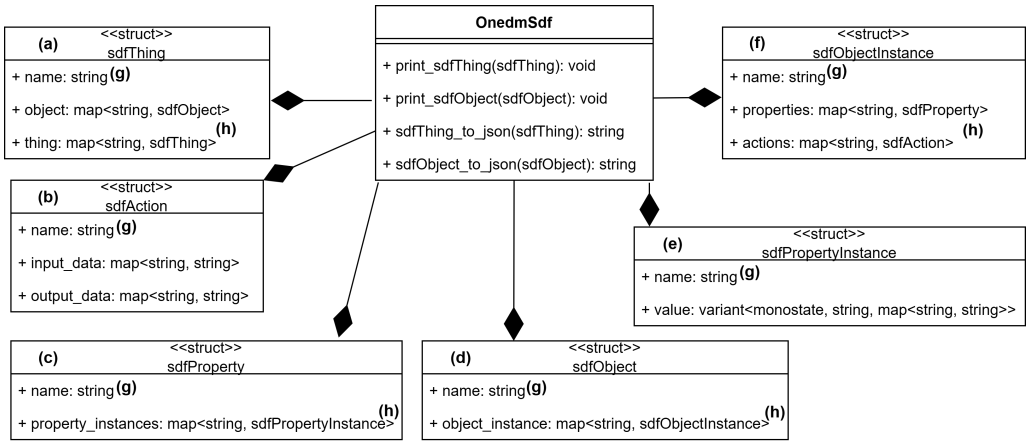


Figure 3.3: NextGenGW's OnedmSdf class.

instance information of these SDF elements. Regarding the sdfPropertyInstance, it holds the property value, which can be a single string or a map containing a list of values and their respective IDs, both of which we encode as strings. This encoding means that the property value is always coded as a string (directly or in a map), and the interpretation of the string content (string, number, boolean, etc.) is done using the information present in the SDF model file that specifies the property. Regarding the sdfObjectInstance, it holds multiple sdfObject instances as per the SDF specification. It is composed of the properties and actions of the object, as well as the name that individually identifies each instance. In fact, we use the name as an identifier for each SDF element, so this attribute is present in all structs (cf. g). To ease the navigation between structs, we also use the name as the key for the maps that link each struct to the other (cf. h). This way, for example, when we are searching for a certain sdfProperty within an sdfObjectInstance, we can directly access it by the map key, instead of searching the multiple elements of the map for the one with the correct name.

3.2.2 Protocol Binding

"Protocol Binding" module holds the binding between SDF and the application protocol, which exposes to its clients the operations GET, POST, PUT, DELETE, OBSERVE_START and OBSERVE_STOP (interpreted according to their definition in [100, 39]), as well as DISCOVERY (for discovering the devices connected to the NextGenGW). We want to note that: (1) using a POST operation in an sdfAction is a request to perform that action; and (2) the DISCOVERY operation is forwarded by the "DownStream Manager" to all "Brokers", which will either return the list of registered devices or scan for new available devices, depending on the "Broker" communication protocol and its specification.

In addition to exposing these operations to its clients, the "Protocol Binding" exposes the callback functions in Listing 3.1, which are used by the "DownStream Manager" to link the messages from the IoT devices received by the "Brokers" with the "Protocol Binding".

Listing 3.1: Protocol Binding functions exposed to the DownStream Manager to link the IoT devices messages with the Protocol Binding.

```
1 void get_callback(uint8_t response_code, map<string, OnedmSdf::sdfObject> object_list,
      string response_path);
2 void post_callback(uint8_t response_code, map<string, OnedmSdf::sdfObject> object_list,
      string response_path);
3 void put_callback(uint8_t response_code, map<string, OnedmSdf::sdfObject> object_list,
      string response_path);
4 void del_callback(uint8_t response_code, map<string, OnedmSdf::sdfObject> object_list,
      string response_path);
5 void announce_callback(OnedmSdf::sdfThing thing, uint8_t update_type);
6 void observe_callback(OnedmSdf::sdfThing thing);
7 void unregistered_callback(string thing_id);
```

The functions *"get_callback"*, *"post_callback"*, *"put_callback"*, and *"del_callback"* (lines 1 to 4, respectively) are associated with the responses to the GET, POST, PUT and DELETE operations, respectively. These operations are of type request-response, and because of this, a response code and response path are available as parameters of the callback functions (the first and last parameters, respectively). These parameters enable the identification of error codes in the response to the request and the path where the "Protocol Binding" mechanism should share such response, which can be useful for publish-subscribe protocols such as MQTT. The other parameter of these functions is a map of sdfObjects (*"map<string, OnedmSdf::sdfObject> object_list"*), which holds the content of the response. One thing to note is that the OBSERVE_START, OBSERVE_STOP and DISCOVERY operations are not associated with a request-response callback, because they are event-driven operations, i.e., the "Broker" does not provide the updates on the observed properties and the discovered IoT devices in a single response, but as events when it detects updates on the observed properties or discovers IoT devices. Nevertheless, future work includes creating such callback functions to allow sharing immediate errors or other response codes related to such operation requests.

The *"announce_callback"*, *"unregistered_callback"*, and *"observe_callback"* (lines 5 to 7, respectively) are event-driven callbacks associated with the OBSERVE_START and DISCOVERY operations. We assume that the event-driven callbacks do not contain a response code and that the response path is fixed by the binding protocol and not defined by third-party applications. The "DownStream Manager" invokes the *"announce_callback"* in three situations: (1) when a new device registers itself with a "Broker"; (2) when a registered device updates its list of sdfObjects, sdfProperties, or sdfActions; or (3) when the "Broker" shares the available devices after the DISCOVERY operation. The *"announce_callback"* parameters contain the updated IoT device in SDF format (*"OnedmSdf::sdfThing thing"*) and the type of update (*"uint8_t update_type"*, which might represent a new device or changes on the sdfObjects, sdfProperties, or sdfActions of a registered device). The *"observe_callback"* function shares updates that result from the OBSERVE_START operation through the *"OnedmSdf::sdfThing thing"* parameter. The "DownStream Manager" invokes the *"unregistered_callback"* function when a device is unregistered from NextGenGW. The parameter of this function (*"string thing_id"*) contains the ID of the unregistered device.

In its current version, NextGenGW binds SDF with MQTT ("MQTT Client" detailed in Section 3.2.2.1), taking into account these operations and callback functions. However, we designed NextGenGW taking into consideration the possibility of binding SDF with other communication protocols (namely CoAP). We believe there is no ideal communication protocol that fits all use cases, and this approach enables the experimentation of bindings with other communication protocols and the adaptation of the solution for different use cases. The current support for MQTT stems from its maturity, ease of use, and alignment with NextGenGW's role in the scope of IoT continuum orchestration. NextGenGW is the interoperability gateway between IoT devices and the orchestration framework, where MQTT is the protocol exposed to the orchestration framework. In this use case, NextGenGW may be running on the Edge (on the device to which the IoT devices are connected) or in the Cloud (to where the messages from IoT devices are forwarded) and should provide reliable, bidirectional communication and mechanisms for both telemetry ingestion and the reception of configuration or action requests. MQTT guarantees reliable communication through its configurable QoS levels, which allow for balancing resource usage with reliability. The publish–subscribe messaging for telemetry data, as well as request–response patterns for configuration and action requests, is supported by MQTT version 5.0. Additionally, the network path between NextGenGW and the orchestration framework might be composed of Network Address Translation (NAT), which MQTT's persistent TCP sessions easily handle. Finally, commercial MQTT brokers, such as HiveMQ[1] and EMQX[2], offer mature horizontal scaling and load balancing, which might be helpful in installations with multiple interoperability gateways and third-party clients connecting to them, such as the one represented in Figure 3.2.

3.2.2.1 MQTT Application Protocol and its Alignment with SDF

At boot time, the "MQTT Client" is just subscribed to the "discovery" topic. Through this topic, the third-party applications can send the DISCOVERY operation. The IoT devices that result from the DISCOVERY operation are published by the "MQTT Client" in the "announce" topic. The "MQTT Client" also uses this topic to publish the other events associated with the *"announce_callback"* function (line 5, Listing 3.1), i.e., device self-registration and device update. The payload that goes on the messages published on the announce topic contains the IoT device in SDF, and follows the template of Listing 3.2. We create this JSON using the *"OnedmSdf::sdfThing"* that comes as an argument in the *"announce_callback"* function. In the JSON, <thing_identifier> (line 1) is a unique identification of the IoT device, <object_name> (line 4) is its SDF object name, and <prop_name> and <action_name> (line 5 and 9, respectively) are the SDF names of the property and action that compose the object. We use the SDF quality "label" (line 3) to specify the ID of the multiple instances of the same object.

When publishing an IoT device in the announce topic, the "MQTT Client" also subscribes to the MQTT topics associated with the announced IoT device, which follow the SDF hierarchy, and we organised as follows:

[1] https://www.hivemq.com/
[2] https://www.emqx.com/en

Listing 3.2: JSON payload for announcing IoT Devices.

```
1  {"<thing_identifier>": {
2    "sdfObject": {
3      "label": "<instance_id>",
4      "<object_name>": {
5        "sdfProperty": {
6          "<prop_name>": "<prop_value>", (...)
7        },
8        "sdfAction": {
9          "<action_name>": "<action_value>", (...)
10       },
11     }, (...)
12   }}}
```

- <thing_identifier>/<object_name>/<instance_id>
- <thing_identifier>/<object_name>/<instance_id>/Property/<prop_name>
- <thing_identifier>/<object_name>/<instance_id>/Action/<action_name>

In these topics, <thing_identifier>, <object_name>, <prop_name> and <action_name> have the same meaning as the payload in Listing 3.2. <instance_id> represents the ID of the sdfObject instance. If the sdfObject has no instances, then the <instance_id> is omitted from the MQTT topics. With this association between topics and IoT devices, one topic is created for each sdfObject, sdfProperty and sdfAction that compose the IoT device, allowing the third-party applications to interact with IoT devices at the object, property, and action levels. To achieve this, we transpose the SDF hierarchy to the payload that goes with the publish requests. The payload comprises a JSON message with the format presented in Listing 3.3 for the publish requests at the object level and the format presented in Listing 3.4 for the publish requests at the property and action level.

In Listings 3.3 and 3.4, <request> holds the GET, POST, PUT, DELETE, OBSERVE_START and OBSERVE_STOP operations. For request-reply operations (GET, POST, PUT and DELETE), the "MQTT Client" allows the third-party applications to specify the response path using the MQTT version 5.0 response topic feature. The payload of the message associated with such responses is the JSON of an SDF object, regardless of whether the request was targeting an object or a property. The JSON is composed by the "MQTT Client" considering the *"OnedmSdf::sdfObject"* that comes as an argument in these operations callback functions (lines 1 to 4, Listing 3.1).

Listing 3.3: JSON payload for request at the object level.

```
1  { "operation": "<request>",
2    "data": {
3      "sdfProperty": {
4        "<prop_name>": "<prop_value>", (...)
5      },
6      "sdfAction": {
7        "<action_name>": "<action_value>", (...)
8      }
9  }}}
```

Listing 3.4: JSON payload for request at the property and action level.

```
1  { "operation": "<request>",
2    "data": "<value>" }
```

The event-driven messages resulting from property updates triggered by an OBSERVE_START operation are published by the "MQTT Client" on the topic corresponding to the observed property. The "MQTT Client" uses the *"OnedmSdf::sdfThing"* argument of the *"observe_callback"* function (line 6, Listing 3.1) to identify the publish topic and build the payload of the publish message, which is just composed of the updated value.

3.2.3 DownStream Manager

The "DownStream Manager" bridges the "Protocol Binding" mechanism with the "Brokers". It does this by mapping the GET, POST, PUT, DELETE, OBSERVE_START, OBSERVE_STOP and DISCOVERY operations of the bounded protocol to the corresponding operations in the "Broker". This mapping is performed using the functions in Listing 3.5, which are invoked by the "Protocol Binding" mechanism when it receives a request to execute one of these operations. The "DownStream Manager" identifies the "Broker" to which the request is addressed using the IoT device name present on the argument *"OnedmSdf::sdfThing thing"* of the functions in Listing 3.5. The argument *"OnedmSdf::sdfThing thing"* is defined by the "Protocol Binding" mechanism according to the operation request details. *"OnedmSdf::sdfThing thing"* is an argument in all functions except "discovery", which is a broadcast operation. The *"string response_path"* argument in the *"get"*, *"post"*, *"put"*, and *"del"* functions (lines 1 to 4) is used to pass to the "Broker" the path where the third-party applications want the response to be made available.

In the opposite direction, the "DownStream Manager" bridges the "Protocol Binding" callback functions with the "Brokers" through its constructor (line 1 in Listing 3.6). At boot time, the "DownStream Manager" is configured with pointers to the callback functions defined by the "Protocol Binding" mechanism (Listing 3.1). These callback functions are invoked by the "DownStream Manager" callback functions (lines 2 to 14 in Listing 3.6), which are triggered by the "Broker" when it receives a message from the IoT devices. We do not directly connect the "Protocol Binding" callback functions with the "Brokers" because the "DownStream Manager" needs to be aware of such messages to track the registration and removal of devices at each "Broker".

Listing 3.5: DownStream Manager functions exposed to Protocol Binding to forward third-party requested operations to the IoT devices.

```
1  void get(OnedmSdf::sdfThing thing, string response_path);
2  void post(OnedmSdf::sdfThing thing, string response_path);
3  void put(OnedmSdf::sdfThing thing, string response_path);
4  void del(OnedmSdf::sdfThing thing, string response_path);
5  void start_observe(OnedmSdf::sdfThing thing);
6  void stop_observe(OnedmSdf::sdfThing thing);
7  void discovery();
```

Listing 3.6: DownStream Manager constructor and callback functions.

```
1  DownstreamManager(void (*announce_callback)(OnedmSdf::sdfThing, uint8_t),
2              void (*unregistered_callback)(string),
3              void (*get_callback)(uint8_t, map<string, OnedmSdf::sdfObject>,string),
4              void (*post_callback)(uint8_t, map<string, OnedmSdf::sdfObject>,string),
5              void (*put_callback)(uint8_t, map<string, OnedmSdf::sdfObject>,string),
6              void (*del_callback)(uint8_t, map<string, OnedmSdf::sdfObject>,string),
7              void (*observe_callback)(uint8_t, OnedmSdf::sdfThing))
8  void dm_get_callback(string broker_id, uint8_t response_code, map<string, OnedmSdf::
       sdfObject> object_list, string response_path);
9  void dm_post_callback(string broker_id, uint8_t response_code, map<string, OnedmSdf::
       sdfObject> object_list, string response_path);
10 void dm_put_callback(string broker_id, uint8_t response_code, map<string, OnedmSdf::
       sdfObject> object_list, string response_path);
11 void dm_del_callback(string broker_id, uint8_t response_code, map<string, OnedmSdf::
       sdfObject> object_list, string response_path);
12 void dm_obs_callback(string broker_id, uint8_t, OnedmSdf::sdfThing thing);
13 void dm_announce_callback(string broker_id,OnedmSdf::sdfThing thing,uint8_t update_type);
14 void dm_unregistered_callback(string broker_id, string thing_id);
```

3.2.4 Broker

The "Broker" is responsible for communicating with the IoT devices associated with it. It instantiates the endpoint responsible for such communication (LwM2M server, IoTivity server, etc.) and interacts with it through its API calls. The "Broker" uses the "Translator" to translate requests received from the Downstream manager (which use the structures defined by the "SDF Representation" module) into the specific data model of the IoT device and vice versa. NextGenGW holds one pair of "Broker" and "Translator" for each IoT device communication protocol (e.g., BLE, LoRa, ZigBee, LwM2M, and IoTivity). It loads each "Broker" as a shared library, allowing for the easy inclusion and removal of "Brokers" according to the communication protocols of the IoT devices present at each installation. These libraries expose their functionalities to the "DownStream Manager" through the set of functions in Listing 3.7. The functions *broker_get*, *broker_post*, *broker_put*, *broker_del*, *broker_start_observe*, *broker_stop_observe*, and *broker_discovery* (lines 1 to 7) are the functions invoked by the "DownStream Manager" when executing the functions in Listing 3.5, respectively. The "Broker" uses the argument *"OnedmSdf::sdfThing thing"* to identify the

Listing 3.7: Broker public functions.

```
1  void broker_get(OnedmSdf::sdfThing thing, string response_path);
2  void broker_post(OnedmSdf::sdfThing thing, string response_path);
3  void broker_put(OnedmSdf::sdfThing thing, string response_path);
4  void broker_del(OnedmSdf::sdfThing thing, string response_path);
5  void broker_start_observe(OnedmSdf::sdfThing thing);
6  void broker_stop_observe(OnedmSdf::sdfThing thing);
7  void broker_discovery();
8  BrokerBuilder(string broker_id,
9          void (*announce_callback)(string, OnedmSdf::sdfThing, uint8_t),
10         void (*unregistered_callback)(string, string),
11         void (*get_callback)(string, uint8_t, map<string, OnedmSdf::sdfObject>,string),
12         void (*post_callback)(string, uint8_t, map<string, OnedmSdf::sdfObject>,string),
13         void (*put_callback)(string, uint8_t, map<string, OnedmSdf::sdfObject>,string),
14         void (*del_callback)(string, uint8_t, map<string, OnedmSdf::sdfObject>,string),
15         void (*obs_callback)(string, uint8_t, OnedmSdf::sdfThing));
```

SDF element to which the operation refers. The *BrokerBuilder* (line 8, Listing 3.7) is the "Broker" constructor and allows the "DownStream Manager" to set the links to its callback functions (lines 8 to 14, Listing 3.6), as well as the "Broker" ID (*"string broker_id"*), which is defined by it and is used by the "Broker" when invoking the "DownStream Manager" callback functions.

3.2.5 Execution Workflow

Figures 3.4 to 3.6 present NextGenGW's activity diagrams, which show its operation and the relationships between the several modules that compose it.

Figure 3.4 presents the NextGenGW boot procedure. This activity begins with the creation and initialisation of the "MQTT client" (cf. a), which runs as a new thread that waits for messages on the subscribed topics (cf. b). The topics subscribed by the "MQTT Client" are explained in Section 3.2.2.1. In addition to instantiating the "MQTT Client", the NextGenGW starts the "Brokers" responsible for communicating with the IoT devices (cf. c). Each "Broker" runs on its own thread, and the instantiation of the "Broker" is managed by the "DownStream Manager", which loads its library and sets the links to its callback functions discussed in Section 3.2.3 (cf. d). After booting, the "Broker" waits for messages from the IoT devices (cf. e).

Figures 3.5 present the processing of messages coming from the third-party applications. These messages are received by the "MQTT Client" that validates if the response topic specified in the MQTT message is within the limit of thirty-three characters (cf. a). If that is not the case, it discards the message (cf. b). Otherwise, it creates a thread to handle the received message (cf. c). This thread parses the MQTT topic and payload to get the operation request (GET, POST, PUT, DELETE, OBSERVE_START, OBSERVE_STOP and DISCOVERY) contained on it and creates the operation payload according to the "SDF Representation" structures. If parsing is not successful, the thread exits (cf. d). Otherwise, the "MQTT Client" identifies the requested operation and invokes the corresponding "DownStream Manager" function (cf. e). In all functions, the "DownStream Manager" uses the device name addressed by the operation to identify the "Broker" to forward the request (cf. f). If no "Broker" has the identified device registered, the thread exits. Otherwise, it invokes the respective function of the "Broker" that hosts the device (cf. g). At this point, the remaining steps to reach the IoT devices depend on each "Broker" implementation. All of them will have to use a "Translator" to convert the operation payload (which uses the structures specified by the "SDF Representation" module) to the payload of the communication protocol it is instantiating. Then they will use the result of the translation to communicate with the IoT device.

The handling of messages coming from the IoT devices is presented in Figure 3.6. The "Broker" receives these messages, translates them to SDF using its "Translator" (cf. a) and interprets the message type, which can be: (1) a response to an operation request (GET, POST, PUT and DELETE) (cf. b); (2) an update associated with an observe procedure (cf. c); or (3) a device update or removal (cf. d). Depending on the message, the "Broker" calls the respective "DowStream Manager" callback function (cf. e), setting its parameters (which use the structures of the "SDF Representation" module) with the translation results. If the message is a device update or removal,

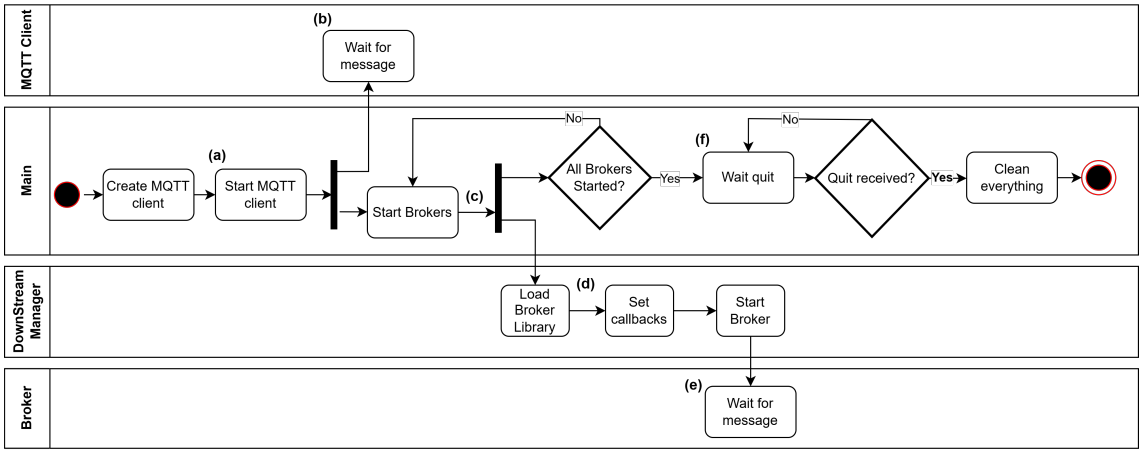


Figure 3.4: Activity diagram of NextGenGW boot procedure.

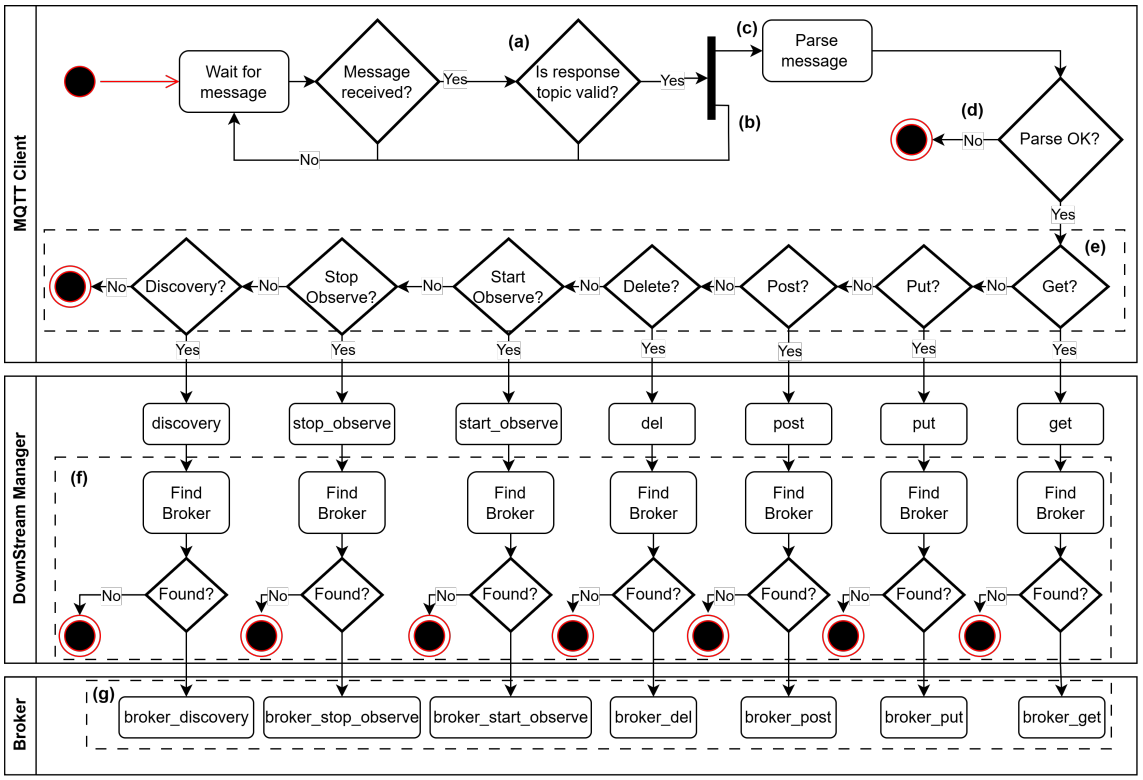


Figure 3.5: Activity diagram of NextGenGW client request handling.

the "DowStream Manager" updates its list of IoT devices per "Broker" before forwarding the message to the "MQTT Client" (cf. f). Otherwise, it directly forwards the messages to the "MQTT Client" (cf. g). At the "MQTT Client" (cf. h), its callback function interprets the SDF content and binds it to MQTT following the procedure defined in Section 3.2.2.1. The message handling ends with the binding result published in the associated MQTT topic.

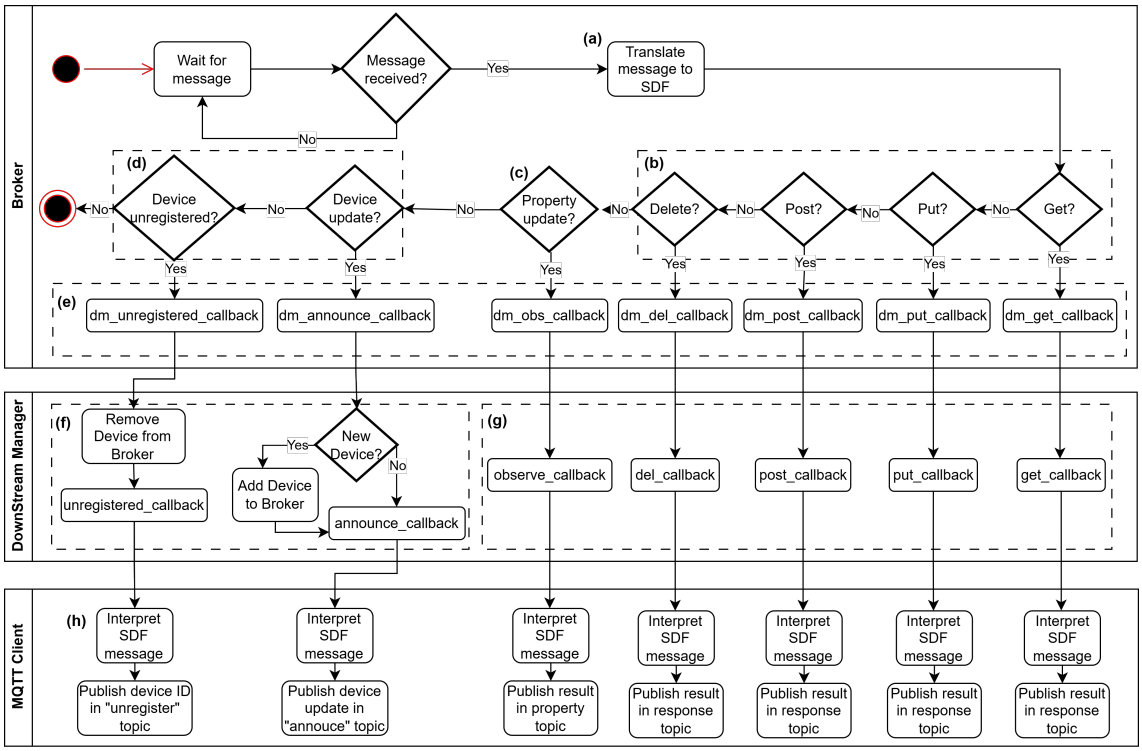


Figure 3.6: Activity diagram of NextGenGW IoT device message handling.

3.3 NextGenGW Evaluation

For the evaluation of NextGenGW, we examine in Section 3.3.1 the evaluation strategies employed by the interoperability solutions identified in Section 2.2.3 and propose a baseline for comparing IoT solutions targeting interoperability. We use the proposed baseline for NextGenGW evaluation, whose evaluation setup is detailed in Section 3.3.2 and the obtained results discussed in Section 3.3.3.

3.3.1 Performance Evaluation Baseline for Solutions Targeting Interoperability

Looking at the evaluation of the IoT gateways in the Gunjan Beniwal and Anita Singhrova survey [8], it is possible to identify that, despite not being always evaluated against the same set of parameters, they are evaluated using one or multiple parameters of the following set: response time, latency, energy consumption, bandwidth usage, security, reliability, cost, execution time and scalability.

Nugur et al. propose a gateway [77] that is evaluated in terms of memory and bandwidth usage. The test setup involves increasing stress levels, comprising both real and emulated devices. The gateway runs on an Intel Celeron® CPUJ1800@2.41GHz with Ubuntu 16.04LTS and 2 GB RAM. The total memory consumption spans from approximately 100 MB to 215 MB, depending on the load and test setup. The bandwidth is dependent on the communication protocol used by the gateway.

Aloi et al. also propose a gateway, but consider CPU, memory, and energy usage for its evaluation. The evaluation setup consists of low and high communication loads, as well as different smartphones serving as gateways with Bluetooth, WiFi, and Zigbee interfaces. CPU usage ranges from 8.17% to 13.69% at the lowest load and from 15.01% to 26.57% at the highest load. Memory usage ranges from 40.22 MB to 83.14 MB at the lowest load and from 98.72 MB to 200.96 MB at the highest load. Energy consumption of a 7.98Wh battery during a 30-minute period ranges from 12% at the lowest load to 20% and 23% at the highest load.

Xie et al. [111] propose an IoT middleware, which they evaluate in a real use case scenario considering response time, reliability, availability, and scalability. They evaluated response time by increasing the number of clients querying the middleware for search and update operations, which responded in an average of 127 ms and a maximum of less than 300 ms. Reliability was measured over a three-month period of operation, yielding a failure rate of 0.0041 times per hour and an availability of 98.8%. The authors tested scalability with up to 200 virtual sewage treatment stations, which increased the response time to 90% over the average value reported above.

Yacchirema et al. [112] evaluate MQTT and CoAP against latency and bandwidth. In terms of bandwidth, CoAP uses less than MQTT. Regarding latency, the round-trip time is 0.17 ms with CoAP and 0.29 ms with MQTT.

Li et al. [64] propose a gateway and evaluate it regarding response time and throughput when communicating with ZigBee and AllJoyn devices. Depending on the test setup, the response time varies between 30 ms and 112 ms for ZigBee and between 36 ms and 94 s for AllJoyn. The throughput ranges from 22 to 57 messages per second for ZigBee and from 25 to 62 messages per second for AllJoyn.

Considering the evaluation strategies of each of these solutions, we can conclude that the IoT solutions targeting interoperability are hardly comparable, not only due to the intrinsic characteristics of each solution and its use case but also because they do not follow a common set of evaluation metrics as depicted in Table 3.1. Given this quantitative comparison issue, we propose a baseline for evaluating IoT solutions targeting interoperability, aiming for the homogenisation of their evaluation metrics. The objective is not to provide a comprehensive evaluation guideline, but rather a baseline for IoT solutions targeting interoperability that the community should consider to facilitate comparison between different solutions.

Our evaluation baseline focuses on the overhead introduced by the interoperability solution, because it should be as non-intrusive as possible in the rest of the system. This overhead is relevant because, in most situations, the interoperability solution does not provide tangible value to the end-user of the final application.

Thus, the proposed performance evaluation baseline focuses on assessing CPU and memory usage, response time, and throughput. The CPU and memory resources consumed by the interoperability solution will impact the type and number of applications that can run along with it in the same hardware, thereby providing value to the end user. On the other hand, a high response time and low throughput will represent a bottleneck for the IoT solution, limiting the use cases that the interoperability solution can target.

Table 3.1: Evaluation metrics of IoT gateways and middleware addressing interoperability ("X" indicates the evaluated metrics; "-" indicates the metrics not evaluated)

Metric	Publication					
	[8]	[77]	[3]	[111]	[112]	[64]
Memory	-	X	X	-	-	-
Bandwidth	X	X	-	-	X	-
CPU	-	-	X	-	-	-
Energy	X	-	X	-	-	-
Response Time	X	-	-	X	-	X
Latency	X	-	-	-	X	-
Security	X	-	-	-	-	-
Reliability	X	-	-	X	-	-
Cost	X	-	-	-	-	-
Execution Time	X	-	-	-	-	-
Scalability	X	-	-	X	-	-
Availability	-	-	-	X	-	-
Throughput	-	-	-	-	-	X

One may argue that bandwidth usage also exhibits these characteristics, but this depends on the specific use case. If the interoperability solution and the applications that interact with it are running on the same hardware, then bandwidth usage becomes less relevant. The same goes for energy consumption. Its level of importance is dependent on the use case. On the other hand, security, reliability and scalability are important quality aspects for interoperability solutions, but their evaluation is more complex and, in many situations, requires a more mature implementation. This is why we have not included them in the baseline for evaluating IoT solutions. Nevertheless, we are not diminishing their importance and consider them fundamental aspects of any IoT solution targeting interoperability.

We use this baseline, along with scalability, to evaluate NextGenGW.

3.3.2 Evaluation Setup

NextGenGW evaluation assesses its performance when installed in demanding use cases with a high number of IoT devices and a high volume of requests, thereby testing the scalability of the proposed solution in terms of the number of IoT devices and the number of messages per time interval.

The evaluation was performed on a Raspberry Pi 4 Model B[3] with 2GB of RAM, acting as a WiFi access point for the IoT devices. We have used this device to run NextGenGW because we think its CPU and memory resources represent what could be available in a device to be installed near the IoT devices, acting as their access point to the remaining IoT system.

For the IoT devices, we considered two standard protocols to demonstrate NextGenGW's interoperability feature, namely LwM2M and IoTivity. The NextGenGW "Broker" for LwM2M was

[3] https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

implemented using Wakaama[4], and the LwM2M "Translator" is based on the tool provided by Ericsson Research[5], which translates the LwM2M XML-based data model to the SDF representation in JSON. The LwM2M "Translator" uses this tool to translate the LwM2M model of each IoT device to its SDF representation, and based on that model translation, it converts the messages in SDF format to LwM2M and vice versa. We implemented the IoTivity "Broker" using the IoTivity library[6]. The IoTivity "Translator" is based on the tools provided by OCF[7] to translate IoTivity Swagger2.0-based data model to the SDF representation. The IoTivity "Translator" utilises this tool to translate the specific model of each IoT device into its SDF representation, and based on that model, converts messages in SDF format to and from IoTivity.

The IoT devices were emulated using the LwM2M Wakaama client example[8] and IoTivity simpleserver example[9]. Due to the use of such examples, the LwM2M IoT devices are more complex than those running IoTivity. The emulated LwM2M IoT devices comprise 6 objects totalling 56 properties, whereas the IoTivity devices comprise 2 objects totalling 4 properties. The object "Device" is the largest LwM2M object, with 14 properties, while the object "Light" is the largest IoTivity object, with three properties.

We emulated all the IoT devices in the same hardware, a Dell Latitude 7300, which connects to the WiFi access point provided by the Raspberry Pi. With such a setup, all the IoT devices share the same WiFi connection to the Raspberry Pi. This network setup was not an issue for our evaluation, as we wanted to focus on the performance of NextGenGW's "Abstraction Layer" rather than the communication protocol of the IoT devices, whose performance depends on the conditions of the communication medium.

With this setup, we have measured NextGenGW's CPU and memory usage, response time, and throughput. Following our focus on measuring just the the performance of NextGenGW's "Abstraction Layer", our measure of response time only considers the overhead introduced by the NextGenGW's "Abstraction Layer" and disregards the time each protocol takes to send a message and receive its response, i.e., the response time is composed of the time the "Abstraction Layer" takes to deliver a request to the Wakaama LwM2M server or IoTivity client, plus the time the "Broker" takes to deliver the response to the "MQTT client" after receiving it from the IoT device.

We have measured the CPU usage, memory usage and response time under five different loads with an increasing number of IoT devices per broker, namely 1, 5, 10, 20 and 50. Such a setup evaluates NextGenGW while handling 2 IoT devices in the lowest load scenario and 100 in the highest load scenario. This dimension is relevant in practice and sufficient for scalability assessment.

Respecting the number of requests, we sent one request every 5 ms to one IoT device to evaluate the solution under a high volume of requests. Such a setup evaluates NextGenGW with

[4] https://www.eclipse.org/wakaama/
[5] https://github.com/openconnectivityfoundation/SDFtooling
[6] https://github.com/iotivity/iotivity-lite
[7] https://github.com/openconnectivityfoundation/SDFtooling
[8] https://github.com/eclipse/wakaama/tree/master/examples/client
[9] https://github.com/iotivity/iotivity-lite/blob/master/apps/simpleserver.c

two new requests in 10 ms in the lowest load scenario, and with 100 new requests in 500 ms in the highest load scenario. We repeated this procedure 10 times at 5 s intervals and applied it to READ, POST, and OBSERVE_START requests. The observation operation lasted 30 s and received updates from the LwM2M IoT devices every 5 s and from the IoTivity IoT devices every second. We performed the READ operation on the largest objects available on the IoT devices, namely the object "Device" for LwM2M and the object "Light" for IoTivity.

To have a point of comparison regarding the overhead in CPU and memory usage added by NextGenGW, we have also applied this evaluation procedure to the LwM2M and IoTivity brokers using the examples provided by these frameworks. We used the Wakaama server[10] and IoTivity simpleclient[11] examples to send the requests directly to the LwM2M client and IoTivity server, respectively.

To evaluate the throughput, we connected the NextGenGW with a single LwM2M IoT device and sent a READ request for the object "Device" every 3 ms, the saturation point of the NextGenGW. We then measured the number of responses received in the MQTT broker in 1 second.

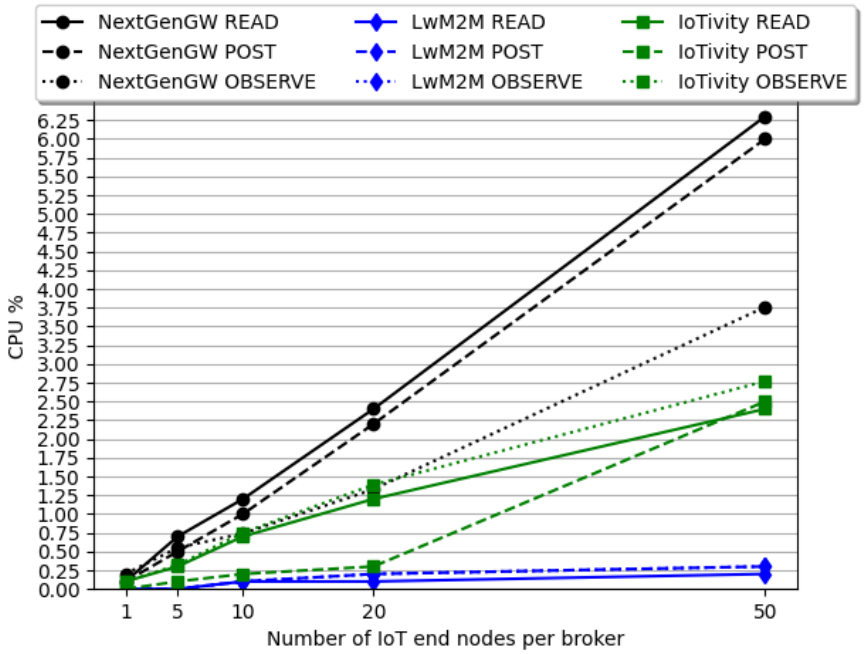
We have used Python 3 to automate the tests, so the CPU and memory usage were collected using the Python psutil package[12]. The response time was measured using the C++ chrono high-resolution clock[13].

3.3.3 Evaluation Results and Discussion

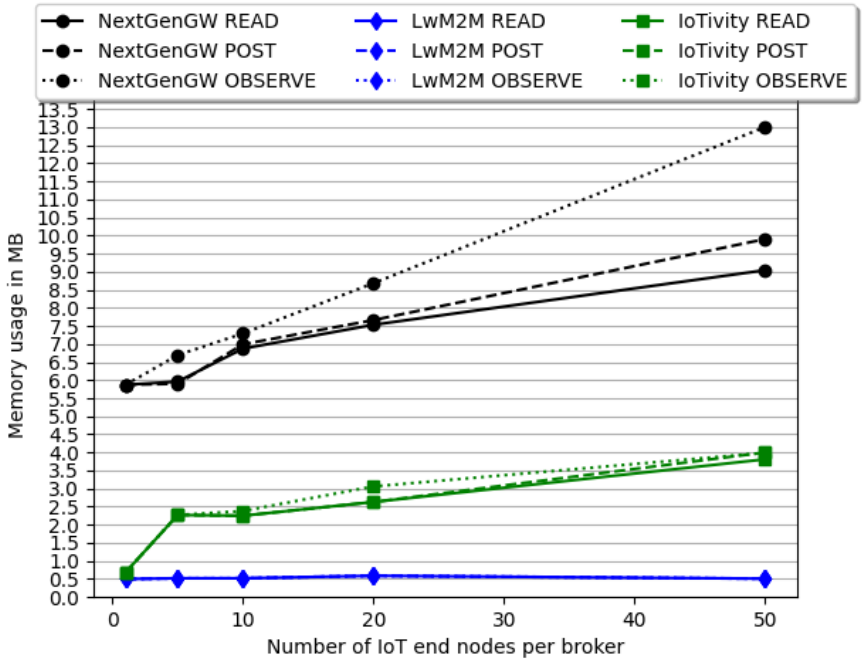
Figure 3.7a presents the CPU usage for NextGenGW, as well as LwM2M and IoTivity frameworks. As expected, the CPU usage increases with the load because more devices generate more requests. Curiously, the variation is approximately linear. Since NextGenGW is running the LwM2M and IoTivity broker simultaneously, we should compare its CPU usage with the sum of these frameworks. This exercise reveals that the NextGenGW uses approximately twice the CPU than the referred sum for the READ and POST requests, which is significant in relative terms. The highest usage overhead for the READ request is 2.4 times higher with 50 IoT devices per broker, and for the POST request is 3.3 times higher with 20 IoT devices per broker. However, the absolute additional CPU usage is low, namely between 0.1% and 3.7% for 1 and 50 IoT devices per broker. The OBSERVE request presents lower additional overhead than the READ and POST requests because the OBSERVE requests are simpler, so they can be translated with less CPU usage.

The memory usage by NextGenGW, LwM2M and IoTivity is available in Figure 3.7b. As with the CPU usage, the reported values are for each process. NextGenGW consumes between 5.37 MB with 1 IoT device per broker and 12.49 MB with 50 IoT devices per broker. This overhead is mainly related to the information saved by each broker to translate the messages from SDF

[10] https://github.com/eclipse/wakaama/tree/master/examples/server
[11] https://github.com/iotivity/iotivity-lite/blob/master/apps/simpleclient.c
[12] https://pypi.org/project/psutil/
[13] https://en.cppreference.com/w/cpp/chrono/high_resolution_clock



(a) NextGenGW, LwM2M and IoTivity CPU usage under different loads and operations.



(b) NextGenGW, LwM2M and IoTivity memory usage under different loads and operations.

Figure 3.7: NextGenGW, LwM2M and IoTivity CPU and memory usage under different loads and operations.

to the IoT device-specific format and vice versa, as well as the information regarding the connected IoT devices. Such information is independent of the READ and POST operations, thus the low variability in these operations. On the other hand, the OBSERVE maintains multiple active

connections to receive updates from the IoT devices, which causes it to consume more memory.

The CPU and memory tests also revealed that IoTivity is more resource-intensive than LwM2M. The latter presents a residual amount of CPU and memory usage in all load levels.

Comparing CPU and memory usage of NextGenGW with the IoT gateways presented in Section 3.3.1, one may conclude that NextGenGW is much more efficient (cf. Table 3.2). The most efficient solution in the literature uses at least 8,17% of CPU and 40.22MB of memory [3]. This comparison, however, needs to be done with care. Although aiming to solve the interoperability problem, each solution follows a completely different approach. They differ in terms of addressed use cases, their evaluation setup, supported protocols and frameworks, and the hardware they are running on. Nevertheless, the comparison is relevant to understand how NextGenGW performance relates to the solutions in the State-of-the-Art.

Figures 3.8 through 3.10 present NextGenGW's response time, measured as the time taken by NextGenGW to send requests to the Wakaama server (Figure 3.8) and IoTivity client (Figure 3.9), plus the time it takes to process the response from the IoT devices (Figure 3.10).

The time taken to send requests to the destination "Broker" increases with the number of IoT devices, and it is also higher for the READ request. This happens because the increase in the number of IoT devices results in a higher number of messages received in a short period. This is also the reason behind the higher processing times of the READ request. As specified in Section 3.3.2, the READ request targets the most complex object in the IoT devices, which is composed of several properties. The object that comes in response to the READ request must also be translated to SDF, which impacts the processing time of the READ requests.

Additionally, the results also show a high difference between the processing time for LwM2M and IoTivity requests, with the latter being much faster. This is mainly related to how the LwM2M translator was implemented, which resulted in a more time-consuming translation than IoTivity. The LwM2M translator uses the LwM2M data model files in XML format and C++ Vectors that hold some key information for the IoT devices connected to NextGenGW. This approach reduces the amount of memory usage, but it has the trade-off of higher processing time due to the searches that need to be performed both in the Vectors and in the XML data model files. The IoTivity translator follows a different trade-off, loading more information into memory. It uses C++ Maps to build dictionaries for translation. This reduces the number of file accesses since the information is accessed directly by using the correct key in the map, making the translation more efficient in processing time.

Regarding the time taken to process the responses received from the broker, we could not distinguish the processing time per broker because the measure is done on the MQTT side, which is not directly aware of which response corresponds to which broker. We could have implemented some logic to determine this, but that would result in a more intrusive time measurement strategy that we opted to avoid. The measured values also indicate a slight increase in the processing time along with the increase in load.

Considering the measured values, 75% of the requests are delivered to the LwM2M "Broker" between 500ms and 3.8s, depending on the load and type of request. IoTivity is between 400ms

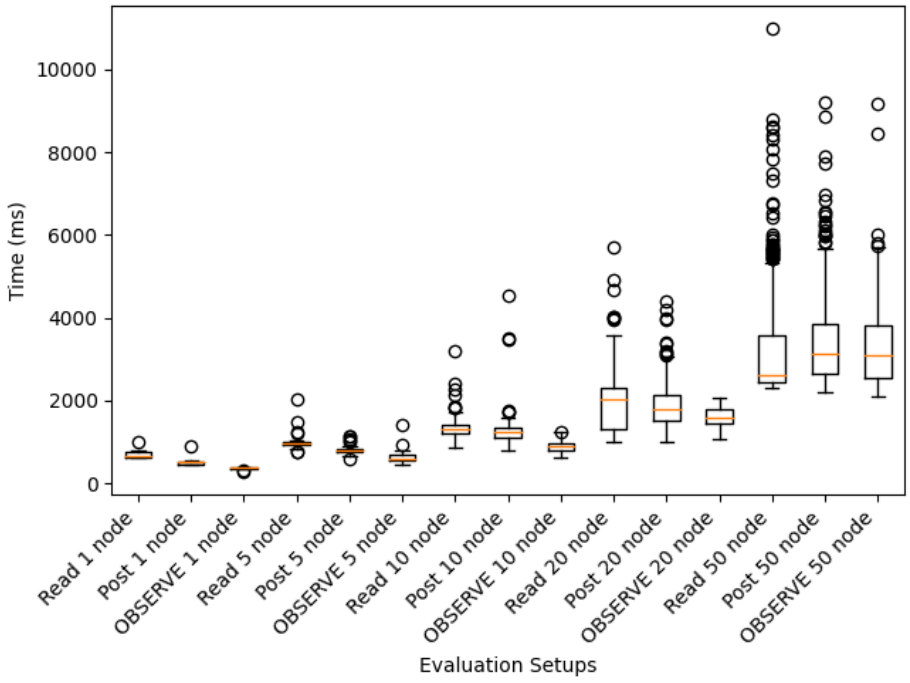


Figure 3.8: The time NextGenGW takes to send requests to the Wakaama server.

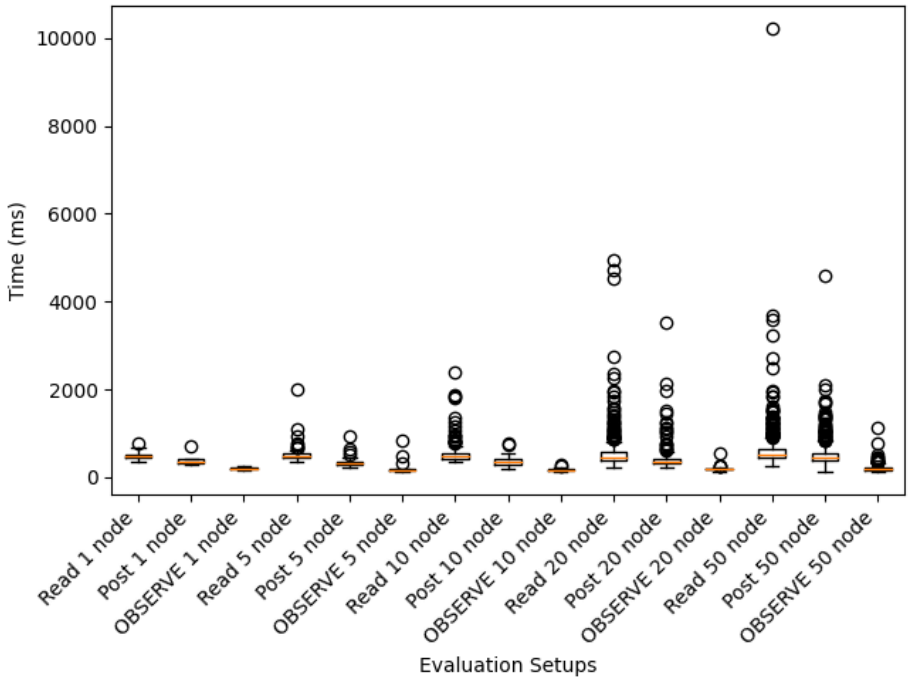


Figure 3.9: The time NextGenGW takes to send requests to the IoTivity client.

and 600ms. However, for both brokers, some outliers go up to 10s, probably related to the data model files access that is halted by higher-priority filesystem access from the operating system. This results in an average response time between 384.54ms to 3.65s. These values are suitable for slow processes, e.g., process control plants or in building automation. Additionally, we foresee that

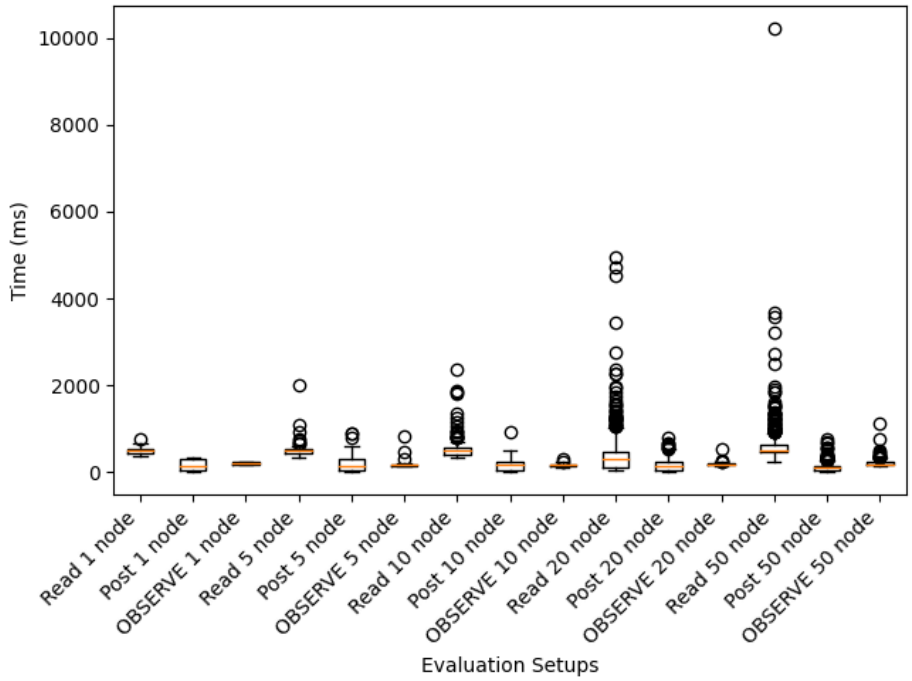


Figure 3.10: The time NextGenGW takes to process the responses.

NextGenGW's response time can be significantly reduced by optimising the LwM2M translator for processing speed, as it was done for IoTivity.

Comparing the measured response times with the solutions in Section 3.3.1, one may conclude that our solution consumes more time per request (cf. Table 3.2). The IoT middleware proposed in [111] has an average response time between 127 ms and 300 ms. Nevertheless, the comparison of the absolute values, as we highlighted in the CPU and memory usage, should consider the distinctive characteristics and evaluation procedures of each solution.

Regarding throughput, NextGenGW can handle 274 requests per second, which is above the throughput of the solutions presented in Section 3.3.1. In the literature, only the solution proposed in [64] measures throughput and can handle 62 messages per second (cf. Table 3.2).

3.4 Summary

In this chapter, we presented NextGenGW, our proposal to address the heterogeneous interaction and data models of IoT devices using IETF SDF. We started the chapter overviewing IETF SDF, and then detailed how we have used it in NextGenGW to homogenise the communication with IoT devices, abstracting their heterogeneity. We showed how we have designed NextGenGW to allow binding SDF with different communication protocols, and how we have used that strategy to bind it with MQTT. We have also detailed how NextGenGW's modular architecture allows for easily adding support to IoT devices with new data and interaction models through C++ shared libraries.

Table 3.2: NextGenGW performance compared with State-of-the-Art solutions.

Comparison Metrics	NextGenGW (%)	State-of-the-Art
CPU usage Range (%)	[0.1-6.3]	[8,17-23] [3]
Minimum Memory usage Range (MB)	[5.37-12.49]	[40.22-200.96] [3]
Response Time Range (ms)	[384.54-3650]	[127-300] [111]
Throughput (requests/s)	274	62 [64]

We closed the chapter with the NextGenGW quantitative evaluation, which demonstrated that it effectively handles the heterogeneous interaction and data models of IoT devices. This interoperability feature comes with a performance overhead in terms of response time and resource usage, which is dependent on the implementation of the brokers that communicate with the IoT devices and translate their interaction and data models to IETF SDF. We concluded that such brokers should be developed with the trade-off between resource usage and response time in mind. The LwM2M broker reported in this chapter minimises resource usage at the expense of a higher response time, which made NextGenGW suitable for slow processes, such as process control plants or building automation. We argued that the NextGenGW resource usage reported in this chapter is low, so modifying the LwM2M broker implementation to prioritise response time over resource usage would be a viable option to broaden the types of applications supported by NextGenGW.

Chapter 4

FITA: Orchestration of Heterogeneous IoT Devices

Section 2.1 showed that the current State-of-the-Art in IoT orchestration frameworks lacks the support for the heterogeneity of IoT devices, both in terms of interaction and data models, as well as runtime environments. The analysed literature shows that the majority of the IoT orchestration solutions only target Fog nodes and are dependent on the use of a certain lightweight virtualisation technology [25, 2, 23, 113, 84, 106]. Only one solution is capable of dealing with the heterogeneity of nodes, including their runtime environment, and supports the provisioning of IoT devices [31]. However, it does not address the vendor lock-in problem, since it uses proprietary communication technologies. Additionally, there is no indication of how to integrate such a solution with existing orchestration platforms, such as K8S, the de facto standard tool for Cloud and Edge orchestration. Such limitation hampers its utilisation in use cases that target the IoT continuum, i.e., from the Cloud to the IoT devices. CNCF projects such as Akri[1] and KubeEdge[2] have integrated IoT devices into K8S. However, they do not consider the possibility of deploying application components into such devices, disregarding their growing computing capabilities. Additionally, these CNCF projects do not follow a standards-based approach to address interoperability. Arrowhead and Margo target interoperability. However, Arrowhead focus on service function chaining, not application component deployment, and the current Margo specification just targets the Edge and disregards the IoT devices.

In this chapter, we propose the Far-edge IoT device mAnagement (FITA) [92], which integrates IoT devices, including resource-constrained ones, into K8S Cloud-native orchestration. The proposed extensions to K8S make use of the homogenisation brought by NextGenGW to deal with the heterogeneity of IoT devices. Section 4.1 overviews K8S and other relevant technologies to provide the basis for understanding FITA. Section 4.2 details FITA architecture and how it addresses the heterogeneous runtime environments, as well as data and interaction models of IoT

[1] https://docs.akri.sh
[2] https://kubeedge.io/

devices. Section 4.3 presents FITA evaluation methodology and results. Section 4.4 concludes this chapter.

The Far-edge prefix associated with FITA and some of its components comes from FITA's definition [92] and refers to the microcontroller-based devices that sense and act upon their environment, and are too resource-constrained to run a full-fledged operating system. Such devices are a subset of the devices of the IoT devices layer in the IoT stack (cf., Figure 1.1). In this Thesis, we expand the usage of FITA to all devices in the IoT devices layer, resource-constrained or not, that are not compliant with K8S and provide a runtime environment for application component deployment.

4.1 Background Technology and Terminology

This section provides a brief overview of K8S, focusing on the most relevant elements necessary for understanding FITA. K8S is a CNCF open-source platform, focused on automating the deployment, scaling, and management of containerised applications. It abstracts the underlying infrastructure, letting developers deploy and run applications reliably at scale. The most relevant components of the K8S Cluster for the scope of FITA are shown in Figure 4.1 and detailed below, along with other relevant K8S mechanisms:

- Pod[3] (cf. a) is the smallest deployable unit. It can be composed of a group of one or more containers that share storage and network resources.

- Deployments[4] (cf. b) manage pods by defining the desired number of replicas, handling rolling updates, and ensuring that the correct Pod templates are used to maintain application availability and consistency. While Pods are ephemeral, the K8S Deployment acts as a controller that continuously monitors Pods and recreates them to guarantee they match the declared configuration.

- Node[5] (cf. c) is a single physical or virtual machine that receives and runs Pods. There are two types of Nodes: 1) Control Plane Nodes, which run the K8S elements responsible for managing the cluster, such as the K8S API server, scheduler, etc.; 2) Worker Nodes that host the Kubelet, which is the K8S agent responsible for communicating with the K8S API server. The Kubelet integrates the Node in the K8S cluster, i.e., it registers the Node and keeps track of its status. The Kubelet also ensures that Containers in a Pod are running in the Node while also exposing telemetry regarding resource usage.

- Cluster[6] (cf. d) is a group of Nodes.

[3] https://kubernetes.io/docs/concepts/workloads/pods/
[4] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
[5] https://kubernetes.io/docs/concepts/architecture/nodes/
[6] https://kubernetes.io/docs/concepts/architecture/

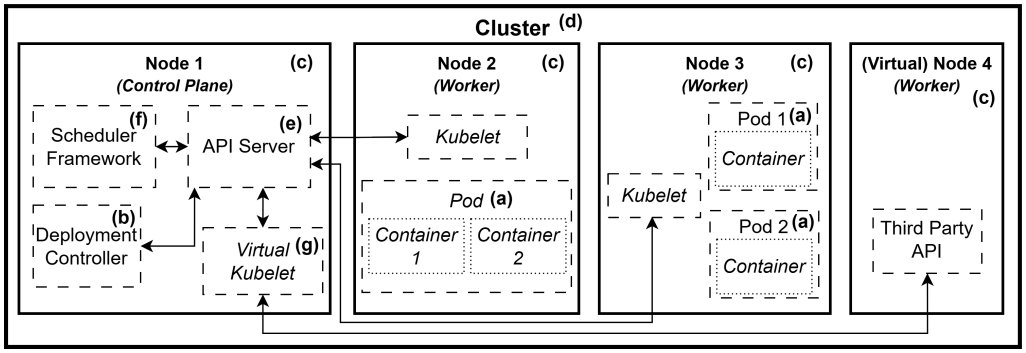


Figure 4.1: Summary of K8S Cluster components. For simplicity, the control plane and workloads are displayed running in different Nodes, but some K8S distributions (Minikube, microK8S, etc.) allow these components to coexist in the same Node.

- API Server[7] (cf. e) is the central control plane interface through which all the K8S operations pass.

- Scheduler[8] (cf. f) is the K8S scheduling mechanism. It defines in which Nodes the Pods are deployed, taking into consideration the Pods' requirements.

For the sake of simplicity, Figure 4.1 considers that the control plane and workloads are running in different Nodes (i.e., Control Plane and Worker), although some K8S distributions (Minikube[9], microK8S[10], etc.) run everything in a single Node.

Focusing on the Kubelet, K8S provides the Virtual Kubelet[11] (cf. g), which is an extension mechanism to integrate non-compliant devices into the cluster, allowing for their orchestration. The Virtual Kubelet is a programmable open-source Kubelet implementation that runs the Kubelet as a Pod in the K8S cluster and connects the K8S API to the APIs of non-compliant devices, such as those of third-party clusters from Cloud providers. When instantiated, the Virtual Kubelet creates a Node in the cluster (referred to as Virtual Node in the rest of this chapter). This Virtual Node is associated with the Virtual Kubelet and managed by it. The Virtual Node associated with the Virtual Kubelet is, from the K8S API server point of view, indistinguishable from a compliant K8S device running the K8S Kubelet agent.

The remainder of this chapter uses the K8S term *Pod* and the term *IoT application component* interchangeably to refer to the components that compose an IoT application, i.e., services in Service-Oriented Architecture terminology. We do not use the term *service* to avoid confusion with K8S' Services. We use the K8S term *Node* interchangeably with the term device to refer to the physical hardware where such components are deployed.

[7] https://kubernetes.io/docs/concepts/architecture/#kube-apiserver
[8] https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/
[9] https://minikube.sigs.k8s.io/docs/
[10] https://microk8s.io/
[11] https://virtual-kubelet.io/

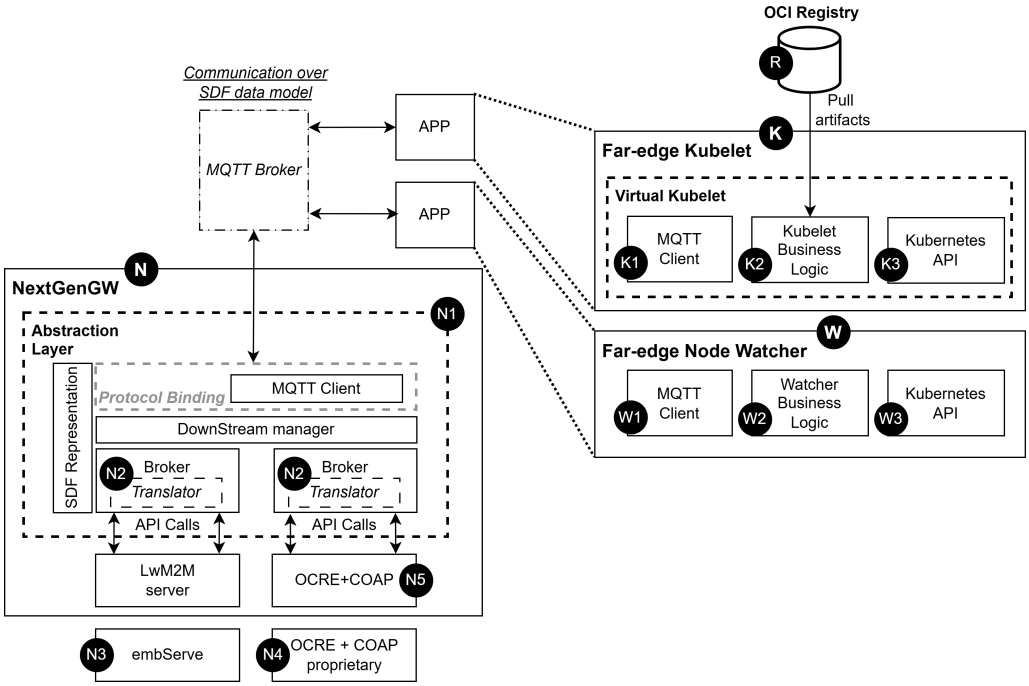


Figure 4.2: FITA high-level architecture showing the NextGenGW, Far-edge Kubelet and Far-edge Node Watcher components.

4.2 FITA

Figure 4.2 presents FITA architecture, which addresses two main issues in IoT orchestration:

- The deployment of application components on heterogeneous IoT devices. These devices may not use the traditional virtualisation technologies currently supported by Cloud-native orchestration solutions for application component deployment. The runtime environments of IoT devices can be based on scripting interpreters, virtual machines, and dynamic loadable components [96], which are exposed through different interaction and data models.
- The integration of heterogeneous IoT devices into K8S Cloud-native orchestration. K8S is an agent-based architecture, where each device runs the Kubelet agent, which is responsible for the interactions with the K8S API (Section 4.1). IoT devices that do not run the Kubelet are not compliant with K8S and cannot be part of its cluster.

We address these problems by combining the NextGenGW (cf. N), Far-edge Node Watcher (cf. W) and Far-edge Kubelet (cf. K), which are detailed in sections 4.2.2 to 4.2.4, respectively. To evaluate the interoperability regarding runtime environments, as well as data and interaction models, we have used two solutions provided by the community: (1) embServe [78] (cf. N3), which uses a dynamic loadable component runtime exposed through LwM2M; and (2) OCRE[12] (cf. N4), which is based on WebAssembly (Wasm) [94] virtual machine exposed by a protocol defined by us as explained in Section 4.2.1.

[12] https://lfedge.org/projects/ocre/

Table 4.1: OCRE API functions and their meaning

OCRE API Function	Meaning
ocre_container_runtime_create_container	Create an OCRE container.
ocre_container_runtime_run_container	Start an OCRE container.
ocre_container_runtime_stop_container	Stop an OCRE container.
ocre_container_runtime_restart_container	Restart an OCRE container.
ocre_container_runtime_get_container_status	Get the status of an OCRE container.
ocre_container_runtime_destroy_container	Destroy an OCRE container.

4.2.1 OCRE Devices Protocol Specification

OCRE is a Linux Foundation project that utilises Wasm virtual machines to provide a container-like experience for the development and management of software for resource-constrained devices that do not support traditional containerisation technologies. The project is focused on developing the runtime and provides a library with a set of functions to interact with it. Table 4.1 presents these functions, which allow the creation, start, stop, restart, destroy and get status of a container. OCRE does not provide a communication protocol to interact with such functions, so we developed a simple one to validate the capability of FITA to handle multiple runtime environments.

We exposed the OCRE API using CoAP. A CoAP server is listening on a single URI - "/ocre/-container/+" - for GET, POST, PUT and DELETE requests. The last part of the URI defines the ID of the container that is the subject of these actions. For example, *GET /ocre/container/<container-id>* sends a GET request for the container with ID <*container-id*>. The mapping between the CoAP requests and the OCRE API functions is the following:

- GET maps to the *ocre_container_runtime_get_container_status* function, i.e., it gets the status of the container specified in the URI.
- POST maps to the *ocre_container_runtime_create_container* function. The request payload contains the container to be instantiated in Base64 format.
- PUT maps to the functions *ocre_container_runtime_run_container*, *ocre_container_runtime_stop_container* and *ocre_container_runtime_restart_container*. The request payload specifies the function to execute as a string, namely "start", "stop", and "restart", respectively.
- DELETE maps to the *ocre_container_runtime_destroy_container* function.

In addition to the CoAP server, we also launch a CoAP client that announces itself to the CoAP server it was configured with. The objective is to inform other CoAP capable devices about the existence of the OCRE device. These CoAP capable devices, for example, the NextGenGW, instantiate a CoAP server and client to interface with devices hosting the OCRE runtime.

4.2.2 NextGenGW

NextGenGW, our proposed standards-based approach to address the heterogeneity of data and interaction models in IoT devices, is detailed in Chapter 3. In this section, we explain how we use

it within the scope of IoT orchestration, specifically in the deployment of application components, considering the heterogeneous runtime environments of IoT devices, as well as their data and interaction models.

We leverage NextGenGW Abstraction Layer (cf. N1, Figure 4.2), which simplifies communication with heterogeneous IoT devices by homogenising the representation of their properties and capabilities with SDF and streamlining their interaction using SDF bond to the MQTT protocol. This homogenisation allows the orchestration mechanisms to interact with IoT devices using a single data and interaction model. Considering the OneDM liaison group vision (the SDF proponents) of having a single data model for each IoT device class, we argue that we can have a single data and interaction model for the deployment of application components in IoT devices. Following our standards-based approach, we identified the LwM2M as, to the best of our knowledge, the only standard that proposes a mechanism for the deployment and management of application components in IoT devices. LwM2M Software Management technical specification [83] defines this mechanism, which we adopted and converted to SDF bound to MQTT.

Following this, NextGenGW exposes each IoT device application component deployment capability using the SDF model present in Listing 4.1, which currently supports a subset of LwM2M Software Management object properties fundamental for application component deployment. In the future, we will add support for the full object by including the non-critical properties.

Orchestration deployment mechanisms, such as the proposed Far-edge Kubelet (Section 4.2.3), utilise the proposed SDF model to deploy the application components in the IoT devices through NextGenGW, following four steps:

1. Instantiate a new software management endpoint by publishing the message:

```
{"operation": "POST",
 "data": "{"label":<instance_id>}"}
```

   In the topic:

```
<iot_device_id>/LWM2M_Software_Management
```

2. Load application component package by publishing the message:

```
{"operation": "POST",
 "data": <pkg_spec>}
```

   In the topic:

```
<iot_device_id>/LWM2M_Software_Management/<instance_id>/Property/Package
```

3. Install the application component by publishing the message:

```
{"operation": "POST"}
```

   In the topic:

```
<iot_device_id>/LWM2M_Software_Management/<instance_id>/Action/Install
```

Listing 4.1: NextGenGW SDF Software Management Model for IoT devices.

```
{
  "sdfObject": {
    "LWM2M_Software_Management": {
      "sdfProperty": {
        "PkgName": {
          "writable": false,
          "type": "string",
          "minLength": 0,
          "maxLength": 255
        },
        "PkgVersion": {
          "writable": false,
          "type": "string",
          "minLength": 0,
          "maxLength": 255
        },
        "Package": {
          "readable": false,
          "type": "string",
          "sdfType": "byte-string"
        },
        "Update_State": {
          "writable": false,
          "type": "integer",
          "minimum": 0,
          "maximum": 4
        },
        "Update_Result": {
          "writable": false,
          "type": "integer",
          "minimum": 0,
          "maximum": 200
        },
        "Activation_State": {
          "writable": false,
          "type": "boolean"
        },
        "Status_Reason": {
          "writable": false,
          "type": "string"
        }
      },
      "sdfAction": {
        "Install": { },
        "Uninstall": { },
        "Activate": { },
        "Deactivate": { }
      },
      "sdfRequired": [
        "#/sdfObject/LWM2M_Software_Management/sdfProperty/PkgName",
        "#/sdfObject/LWM2M_Software_Management/sdfProperty/PkgVersion",
        "#/sdfObject/LWM2M_Software_Management/sdfAction/Install",
        "#/sdfObject/LWM2M_Software_Management/sdfAction/Uninstall",
        "#/sdfObject/LWM2M_Software_Management/sdfProperty/Update_State",
        "#/sdfObject/LWM2M_Software_Management/sdfProperty/Update_Result",
        "#/sdfObject/LWM2M_Software_Management/sdfAction/Activate",
        "#/sdfObject/LWM2M_Software_Management/sdfAction/Deactivate",
        "#/sdfObject/LWM2M_Software_Management/sdfProperty/Activation_State"
      ]
    }
  }
}
```

4. Activate the application component by publishing the message:

```
{"operation": "POST"}
```

   In the topic:

```
<iot_device_id>/LWM2M_Software_Management/<instance_id>/Action/Activate
```

Note that <instance_id> is the ID of the instance to be created/manipulated, <iot_device_id> is the ID of the IoT device to interface with and <pkg_spec> is the application component package to install in the IoT device.

In the scope of this Thesis, we used such an approach to deploy application components in IoT devices that host embServe runtime, which uses LwM2M as the communication protocol (cf. N3, Figure 4.2), and in IoT devices that host OCRE runtime (cf. N4) and use the protocol specified in Section 4.2.1. With this setup, we validate the proposed approach regarding both runtime heterogeneity (embServe utilises dynamic loadable components, and OCRE employs a Wasm virtual machine) and the data and interaction model used to expose it.

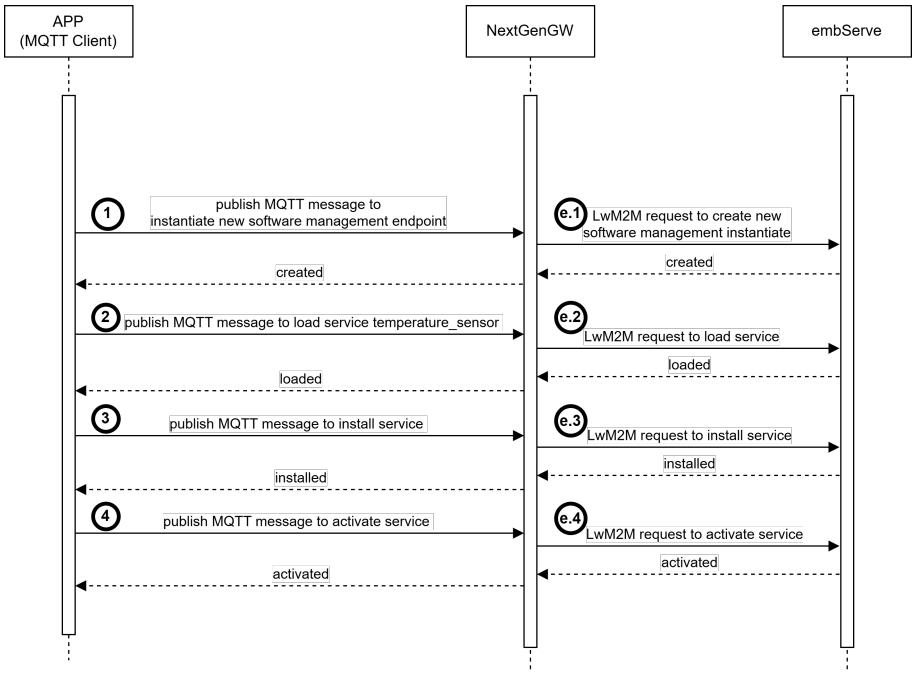
Figure 4.3 presents the sequence diagrams for the deployment of application components on embServe (Figure 4.3a) and OCRE (Figure 4.3b) devices using NextGenGW. The interactions labelled with 1 to 4 in both figures represent the four steps of the deployment procedure detailed above.

Focusing on embServe, all these interactions are translated into LwM2M and correspond to a request to the IoT device, i.e., there is a one-to-one correspondence between the requests received by the NextGenGW and the requests sent to the IoT devices running embServe (cf. e1 to e4). This one-to-one mapping occurs because embServe exposes its runtime environment application component deployment mechanism using LwM2M Software Management object, which is also the object we incorporated in the SDF to expose the application component deployment capabilities of IoT devices.

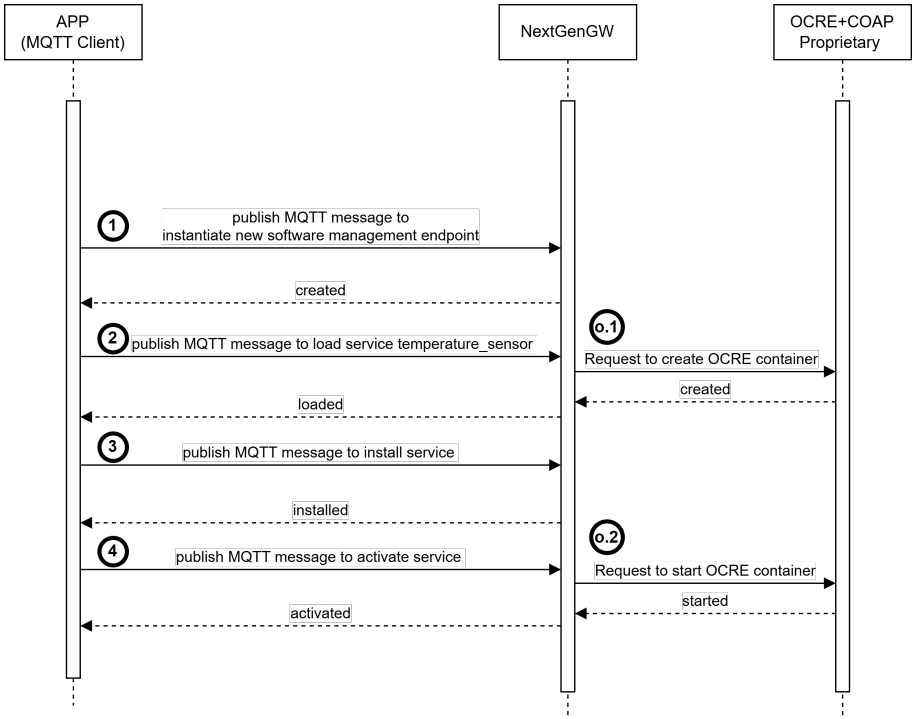
OCRE uses a simpler protocol (Section 4.2.1) that the sequence diagram of Figure 4.3b illustrates. Only two requests received by NextGenGW correspond to a CoAP request sent to the IoT devices running OCRE (cf. o1 and o2, respectively). The OCRE+COAP instance running in NextGenGW (cf. N5) handles the other requests by creating internal structures that help it manage the containers and returns as soon as such internal structures are created.

In addition to this adaptation of the messages transmitted to the IoT devices, the payload used in the second request sent to the NextGenGW differs depending on the runtime. For embServe, the <pkg_spec> contains the embServe binary blob. For OCRE, it contains the OCRE container.

Support for other runtime environments, as well as the data and interaction models used to expose its application component deployment mechanisms, can be added by following an approach similar to the one we applied with embServe and OCRE. The user implements translators (cf. N2) between the deployment procedure detailed in this section and the deployment procedure of the IoT devices the user wants to integrate with the NextGenGW. Then, in the request to load a new application component (step with label 2), the <pkg_spec> is created based on the package specification supported by the runtime of the new device.



(a) Sequence diagram for deploying an application component on devices running embServe runtime.



(b) Sequence diagram for deploying an application component on devices running OCRE with a COAP proprietary data and interaction model.

Figure 4.3: Sequence diagram for deploying an application component on IoT devices using NextGenGW.

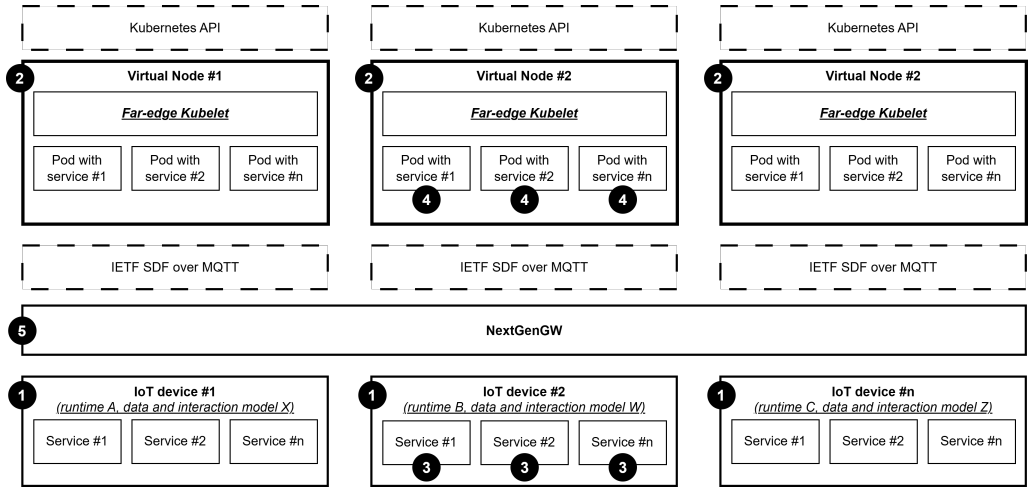


Figure 4.4: Details on the Far-edge Kubelet and its interactions with K8S and NextGenGW.

4.2.3 Far-edge Kubelet

The Far-edge Kubelet (cf. K, Figure 4.2) is responsible for hosting in the K8S cluster the IoT devices that do not run the Kubelet agent due to, for example, their reduced resources. It does so by building on top of the K8S Virtual Kubelet (Section 4.1), leveraging the abstraction and simplification it offers for implementing a custom Kubelet agent. Although developed for third-party Cloud clusters, the Virtual Kubelet fits our purpose because, from our perspective, a network of IoT devices is a third-party cluster that should be connected to the K8S cluster.

Each Far-edge Kubelet manages one IoT device, which it represents in the K8S cluster using a Virtual Node. The Far-edge Kubelet integrates the IoT device it is associated with in the K8S cluster by registering it in the K8S API. The Far-edge Kubelet also bridges the K8S API with the NextGenGW to allow K8S to deploy application components on the IoT device associated with the Far-edge Kubelet following the procedure detailed in Section 4.2.2.

Figure 4.4 summarises this relation between IoT devices, Far-edge Kubelet and NextGenGW. Each IoT device (cf. 1) has its own Virtual Node, which the Far-edge Kubelet manages (cf. 2). The list of application components running on the IoT device (cf. 3) also has its virtual counterpart in the Virtual Node. These are the Pods (cf. 4) that the K8S scheduler assigned to the Virtual Node. The NextGenGW (cf. 5) serves as a bridge between the different IoT devices, which have different runtimes, and data and interaction models. The NextGenGW abstracts this heterogeneity, allowing the Far-edge Kubelet to use a single data and interaction model (as explained in Section 4.2.2) independently of the one provided by the IoT devices. Using such homogenisation allows us to have a single Far-edge Kubelet image that is instantiated for each IoT device, independently of its runtime, as well as data and interaction models.

The IoT device CPU, memory, architecture, operating system, runtime, sensors and actuators characteristics are set in the Far-edge Kubelet at boot time using environmental variables, which it digitally represents in the Virtual Node. This digital representation uses the K8S Node's *CPU*, *Memory*, *Architecture*, and *Labels* properties. We use the Node *Labels* to set the runtime, sensors

Listing 4.2: Virtual Node YAML representation. We omitted some details for simplicity.

```
1  apiVersion: v1
2  kind: Node
3  metadata:
4    labels:
5      alpha.service-controller.kubernetes.io/exclude-balancer: "true"
6      extra.resources.fhp/accelerometer: "true"
7      extra.resources.fhp/gyroscope: "true"
8      extra.resources.fhp/magnetometer: "true"
9      extra.resources.fhp/temperature: "true"
10     extra.resources.fhp/runtime: "embserve"
11     kubernetes.io/hostname: iot-device1
12     kubernetes.io/role: agent
13     node.kubernetes.io/exclude-from-external-load-balancers: "true"
14     type: virtual-kubelet
15   name: iot-device1
16 status:
17   allocatable:
18     cpu: "1"
19     memory: 256k
20     pods: "5"
21   capacity:
22     cpu: "1"
23     memory: 256k
24     pods: "5"
25   nodeInfo:
26     architecture: arm-v7
27     kubeletVersion: v1.15.2-vk-3756060-dev
28     operatingSystem: zephyr
29   phase: Running
```

and actuators. Following the example provided in Listing 4.2, the Node *iot-device1* (cf. line 15) has *one* CPU (cf. line 22) whose architecture is *arm-v7* (cf. line 26), *256kB* of memory (cf. line 23), runs the operating system *zephyr* (cf. line 28) along with the *embServe* runtime (cf. line 10), and supports the deployment of *five* Pods (cf. line 24). It is composed of four sensors, namely *accelerometer*, *gyroscope*, *magnetometer* and *temperature* (cf. lines 6 to 9). Using labels to set the runtime, sensors and actuators, although not following a standards-based approach, already allows application architects to control where to deploy application components that depend on them. The K8S scheduler can use this information to ensure it schedules each Pod to the Virtual Node with the correct capabilities, which is fundamental for optimised IoT applications. Our future work (Section 6.2) includes a set of research directions on how to improve this representation using standards.

Independently of the current non-standard or future standards-based representation, the Far-edge Kubelet serves all the requests sent to the Virtual Node by mimicking the interactions that a real Node performs using its Kubelet. The Far-edge Kubelet Business Logic (cf. K2, Figure 4.2) is the element responsible for this. It bridges the K8S API (cf. K3) with the MQTT Client (cf. K1) that interacts with the NextGenGW. The Far-edge Kubelet Business Logic implements all the necessary mechanisms for Pod deployment in the IoT device, translating it into the application component deployment procedure detailed in section 4.2.2. In our solution, we map Pod containers to the application components deployed by NextGenGW. To standardise the application

Listing 4.3: Temperature Application Component Deployment YAML representation.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: temperature-deployment
5    labels:
6      app: temperature
7  spec:
8    replicas: 2
9    selector:
10     matchLabels:
11       app: temperature
12   template:
13     metadata:
14       labels:
15         app: temperature
16     spec:
17       containers:
18       - name: temperature
19         image: fhp/temperature_sensor:0.0.1
20         imagePullPolicy: IfNotPresent
21       nodeSelector:
22         extra.resources.fhp/runtime: "embserve"
23         extra.resources.fhp/temperature: "true"
```

component packaging, we use the OCI mechanisms[13], namely the OCI artifacts. The application components of each IoT device runtime are packaged as an OCI artifact and stored in an OCI compliant registry (cf. R). The Far-edge Kubelet pulls the artifact from the registry and uses it as <pkg_spec> in the communication with NextGenGW (refer to Section 4.2.2).

To better show the integration enabled by our solution, we describe a step-by-step explanation of the example K8S Deployment in Listing 4.3. This K8S Deployment requests the deployment of two replicas of a Pod containing the Container *temperature* (cf. 17) on a Node with a temperature sensor (cf. 23) and running *embServe* (cf. 22). Figure 4.5 presents the sequence diagram for such a K8S Deployment. The deployment procedure starts with the user requesting the K8S Deployment using K8S tools (step 1). The K8S scheduler analyses the Deployment and defines the Node with the labels *extra.resources.fhp/runtime: "embserve"* and *extra.resources.fhp/temperature: "true"* to which such deployment is bound. The selected Node is a Virtual Node associated with a Far-edge Kubelet. The K8S API sends the create Pod command to that Node (step 2). The Far-edge Kubelet that receives such a request downloads the image *fhp/temperature_sensor:0.0.1* from the OCI image repository (step 3). This image is associated with the *temperature* container that composes the Pod, but in fact, is a binary blob compatible with the embServe runtime. When the image is retrieved, the Far-edge Kubelet starts interacting with NextGenGW, following the steps outlined in Figure 4.3a, which abstracts the data and interaction models heterogeneity of IoT devices in application component deployment. The sequence for applying the *temperature-deployment* on the IoT device concludes with the Far-edge Kubelet informing the K8S API that the Pod has been created and is running (step 4). For other runtimes, such as the OCRE instantiation detailed in Section 4.2.1, the Far-edge Kubelet follows the same procedure, but the pulled OCI artifact is an

[13] https://opencontainers.org/

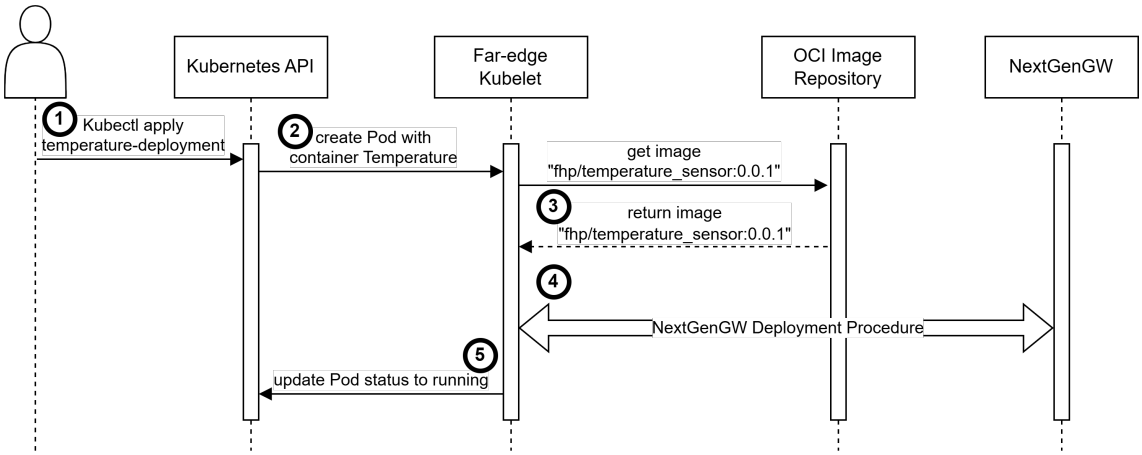


Figure 4.5: Sequence diagram for applying the Temperature Deployment using FITA.

OCRE container. On the NextGenGW side, the interaction with the Far-edge Kubelet is always the same. What differs is its interaction with the IoT device. For the OCRE instantiation, it uses the procedure of Figure 4.3b.

This simple example highlights two important characteristics of the proposed solution:

1. The integration of IoT devices with different runtime environments, and data and interaction models does not require any modification at the Far-edge Kubelet. One just needs to create the respective NextGenGW server and translator

2. The enablement of IoT continuum orchestration provided by combining IoT deployments, such as the one depicted in Listing 4.3, with deployments targeting standard K8S Nodes.

4.2.4 Far-edge Node Watcher

The Far-edge Node Watcher (cf., W in Figure 4.2) is responsible for creating and deleting the Far-edge Kubelet when the corresponding IoT device is registered in or unregistered from the NextGenGW, respectively. It is a containerised Python application that runs as a Pod in the K8S cluster. It uses an MQTT Client (cf. W1) to subscribe to the *announce* and *unregister* topics provided by NextGenGW. Through these topics, it receives up-to-date information when devices are added or removed from the IoT networks covered by the NextGenGW. For each message received in these topics, it utilises the K8S API (cf. W3) to request the creation or deletion of Far-edge Kubelet Pods. The Far-edge Node Watcher Business Logic (cf. W2) is the element responsible for guaranteeing the correct creation and deletion of such Pods. It parses the message received by the MQTT client on the announce topic, retrieving the new device CPU, memory, architecture, operating system, runtime, sensor, and actuator characteristics, and configuring them in the environmental variable settings of the Far-edge Kubelet's Pod creation request. On the opposite direction, the Far-edge Node Watcher Business Logic gets the ID of the unregistered IoT device from the message received in the unregister topic and deletes the Far-edge Kubelet hosting such a device.

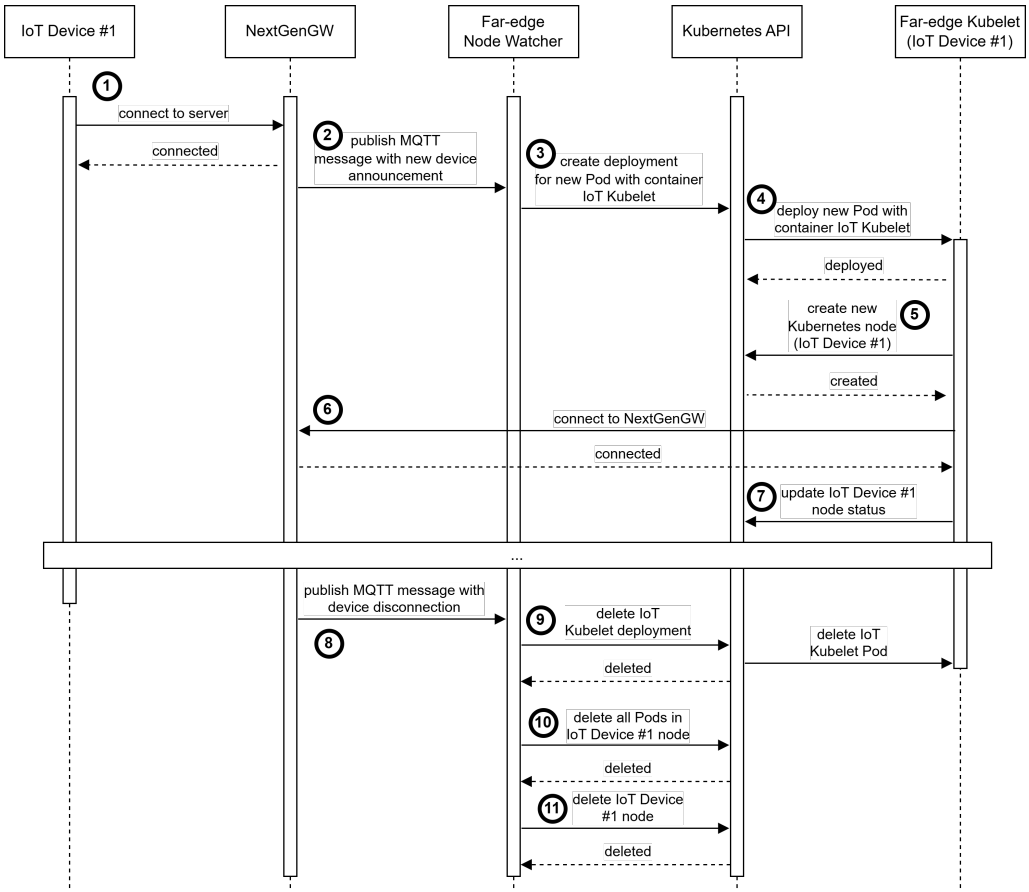


Figure 4.6: Sequence diagram for IoT device connection and disconnection in FITA.

The sequence diagrams in Figure 4.6 detail the flow of information for adding and deleting IoT devices from the IoT cluster, considering both the creation of the Far-edge Kubelet by the Far-edge Node Watcher and the creation of the Virtual Node by the Far-edge Kubelet.

The inclusion of IoT devices starts when a device registers in one of the servers running with NextGenGW (step 1) (in the scope of this Thesis, it might be an LwM2M server or a CoAP server, depending on whether the devices are running embServe or OCRE, respectively). NextGenGW interprets the messages of the registration procedure, translates the device information into IETF SDF, and publishes it on the MQTT topic *announce* (step 2). The Far-edge Node Watcher receives this information, interprets the IoT device information coded as an SDF Thing and creates the K8S Pod composed of a single container with the Far-edge Kubelet, setting its environmental variables according to the IoT device characteristics (step 3). Listing 4.4 shows an example of such K8S Pod. Afterwards, the K8S API deploys this Pod in the Node defined by the scheduler, starting the Far-edge Kubelet for the new IoT device (step 4). The Far-edge Kubelet creates the Virtual Node associated with it by requesting the K8S API to create a K8S Node with the characteristics and labels the Far-edge Node Watcher defined in the Far-edge Kubelet environmental variables (step 5). It also collects the IoT device status using the NextGenGW Abstraction Layer (step 6). Finally, the Far-edge Kubelet updates the Virtual Node information using the K8S API (step 7).

Listing 4.4: Far-edge Kubelet Pod YAML representation.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: far-edge-kubelet-device1
spec:
  containers:
  - name: far-edge-kubelet
    image: fhp/far-edge-kubelet:1.0.0
    imagePullPolicy: Never
    command:
    - "/far-edge-kubelet"
    args:
    - "--provider"
    - fhpAicos
    - "--nodename"
    - iot-device1
    - "--os"
    - zephyr
    env:
    - name: KUBELET_PORT
      value: '10250'
    - name: VKUBELET_POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    - name: MQTT_BROKER_URI
      value: aicos-ioto-services
    - name: MQTT_BROKER_PORT
      value: '1883'
    - name: NODE_ID
      value: iot-device1
    - name: REMOTE_REGISTRY
      value: registry-1.docker.io
    - name: LOCAL_REGISTRY
      value: "/var/tmp/vk-embserve"
    - name: NODE_CPU_CAP
      value: '1'
    - name: NODE_MEM_CAP
      value: 256k
    - name: NODE_POD_CAP
      value: '5'
    - name: NODE_ARCH
      value: arm-v7
    - name: NODE_CAPS
      value: "accelerometer;gyroscope;magnetometer;temperature"
    - name: NODE_RUNTIME
      value: embserve
  nodeName: labnuc01
```

The deletion of an IoT device from the K8S cluster begins when it disconnects from the NextGenGW server. NextGenGW takes the ID of such a device and publishes it in the *unregister* topic through its MQTT client (step 8). In the case of the LwM2M and CoAP servers, the IoT devices are removed from the K8S cluster if they fail a keep-alive message during the lifetime interval. When the Far-edge Node Watcher receives the message in the *unregister* topic, it deletes the Far-edge Kubelet Pod associated with that device (step 9), followed by all the Pods hosted in it (step 10) and the Virtual Node itself (step 11).

4.3 Evaluation

FITA uniqueness in integrating IoT devices in Cloud orchestration, considering the heterogeneous data and interaction model of IoT devices, as well as runtime environments, hampers its comparison with the State-of-the-Art solutions reviewed in Section 2.1. To the best of our knowledge, FITA is the first solution to truly enable IoT continuum orchestration by considering IoT devices as legitimate resources for application component deployment. Considering this, we evaluated FITA by characterising its overhead, performance, and scalability under different cluster loads, and comparing it with the different baselines described in Section 4.3.1.1, which aim to highlight the trade-offs of: (1) application component deployment on IoT devices using K8S Cloud orchestration tools; and (2) homogenising, within the orchestration context, the heterogeneous data and interaction models, as well as runtime environments of IoT devices, using NextGenGW. This evaluation aims to assess which deployment scenarios are FITA compliant with.

Section 4.3.1 details the evaluation setup, including the baseline scenarios used for comparison, and the software and hardware setup used during the tests. Section 4.3.2 presents the evaluated FITA functionalities, the cluster loads used for stressing them, and discusses the obtained results.

4.3.1 Experimental Setup

The setup used for FITA evaluation is composed of the six evaluation solutions described in Section 4.3.1.1, which comprise FITA and the baselines against which we compare. Section 4.3.1.2 details the hardware and software setup used for such evaluation.

4.3.1.1 Evaluated Solutions

We have prepared and evaluated six solutions to assess the overhead and trade-offs of FITA in using K8S to deploy application components on IoT devices, as well as in homogenising the heterogeneous data and interaction models, and runtime environments of IoT devices. Figure 4.7 illustrates these solutions, which are detailed below. To clearly distinguish between general technologies and the specific solution names, we italicise the names of the analysed solutions throughout the remainder of this section.

Leshan is based on the Leshan LwM2M Server (v2.0) and implements a server-centric approach where IoT deployments are executed using the provided REST API.

NextGenGW w/ LwM2M and ***NextGenGW w/ OCRE*** are based on the NextGenGW framework, with deployments executed using the SDF-based protocol over MQTT. The former solution focuses on LwM2M devices, while the latter focuses on OCRE devices.

K8S w/Leshan integrates Leshan with K8S by leveraging a modified version of the proposed Far-edge Kubelet and Far-edge Node Watcher, which communicate with the Leshan REST API instead of the NextGenGW SDF-based API.

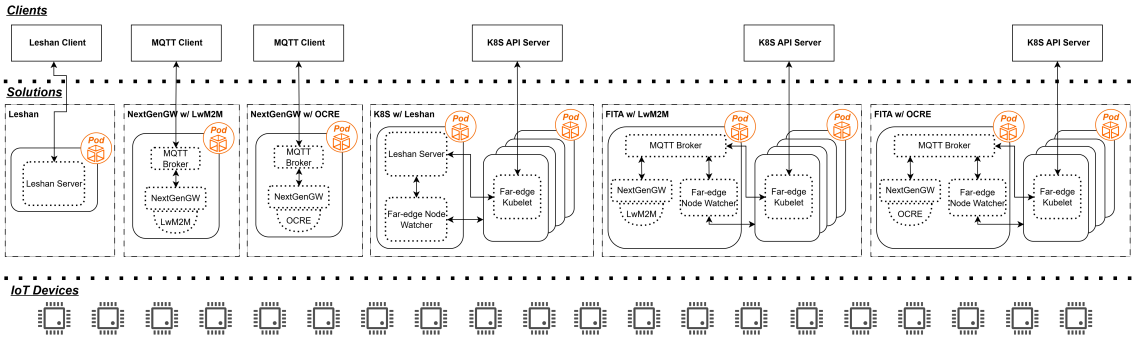


Figure 4.7: Solutions considered for testing and evaluation.

FITA w/ LwM2M and ***FITA w/ OCRE*** represent our proposed solution for IoT device orchestration. The former solution focuses on LwM2M devices running embServe, while the latter focuses on OCRE devices.

These solutions can be categorised according to two criteria: (1) the usage or not of cloud orchestration tools (*K8S w/ Leshan*, *FITA w/ LwM2M*, and *FITA w/ OCRE* versus *Leshan*, *NextGenGW w/ LwM2M*, and *NextGenGW w/ OCRE*, respectively); (2) the support or not for the heterogeneous data and interaction models, as well as runtime environments of IoT devices (*NextGenGW w/ LwM2M*, *NextGenGW w/ OCRE*, *FITA w/ LwM2M*, and *FITA w/ OCRE* versus *Leshan* and *K8S w/ Leshan*, respectively). We have not evaluated NextGenGW-based solutions with LwM2M and OCRE devices simultaneously. We have already evaluated NextGenGW simultaneous interaction with heterogeneous IoT devices in Section 4.3. In the FITA evaluation, we aimed to directly compare solutions based on NextGenGW with those based on Leshan to assess the homogenisation overhead in the orchestration context. Additionally, we wanted to assess the impact of the runtime on FITA performance.

Considering this, comparing *Leshan* against *NextGenGW w/ LwM2M* provides insights into the differences between a solution that supports a single protocol (Leshan) and a solution designed for protocol interoperability (NextGenGW) in the context of IoT orchestration. Comparing *Leshan w/K8S*, *FITA w/ LwM2M*, and *FITA w/ OCRE* against the other three solutions allows us to understand the performance of K8S-based IoT orchestration. Comparing *Leshan w/K8S* against *FITA w/ LwM2M* helps determine if the insights gained from comparing *Leshan* and *NextGenGW w/ LwM2M* are relevant in K8S-based IoT orchestration. Finally, comparing *FITA w/ LwM2M* against *FITA w/ OCRE* allows us to understand the impact of the runtime on the orchestration tasks. In conclusion, the performance results obtained from each solution enable us to understand the trade-offs of adding interoperability and orchestration features to an IoT installation, allowing for an informed analysis of the benefits and overheads of using each solution in specific use cases.

During the tests, we deploy each solution as a Pod in the cluster, with the Leshan Server and NextGenGW exposed as K8S Services to connect the IoT devices.

As a final note, we would like to highlight that, following the conclusions of NextGenGW evaluation in Section 3.3.3, we have used in FITA's evaluation an updated version of NextGenGW's LwM2M Broker that improved the Broker response time at the expense of an increase in memory

usage.

4.3.1.2 Hardware and Software

In terms of hardware, the experimental setup is composed of a mini-PC and a host computer. The mini-PC is an Intel NUC 13 Pro equipped with an Intel Core i3-1315U CPU, 16GB of RAM, and a 256GB NVMe SSD. The host computer features an Intel Core i7-8700K CPU, 32GB of RAM, and a 256GB NVMe SSD. Both machines are running Ubuntu 22.04 LTS and are connected to the same local network via Ethernet running at 100 Mbps.

The mini-PC hosts the K8S distribution, namely microk8s v1.32.9 [10], along with the solutions detailed in section 4.3.1.1. The host computer emulates the IoT devices for each test. LwM2M devices are emulated with IoTNetEmu [79], which is running embServe. OCRE devices are emulated with OCRE *native_sim* simulator[14]. The host computer is also running our private container registry, where we store all the OCI artifacts used during the tests. The registry uses the official Docker *registry* v3 image.

We would like to emphasise that our setup hosts the Pods that comprise the solutions in Section 4.3.1.1 on the Control Plane Node (the mini-PC). While some use cases may place FITA Pods in one or more worker Nodes, this setup is more convenient for our experiments and provides viable insights into how to distribute FITA Pods according to the use case requirements and the Nodes in the cluster. Our results also do not reflect network-induced delays. Our focus is on assessing FITA's performance independently of the use case and its associated network setup. With knowledge of which use cases are FITA compliant, the next steps would be to test FITA under such use cases, considering the network setup for each of them.

4.3.2 Experimental Results

FITA's evaluation focused on its three core functionalities, namely: (1) application component deployment (Section 4.3.2.1), which assess the time it takes to deploy an application component in IoT devices; (2) application component recovery (Section 4.3.2.2), which assess the time it takes to migrate a Pod between different Nodes of the cluster; (3) device registration (Section 4.3.2.3), i.e., the time it takes to register a Node in the cluster.

We stress these functionalities under different cluster loads by varying the number of Nodes in the cluster (10, 50, and 100) and the number of Pods per Node (0, 1, and 5). We have also performed FITA's resource analysis (Section 4.3.2.4) by evaluating the memory and CPU usage of each evaluated solution under such loads.

4.3.2.1 Application Component Deployment

The application component deployment test aims to characterise FITA's deployment ability. We set up each cluster load under test and measure the deployment duration for a single application

[14] https://docs.project-ocre.org/quickstart/firmware/simulated/

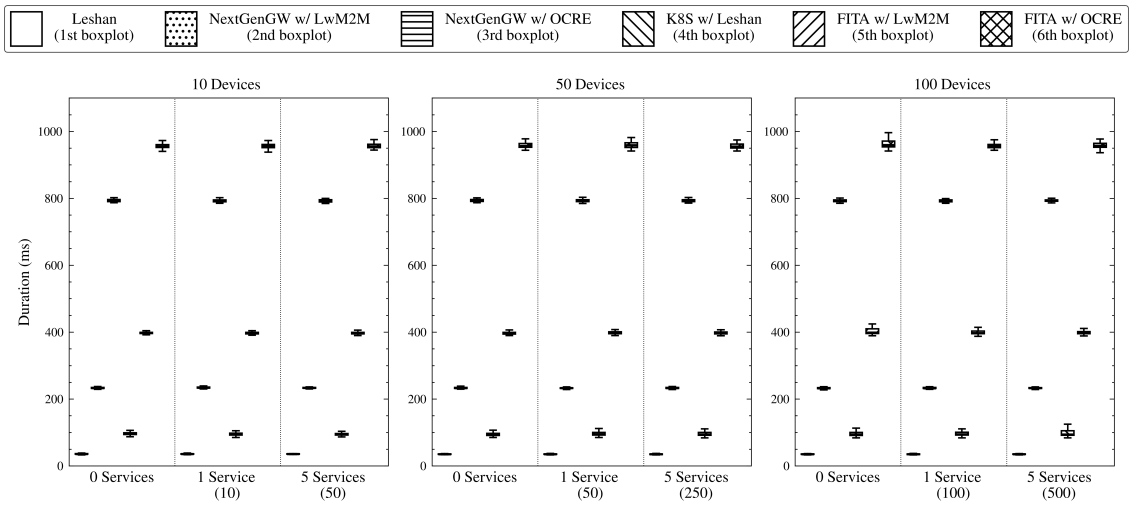


Figure 4.8: The time taken to deploy one Pod/application component in the solutions under analysis, considering a varying number of devices and deployed Pods/application components in the cluster.

component after fully configuring the cluster load. For the K8S based solutions (*K8S w/Leshan*, *FITA w/ LwM2M* and *FITA w/ OCRE*), we measure the time from the moment a Pod creation request is submitted to the K8S API server until the moment the K8S API server marks that Pod as running. For the other solutions, we measure from the moment the first command of the deployment sequence (see Section 4.2.2) is sent until the moment the platforms (Leshan or NextGenGW) respond positively to the last command of the deployment sequence. Leshan uses HTTP for such communications, while NextGenGW uses MQTT.

The application components used during the test are packaged as OCI artifacts (see Section 4.2) and stored in the local registry (see Section 4.3.1.2). Their size is approximately 2.6 KB for embServe devices, while for OCRE devices, it is approximately 5.5 KB, which is OCRE's minimum container size. We repeated each test 100 times.

Figure 4.8 shows the deployment durations for the six solutions under the specified cluster load variations. Results show that deployment duration is independent of the cluster load for all the solutions under analysis. Such constant deployment duration means that, if the deployment APIs are not stressed with multiple deployment requests (we are evaluating the deployment of a single application component), the number of Nodes and Pods in the cluster does not influence the deployment duration. Therefore, all the solutions demonstrate similar scalability as the number of Nodes and application components in the cluster increases.

When comparing Leshan-based solutions (*Leshan* and *K8S w/Leshan*) with NextGenGW-based ones connected to LwM2M devices (*NextGenGW w/ LwM2M* and *FITA w/ LwM2M*), we can assess that the former are consistently faster than the latter. Such a difference comes from the overhead introduced by NextGenGW for interoperability via SDF translation. This overhead, when considering median values, is around 197 ms when comparing *Leshan* against *NextGenGW w/ LwM2M* and around 301 ms when comparing *K8S w/Leshan* against *FITA w/ LwM2M*. The

additional overhead introduced by interoperability, although significant, may not be relevant to the scope of application component deployment. The median application deployment duration with *NextGenGW w/ LwM2M* is approximately 233 ms, and for *FITA w/ LwM2M*, it is approximately 398 ms, which are acceptable values for deployment tasks, as these are typically not time-constrained. Nevertheless, in time-constrained scenarios where deployment needs to be performed below the durations measured for the NextGenGW-based solution, the interoperability feature brought by NextGenGW might not be applicable due to its timing overhead.

Considering the scenarios with OCRE-based devices (*NextGenGW w/ OCRE* and *FITA w/ OCRE*), we can observe that the runtime has a significant influence on the deployment duration. Although the developed OCRE API requires fewer communications to be sent to the physical devices when compared to embServe (NextGenGW only sends two commands to OCRE devices versus four to embServe devices, see Figure 4.3), the minimum size of its deployment binary is approximately 2.1 times bigger than the embServe binaries (2.6 KB for embServe vs. 5.5 KB for OCRE). The larger binary size introduces communication overhead that exceeds the impact of the reduced number of commands required by OCRE devices for application component deployment. OCRE deployment duration takes approximately 560 ms more when compared to embServe, both when using and not using orchestration (*NextGenGW w/ LwM2M* vs. *NextGenGW w/ OCRE* and *FITA w/ LwM2M* vs. *FITA w/ OCRE*, respectively).

4.3.2.2 Aplication Component Recovery

In this test, we evaluated the K8S recovery mechanism for application components deployed on IoT devices by comparing the duration needed to redeploy a failed component using the solutions *K8S w/Leshan*, *FITA w/ LwM2M* and *FITA w/ OCRE*. As with the other tests, we varied the number of devices in the cluster. However, we only considered the deployment of one application component, the one being recovered.

We set up the test by creating a K8S Deployment composed of a single Pod that contains an application component equal to the ones used in Section 4.3.2.1. During the test, we check the Virtual Node to which the scheduler assigned this Pod, and send a request to the K8S API to mark it as unschedulable and to evict the Pod from that Node, starting the recovery and measurement procedure. The measurement is made from the moment the request to evict the Pod is submitted to the K8S API server, until the moment the Pod is removed from the Node, deployed in another one and marked as running by the K8S API server.

Figure 4.9 presents the measured values, which are of the same order of magnitude as the Pod deployment duration reported in Section 4.3.2.1. *K8S w/Leshan* is still faster than *FITA w/ LwM2M*, with a difference of approximately 200 ms independently of the number of Nodes in the cluster, when considering median values. The difference between *FITA w/ LwM2M* and *FITA w/ OCRE* is 620 ms if we consider the median values, and it is also independent of the number of Nodes in the cluster. The order of magnitude of the median recovery duration and the differences between the evaluated solutions show that the deployment duration mainly determines the

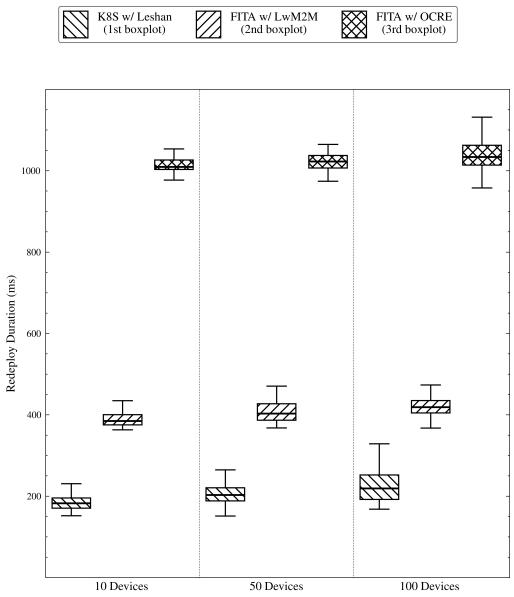


Figure 4.9: The time taken to redeploy an application component when a device fails in the three K8S-based solutions with a varying number of devices in the cluster.

recovery duration. However, the K8S Pod eviction mechanism also contributes to the total recovery duration. This contribution is visible in the increase of the recovery median value with the increase in the number of Nodes in the cluster for all solutions, as well as in the increase of its interquartile range. Such variability is not present in the Pod deployment results.

Overall, FITA can complete application component migration in clusters of 100 embServe devices in under 450 ms and around 1000 ms for setups with 100 OCRE devices. This is a period of application component unavailability that determines the applicability of FITA. In use cases that are not compliant with such recovery durations, one could consider implementing mechanisms to manage redundant replicas of critical application components. Thus, removing their deployment duration from the total recovery duration, which is the main contributor to recovery in our evaluation. In stateful application components, one would also need to explore how to use (semi-)active or (semi-)passive replication mechanisms in K8S to guarantee the consistency between replicas. This topic is beyond the scope of our Thesis.

4.3.2.3 Device Registration

This test assesses FITA's registration procedure, detailed in Section 4.2.4, by measuring for all evaluated solutions the time needed to add a new device to the cluster. For the K8S based solutions (*K8S w/Leshan*, *FITA w/ LwM2M* and *FITA w/ OCRE*), we measured that time lapse by considering the moment when the device starts until the moment the K8S API server marks it as ready. For *Leshan*, we measured the duration from the moment the device starts until the moment the Leshan server sends the registration message indicating that it has added the new device. For *NextGenGW w/ LwM2M* and *NextGenGW w/ OCRE*, we measured the duration from the moment the device starts until the moment NextGenGW published the device in the *announce* topic.

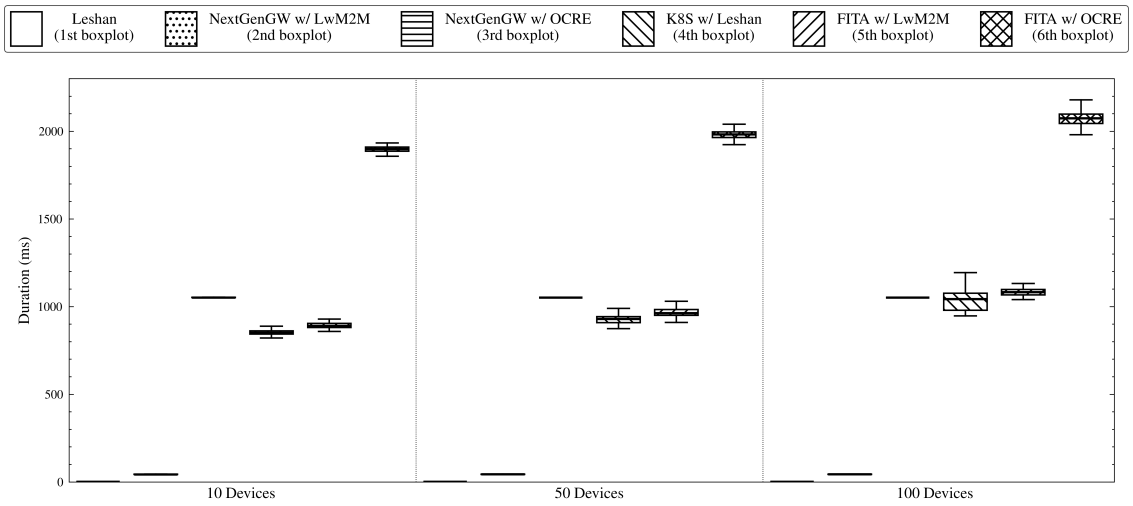


Figure 4.10: The time taken to report a new device as ready in the six considered solutions with a varying number of devices in the cluster.

In the device registration test, we varied the number of Nodes in the cluster and did not consider any application component. We repeated the tests 100 times.

The results shown in Figure 4.10 indicate that, considering median values, the NextGenGW-based solutions connected to LwM2M devices (*NextGenGW w/ LwM2M* and *FITA w/ LwM2M*) have an approximately constant overhead of 40 ms when compared with Leshan-based solutions (*Leshan* and *K8S w/Leshan*). NextGenGW registration procedures are the reason for this additional time. In addition to handling the translation between LwM2M and SDF bound with MQTT, NextGenGW inspects all the objects and resources of the registered device to create its digital copy. On the other hand, Leshan simply parses the objects reported by the device, assuming it supports all the LwM2M resources listed in the object specification.

When comparing *NextGenGW w/ OCRE* against *NextGenGW w/ LwM2M*, we observe that OCRE devices require approximately 1008 ms more to register. The comparison between *FITA w/ OCRE* and *FITA w/ LwM2M* provides similar results, but the difference varies between 1020 ms and 992 ms. The extra time for OCRE device registration is related to the runtime, which requires more time to complete its boot sequence, and not to NextGenGW or FITA mechanisms. Considering this, we can conclude that, depending on the runtime, the overhead introduced by the NextGenGW registration procedure might not be relevant for the total registration duration.

Looking at the K8S-based solutions and comparing them with the non-K8S solutions, we can see that while non-K8S solutions have a constant registration duration independently of the number of Nodes in the cluster (1,43 ms for *Leshan*, 42,4 ms for *NextGenGW w/ LwM2M* and 1008 ms for *NextGenGW w/ OCRE*), the K8S-based solutions show increasing registration duration with increasing number of Nodes in the cluster. This increase is 192 ms for *K8S w/Leshan* and *FITA w/ LwM2M* and 175 ms for *FITA w/ OCRE*, when considering the median values. This relatively small influence of the cluster size on the K8S registration procedure is likely due to the increase in the number of Nodes that are considered when scheduling the Far-edge Kubelet instance for

the new Node. Additionally, the minimum and maximum times the K8S registration procedure takes after the device registration in the Leshan or NextGenGW frameworks are approximately 846 ms and 1147 ms, which occur with OCRE devices, considering 10 and 100 Nodes in the cluster, respectively. This extra time outweighs the overhead introduced by NextGenGW when compared with Leshan, and it is related to the need to deploy a new container with the Far-edge Kubelet instance and the K8S Node registration process.

Focusing on FITA device registration duration, we have a minimum value of 850 ms (for *FITA w/ LwM2M* in a cluster with 10 Nodes) and a maximum value of 2200 ms (for *FITA w/ OCRE* in a cluster with 100 Nodes), which are reasonable intervals for such a procedure that is normally not time constrained.

4.3.2.4 Resource Analysis

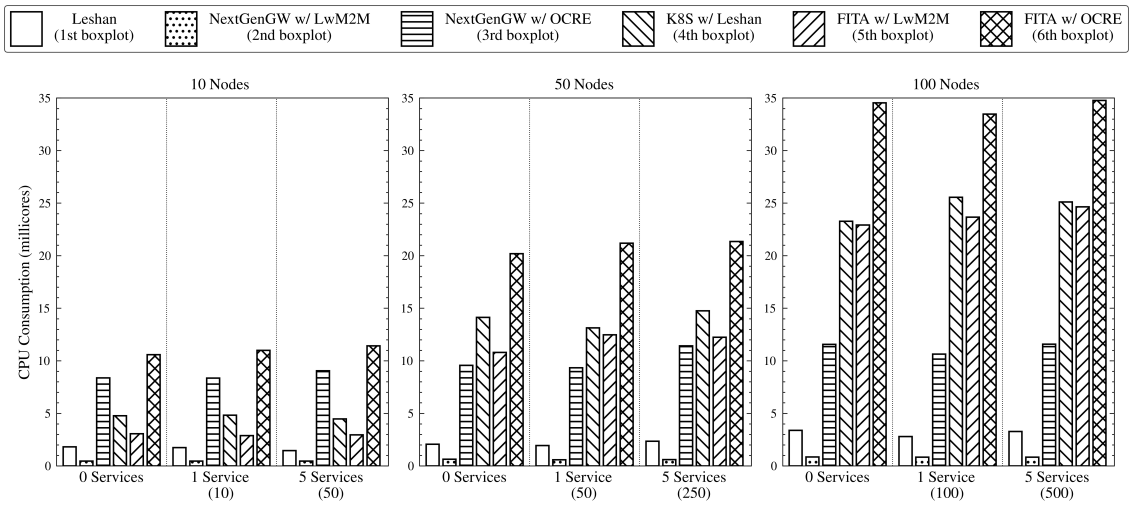
In the resource analysis tests, we assess the resource usage in terms of CPU and memory of FITA and compare it with the other evaluated solutions to understand the introduced resource overhead. We collected this data using the K8S *metric-server*, which provides information about the status of various K8S elements, including the Pod's CPU[15] and memory usage. The K8S *metric-server* is configured with a resolution of 15 seconds, which means that the reported CPU usage relates to the average of the last 15 seconds, and the memory data reports instantaneous usage every 15 seconds. Although this approach may not provide highly accurate resource data when compared to the usage of more complex and fine-grained solutions that include extra tools, such as Prometheus[16], it offers a reasonable approximation for our analysis and avoids the need for complex probing strategies.

The CPU and memory usage we collect from the K8S *metric-server* depends on the evaluated solution: in *Leshan*, we gather metrics from the Leshan Server container; in *NextGenGW w/ LwM2M* and *NextGenGW w/ OCRE*, we collect metrics from both the NextGenGW and MQTT broker containers; for *K8S w/Leshan*, we include to the metrics of the Leshan Server container the ones from the modified version of the Far-edge Kubelet (one per Node in the cluster) and Far-edge Node Watcher; for *FITA w/ LwM2M* and *FITA w/ OCRE*, we include to the metrics of the NextGenGW and MQTT containers the ones from the Far-edge Node Watcher and the multiple Far-edge Kubelet containers. We collected the CPU and memory data over 1000 seconds and averaged the set. The collection begins after completing device registration and deploying all the application components of the cluster setup under evaluation, which varies in the number of Nodes in the cluster (10, 50, and 100) and the Number of Pods per Node (0, 1, and 5).

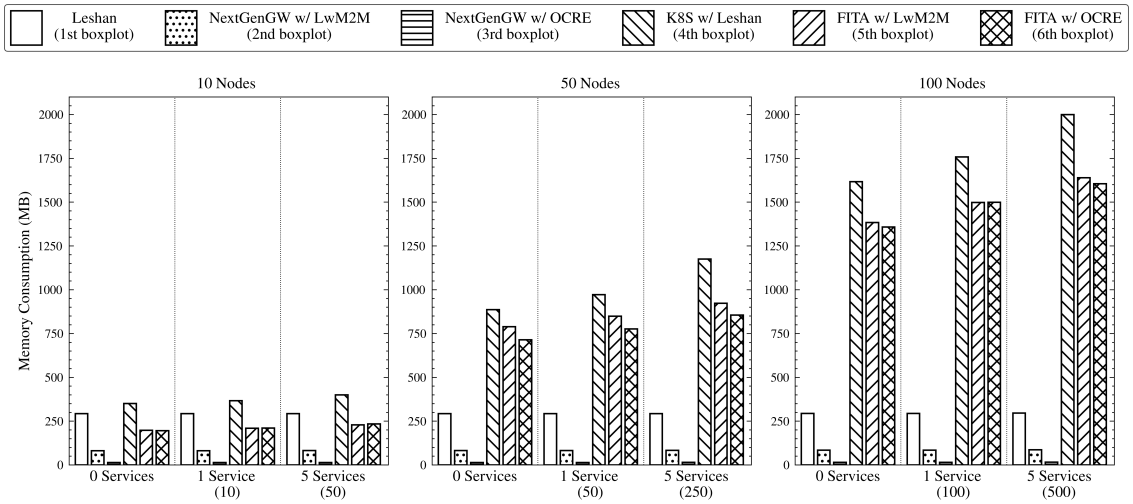
Figure 4.11a and 4.11b display the results for CPU and memory usage, respectively. Comparing *Leshan* against *NextGenGW w/ LwM2M*, we can conclude that NextGenGW requires fewer CPU and memory resources. The CPU difference varies between 1 and 2.54 millicores, while the memory difference has a constant value of approximately 210 MB across all cluster setups. Given that both solutions are mainly waiting for events on their communication interfaces, occasionally

[15] In K8S, CPU usage is reported in millicore, where 1 CPU unit is equivalent to 1 physical CPU core. Therefore, one millicore refers to 1/1000 of a physical CPU.

[16] https://prometheus.io/



(a) The CPU usage of the containers composing each of the six solutions under analysis, considering a varying number of devices and deployed Pods/application components in the cluster.



(b) The total memory usage of the containers composing each of the six solutions under analysis, considering a varying number of devices and deployed Pods/application components in the cluster.

Figure 4.11: Resource usage of the containers composing each of the six solutions under analysis, considering a varying number of devices and deployed Pods/application components in the cluster.

waking to handle LwM2M registration update messages, this seems to indicate that Leshan is less efficient in processing these messages, likely due to the overhead introduced by its runtime (Java) and its embedded web server. This difference in the efficiency of resource usage is also evident when considering the integration with K8S (*K8S w/Leshan* vs. *FITA w/ LwM2M*), with CPU varying between 0.46 and 3.33 millicores and memory between 98 MB and 360 MB. The higher variability comes from the Far-edge Node Watcher and Far-edge Kubelet, which differ between *K8S w/Leshan* and *FITA w/ LwM2M*. The former uses a modified version of these software modules that communicates with the Leshan server instead of NextGenGW MQTT broker.

When comparing *NextGenGW w/ OCRE* against *NextGenGW w/ LwM2M*, we can see the runtime influence in the NextGenGW-based solutions. The NextGenGW library that interacts

with OCRE consumes more CPU and less memory than the library that interacts with LwM2M. The implementation details of the libraries justify the CPU difference. The library that communicates with OCRE devices uses a libcoap mechanism that waits for CoAP related events ("coap_io_process") and is configured to time out after 5 ms. The library that communicates with LwM2M devices uses a similar mechanism based on Wakaama features and the C "select" socket monitoring function, which is configured to timeout by default after 60 seconds. During the resource analysis tests, the libraries are iterating over these functions, and consequently, the implementation with a longer timeout consumes less CPU. Regarding the memory usage difference, the OCRE data model is simpler than the LwM2M data model, as it contains fewer details about the device. Because of this, the library that communicates with OCRE devices uses less memory to hold their digital copy. With the integration of K8S, the CPU difference is also verified, but the memory difference is diluted. The Far-edge Node Watcher and Far-edge Kubelet memory usage are the reasons for the change in memory usage pattern. These two modules are the primary contributors to the total memory consumption in K8S- based solutions, especially the Far-edge Kubelet, due to its per-device instantiation.

Regarding scalability, *Leshan*, *NextGenGW w/ LwM2M*, and *NextGenGW w/ OCRE* show negligible differences in CPU and memory usage as the number of devices and Pods increases. The maximum difference in CPU usage is 3.0 millicores in *NextGenGW w/ OCRE*, and in memory usage is 4.63 MB in *NextGenGW w/ LwM2M*. On the other hand, *K8S w/Leshan*, *FITA w/ LwM2M*, and *FITA w/ OCRE* are affected both by the increase in the number of Nodes and Pods. The increase in the number of Pods has no significant impact on CPU usage (variations are around 0.37 millicores and 2.28 millicores for all K8S based solutions) and a slight impact on memory usage (maximum increase is 382 MB in *K8S w/Leshan* with 100 devices). The slight increase in memory usage is related to the additional memory required by each Far-edge Kubelet to track its associated Pods. The Far-edge Kubelet is also responsible for the biggest increase in CPU and memory usage, which occurs with the increase in the number of Nodes in the cluster. This increase is expected because K8S-based solutions instantiate a new Far-edge Kubelet for each new Node. Focusing on FITA based solutions (*FITA w/ LwM2M* and *FITA w/ OCRE*), if we make the CPU and memory difference between two setups with the same number of Pods and divide the difference by the increase in the number of Nodes, we get a relatively constant variation between 0.20 millicore and 0.26 millicores for CPU and 13 MB and 15MB for memory across all cluster loads. This is the amount of CPU and memory each new IoT Kubelet adds to resource usage, and could be considered to estimate the maximum number of Nodes supported by the hardware hosting the Far-edge Kubelets. Our setup consists of 8 CPUs and 16GB of RAM, so the scalability of the Nodes is not an issue, as can be seen by the measured results, which reach a maximum of 35 millicores for 100 OCRE Nodes and 1.65GB of memory for 100 LwM2M Nodes.

FITA evaluation shows that using Cloud native orchestration solutions, such as K8S, for the management of IoT installations comes with a runtime overhead, both in terms of resource usage and the duration of deployment and recovery of application components, as well as device regis-

tration. Considering the approach proposed with FITA, this overhead comes from K8S intrinsic mechanisms, and the software modules that compose it, namely: NextGenGW, responsible for addressing IoT devices heterogeneity; Far-edge Node Watcher, responsible for the registration and removal of IoT devices from the cluster; and Far-edge Kubelet, responsible for hosting the virtual instance of the IoT device in the cluster and bridging the K8S API server with NextGenGW.

When we weigh the added overhead with the benefits of facilitated IoT continuum orchestration, the results indicate that such overhead is acceptable for IoT use cases without stringent time constraints. In use cases with stringent time constraints for application component deployment and recovery, the use of NextGenGW for homogenising IoT runtime environments, as well as data and interaction models, could be reconsidered. The homogenisation feature is particularly beneficial in installations with a high number of heterogeneous devices, such as those envisioned by IoT (Smart Cities, Industrial IoT, and Smart Buildings, to name a few). However, such benefits might not be relevant for use cases where the timing requirements are not compliant with the measured overhead. Regarding application component recovery, one could also explore the use of mechanisms that manage redundant replicas of critical application components, thereby avoiding the time spent redeploying them in case of failure.

In conclusion, the results demonstrate that FITA effectively achieves its purpose with an acceptable overhead. Nevertheless, the user should always consider the measured overhead when planning their IoT installations and weigh it against the benefits brought with the proposed approach towards IoT continuum orchestration.

4.4 Summary

In this chapter, we presented FITA, which integrates IoT devices in K8S, considering their heterogeneous runtime environments, as well as data and interaction models. We began the chapter by providing an overview of K8S and some of its key concepts to facilitate an understanding of FITA. We then presented FITA and its components, namely: NextGenGW, Far-edge Kubelet and Far-edge Node Watcher. Regarding NextGenGW, we focused on explaining how we leveraged its features to address IoT devices heterogeneous runtime environments, as well as data and interaction models. For that purpose, we used embServe, a runtime based on dynamic loadable components exposed via LwM2M, and OCRE, which is based on the Wasm virtual machine and is exposed via CoAP. The pairing between OCRE and CoAP was implemented by us, so we have also presented it in this chapter. Concerning the Far-edge Kubelet, we have detailed how it bridges the K8S API with NextGenGW, and its one-to-one relation with the IoT devices and their digital representation in the cluster through Virtual Nodes. In what respects the Far-edge Node Watcher, we have explained how it is used to add and remove IoT devices to/from the K8S cluster through the creation and deletion of the Far-edge Kubelets and Virtual Nodes, which digitally represent them. We have also explained how the specific characteristics of IoT devices, such as their sensors, actuators, runtime, etc., are digitally represented in the Virtual Node.

We closed the chapter by evaluating FITA performance, which we compared with different baselines to assess the overhead introduced by the orchestration features provided by K8S and the heterogeneity abstraction brought by NextGenGW. We measured and analysed FITA's resource usage, deployment and recovery duration, as well as device registration, and concluded that the introduced overhead is acceptable for IoT use cases without stringent time constraints, specially in installations with a high number of heterogeneous devices, such as those envisioned by Smart Cities, Industrial IoT, and Smart Buildings. In use cases with stringent time constraints, we should consider the measured results to evaluate the applicability of FITA.

Chapter 5

QOIA: Quality-aware Orchestration

In Section 2.3.1, we observed that the literature includes intense discussions on the quality characteristics that affect IoT applications because traditional QoS and QoE approaches are insufficient in this scope. IoT applications might not have a human as a consumer, might run on heterogeneous devices and be composed of multiple services featuring sensing, computing, communication, and actuation aspects, each with its quality metrics [32, 62, 99]. New terms, such as QoT, QoIoT or Quality Enabled IoT Applications, were proposed to define the quality aspects of IoT applications [32]. However, independently of the definition, no standard set of characteristics and metrics defines quality for IoT applications. Each application will have its requirements and value certain characteristics and metrics over others, needing its own quality model [99][85].

Regarding quality guarantees, enforcing a quality model in the IoT requires appropriate application orchestration. Section 2.3.2 shows that the research community has proposed schedulers for application component selection and placement, but these are application-specific and tied to particular sets of quality characteristics, metrics, and models. Such a per-application approach might provide fine-tuned scheduling results, but it requires a different scheduler for each running application. We argue this is not scalable and hampers IoT dissemination.

In this chapter, we propose the Quality-aware Orchestration for IoT Applications (QOIA) to address this issue. QOIA tackles both the scheduling and life cycle management of IoT applications, considering a configurable approach regarding their quality requirements and scheduling optimisation objectives. The study of how to measure the quality metrics that feed the orchestration is a research topic per se [32], and it is outside the scope of this Thesis. Section 5.1 overviews K8S mechanisms and background technology used by QOIA. Section 5.2 details QOIA. Section 5.3 demonstrates QOIA's configurability through three different use cases: Smart Building, Smart Industry, and Smart Lighting. Section 5.4 evaluates QOIA's performance and scalability regarding the number of devices and applications, showcasing its capability to handle the quality requirements of IoT applications at scale. Section 5.5 concludes this chapter.

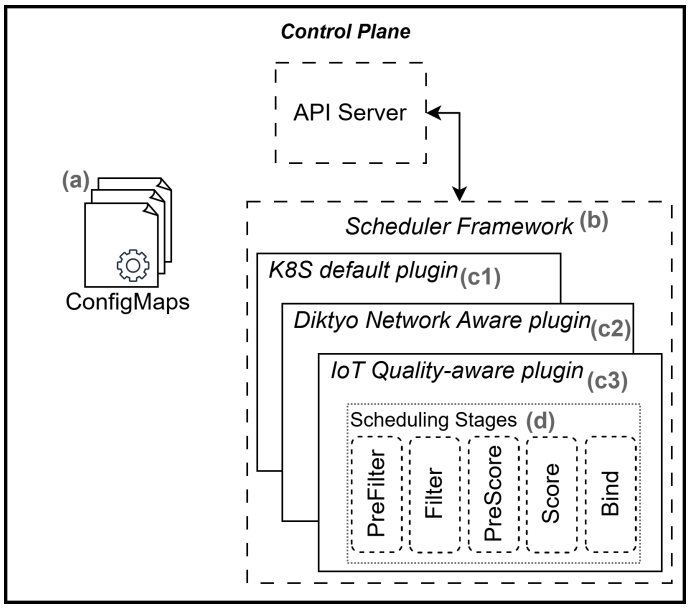


Figure 5.1: K8S configMaps and Scheduling Framework, including Diktyo Network-aware plugin.

5.1 Background Technology and Terminology

In Section 4.1, we overviewed the K8S elements necessary for understanding FITA. For understanding QOIA, in addition to those elements, we need to introduce: (1) the K8S configMap (cf. a, Figure 5.1); detail the K8S scheduling framework (cf. b, Figure 5.1); and overview Diktyo Network-aware plugin (cf. c2, Figure 5.1).

The K8S configMap[1] is a YAML configuration mechanism hosted in the control plane that allows defining configurations for Pods and other K8S entities.

The K8S Scheduler Framework[2] defines how Pods are bound to nodes using two mechanisms, the scheduling plugins (cf. c1 to c3) and stages (cf. d). The scheduling stages define the sequence of tasks to be performed, from the time a user requests to schedule a Pod until that Pod is bound to a Node. The plugins allow having different scheduling perspectives for these stages, which, in combination, define how Pods are scheduled. Each stage can be associated with multiple plugins, i.e., one can combine multiple per-stage plugins to build a complex scheduler that considers multiple scheduling perspectives or objectives.

The default K8S scheduling plugin places Pods in Nodes considering CPU and memory requirements, Node availability, affinity and anti-affinity rules, taints and tolerations, as well as topology constraints. It does not support the IoT Quality requirements discussed in Section 2.3.1. However, one can define custom scheduling plugins that add support for new scheduling requirements and objectives by specifying the operations to be executed at each scheduling stage. We follow this strategy with QOIA to bring Quality-aware IoT application orchestration into K8S. The relevant scheduling stages to understand QOIA are:

[1] https://kubernetes.io/docs/concepts/configuration/configmap/
[2] https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

- QueueSort: sorts Pods in the schedule queue before the scheduling procedure starts;
- PreFilter: has full Cluster access, allowing pre-processing Pod and/or Cluster information and run pre-check conditions on the Cluster and/or Pod before the Filter stage;
- Filter: filters the Nodes compliant with the Pods scheduling requirements;
- PreScore: has access to the list of filtered Nodes, allowing a pre-assessment before the scoring stage. For example, it can analyse the list of filtered Nodes and provide information to the Score stage beyond the Node under analysis;
- Score: ranks the filtered Nodes to optimise the Pod placement according to a score optimisation logic;
- Bind: binds the Pod to the selected Node. The binding information is shared with the API server, which is responsible for informing the selected Node about the Pod it needs to instantiate.

Diktyo is the Network-aware K8S scheduling plugin proposed by José Santos et al. [97] and available open source in the K8S out-of-tree scheduler plugins repository[3]. We use Diktyo to handle Pods QoS requirements regarding communication latency and bandwidth. Diktyo schedules applications considering these QoS metrics through the definition of the application graph and the network topology of the Nodes. Both are defined as K8S Custom Resource Definitions (CRD)[4], which are a K8S extension mechanism to integrate new resources into K8S. The application graph is defined in the AppGroup CRD and specifies the components that comprise the application, the dependencies between them, and the latency and bandwidth requirements for these dependency links. The Network Topology CRD identifies the zones and regions to which the Nodes belong, as well as the communication links between these regions and zones, along with the latency and bandwidth associated with the links. With this information, Diktyo sorts the Pods according to the topological order of the application graph, and then bounds the Pods to the Nodes based on the latency and bandwidth requirements specified in the AppGroup CRD, as well as the current latency and bandwidth information provided by the Network Topology CRD.

5.2 Quality-aware Orchestrator for IoT Applications

Figure 5.2 presents QOIA, our proposal for a configurable orchestration solution, decoupled from a particular set of quality characteristics or metrics. QOIA is highlighted using black lines, while the elements with which QOIA communicates are represented in grey.

[3] https://github.com/kubernetes-sigs/scheduler-plugins/blob/master/pkg/networkaware/README.md

[4] https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

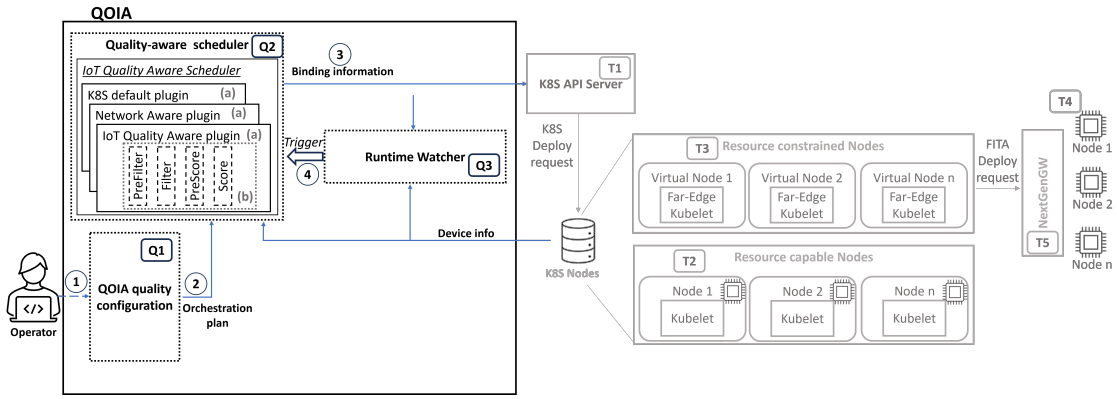


Figure 5.2: IoT Quality-aware Orchestrator.

QOIA handles both scheduling and runtime management, and uses K8S to enable Cloud to IoT continuum orchestration. The configurable scheduling and runtime management are achieved in four steps (cf. 1 to 4 in Figure 5.2):

1. The operator uses QOIA specification language (cf. Q1 and detailed in Section 5.2.1) to define the quality requirements and scheduling optimisation objectives of each application component. This produces what we call an orchestration plan.

2. The orchestration plan feeds QOIA's Quality-aware Scheduler (cf. Q2 and detailed in Section 5.2.2), which interprets the orchestration plan, identifies the devices in the cluster that are compliant with the plan and binds application components to devices.

3. The binding information is shared with the K8S API server and the Runtime Watcher (cf. Q3 and detailed in Section 5.2.3). The former triggers the application component deployment, and the latter checks at runtime if the deployed application components are still compliant with the specified quality requirements.

4. If a deployed application component is no longer compliant with the specified quality requirements, the Runtime Watcher triggers a reschedule.

Compared to the State-of-the-Art, this approach frees the operators from having to build their own scheduler and life cycle management solutions. They just need to specify their quality requirements and optimisation objectives using QOIA's specification language, which is not tied to a specific set of quality characteristics or metrics.

Although the deployment procedure shown in grey in Figure 5.2 is not the main focus of this chapter, we briefly describe it to facilitate a better understanding of QOIA and how we used resource-constrained and resourceful Nodes in its evaluation. The binding of Pods to Nodes is shared with the K8S API server (cf. T1), which triggers the deployment of the Pod by sending a start deployment request to the Node bound with such Pod. In resourceful Nodes (cf. T2), this procedure follows the normal K8S flow controlled by the K8S Kubelet. In resource-constrained Nodes (cf. T3), the procedure is controlled by FITA, namely its Far-Edge Kubelet that is associated

Listing 5.1: Quality Configuration Example.

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: temperature-quality-configs
5    namespace: quality-aware-scheduling
6  data:
7    temperature-quality-configs.yaml: |
8      filter:
9        - metrics:
10         operator: "LtEq"
11         referenceValue: 1
12         metacc: "temp/accuracy"
13       - metrics:
14         operator: "Eq"
15         referenceValue: "zoneX"
16         metl: "location"
17       - metrics:
18         operator: "GtEq"
19         referenceValue: 0.8
20         metts: "TrSc"
21     prescore:
22       function: "Max"
23       metrics:
24         mettrsc: "TrSc"
25     score:
26       function: "Max"
27       metrics:
28         mettrsc: "TrSc"
```

with a virtual Node inside the K8S Cluster and receives the K8S API server deployment request. The Far-Edge Kubelet bridges such requests with the actual resource-constrained Node (cf. T4), whose deployment API is abstracted and exposed through the NextGenGW (cf. T5). Refer to Chapter 4 for a more detailed explanation of FITA components and operation.

5.2.1 QOIA Quality Configuration

QOIA quality configuration is specified using a YAML-based specification language paired with K8S configMaps. The K8S configMaps are the K8S resource we use to inform the IoT Quality-aware Scheduler and the Runtime Watcher about each Pod's quality requirements and optimisation objectives.

Listing 5.1 presents a quality configuration example. The YAML file with the quality configurations is defined in the key "data" (cf. line 6) of the configMap. It defines the configuration for the Filter (cf. lines 8-20), PreScore (cf. lines 21-24), and Score (cf. lines 25-28) stages of the K8S scheduling framework, which are used by the IoT Quality Aware Scheduler detailed in Section 5.2.2. The name (cf. line 4) and namespace (cf. line 5) of the configMap identify the quality configuration and are used to associate it with the Pods it relates to. Such association is done by using the labels "qoia/quality-configmap" and "qoia/quality-configmap-namespace" in the metadata of the Pod definition (cf. lines 7 and 8 of Listing 5.2).

The Filter configuration is defined with the key "filter" (cf. line 8), comprising the list of quality requirements the application component needs to work properly. Each element in this list

Listing 5.2: Temperature Pod Example.

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: temperature
5    labels:
6      app: temperature
7      qoia/quality-configmap: temperature-quality-configs
8      qoia/quality-configmap-namespace: quality-aware-scheduling
9  spec:
10   schedulerName: qoia-scheduler
11 (...)
```

is identified with the key "metrics" (cf. lines 9, 13, and 17) and composed of a three-element tuple: (1) the key "operator" (cf. line 10, 14, and 18) defines the operation used to evaluate the quality metric, (2) the key "referenceValue" (cf. lines 11, 15, and 19) is the reference value against which the operator will reason, and (3) the "metacc", "metl" and "metts" (cf. lines 12, 16, and 20, respectively) identify the quality metric to consider for evaluation. We do not define a specific key for the quality metric, allowing the user to specify different keys for different quality metrics, which may be useful for the custom operators explained later in this section.

The PreScore configuration is defined with the "prescore" key (cf. line 21) and is composed of a two-element tuple: (1) the key "function" (cf. line 22) specifies the function to use at the PreScore stage, and (2) the key "metrics" (cf. line 23) specifies the inputs of such function.

The Score configuration is defined with the key "score" (cf. line 25) and comprises a two-element tuple: (1) the key "function" (cf. line 26) defines the score optimisation function, and (2) the key "metrics" (cf. line 27) defines the metrics that feed it.

Regarding the values accepted by the different keys of the YAML configuration file, the key "referenceValue" of the Filter stage accepts a number, a string or a list of strings or numbers. The keys "operator" of the Filter stage and "function" of the PreScore and Score stage accept a string with the operation and function names, respectively. QOIA supports a set of built-in operators and functions. For the Filter stage, QOIA supports the operations equal ("Eq"), different ("Df"), greater than ("Gt"), less than ("Lt"), greater than or equal ("GtEq"), and less than or equal ("LtEq"). At the Score stage, it supports the optimisation functions minimisation ("Min") and maximisation ("Max") of one metric.

The configuration options explained above already allow QOIA to support a wide range of use cases. Nevertheless, we provide an extension mechanism that allows users to develop extra operators and functions if the QOIA default ones are not enough to handle the intended quality requirements and optimisation objectives. Using the Go plugin feature, users can create such custom quality models and optimisation functions, aggregating them into a set of libraries that are regularly updated and where new complex operators and functions are added over time, thereby extending the list of operators and functions supported by QOIA.

The new operators and functions are identified in the QOIA configuration file by merging the name of the plugin and the plugin symbol in the following format: "<plugin-name>/<plugin-

symbol-name>". In Listing 5.4, QOIA is configured to use the plugin name "custom-operator-plugin", which contains the symbols "QoDNoise", "MinTotalPower" and "NormalizeNodeCost" (cf. lines 6, 10, and 15).

QOIA exposes this extension mechanism, but is not responsible for managing the libraries of new custom operators and functions. It simply utilises them as a source of operators and functions for scheduling and life cycle management. These libraries need to be available in the Node where QOIA scheduler and life cycle management modules are running. Their location in the Node is identified with the command line argument "custom_quality_models_path" of QOIA scheduler and life cycle management modules.

Considering QOIA's quality configuration approach presented in this section, users do not need programming knowledge to schedule and monitor applications at runtime. They just need to configure their intended scheduling requirements and optimisation goals in YAML configuration files. QOIA removes the burden of building the scheduler and life cycle management modules by providing them (Section 5.2.2 and Section 5.2.3, respectively) as configurable solutions that follow QOIA's quality configuration approach. Users with programming knowledge can extend QOIA with custom operators and functions if the ones supported by default by QOIA are not enough. The burden of programming these quality models and optimisation functions is nevertheless reduced when compared to building the complete scheduler and life cycle management modules. Additionally, such custom operators and functions can be shared by the users who developed them as libraries, making them available to users without programming knowledge.

Section 5.3 models QOIA's quality configuration approach against different use cases, highlighting QOIA's versatility and ease of use, considering concrete configuration examples that include the usage of custom operators and functions through the Go plugin feature.

To close the description of QOIA's quality configuration, we note that the QoS requirements of each application component are currently defined in Diktyo's AppGroup custom resource (Section 5.1). We plan to integrate them in the QOIA configMap to provide the user with a single configuration point. We will explore K8S' Admission Controllers[5], which can track QOIA configMap creation and update, extract the QoS info from it and update Diktyo's AppGroup accordingly.

5.2.2 IoT Quality-aware Scheduler

The IoT Quality-aware Scheduler brings Quality-aware IoT application component scheduling to the K8S scheduler. It follows a configurable approach that makes it independent from a particular set of quality characteristics or metrics. The proposed IoT Quality-aware Scheduler takes advantage of both the multiple scheduling plugins (cf. (a) in Figure 5.2) and multiple scheduling stages (cf. (b) in Figure 5.2) of the K8S Scheduling Framework. Regarding the scheduling plugins, the IoT Quality-aware Scheduler is composed of:

- K8S default plugin: handles all the application component scheduling requirements already supported by K8S (Section 5.1);

[5] https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers

- Diktyo Network-aware plugin: handles the application component scheduling requirements for latency and bandwidth (Section 5.1);
- IoT Quality-aware plugin: a configurable scheduler (our proposed scheduler plugin) that is fed by QOIA's quality configuration and defines the binding of IoT application components to Nodes, considering their quality requirements and scheduling optimisation objectives (Section 5.2.2.1).

In our implementation, the metrics used to evaluate the IoT application component quality characteristics are provided as labels in the K8S Nodes. The exception is the QoS metrics, which are defined in Diktyo's Network Topology CRD. As a future research direction (Section 6.2.1), we will explore how to utilise CRDs with IETF SDF [57] to represent metrics in a standardised format.

5.2.2.1 IoT Quality-aware plugin

The IoT Quality-aware plugin is built in Go and is highly configurable. It considers user-defined quality requirements and optimisation objectives (Section 5.2.1) by providing extensions to the PreFilter, Filter, PreScore, and Score stages (cf., (b) in Figure 5.2) of the K8S Scheduling framework. This section details the mechanisms used in each stage.

PreFilter

This step loads the Pod's configMap (Section 5.2.1). It parses the YAML file contained in the configMap and sets the internal IoT Quality-aware plugin data structures for each scheduling stage.

For the Filter stage, it sets the three-element tuple of Equation (5.1c), where q_{Fl} is the l^{th} quality requirement of the Pod's quality set. The tuple mapping is: (1) $q_{Fl.metrics}$ is the list of metrics that compose the quality (cf. lines 9, 13, and 17 in Listing 5.1), (2) $q_{Fl.operator}$ is the value of the key "operator" (cf. lines 10, 14, and 18), and (3) $q_{Fl.refValue}$ is the value of the key "referenceValue" (cf. lines 11, 15, and 19).

In the PreScore stage, it sets the tuple in Equation (5.2a), where $Q_{PS.function}$ is the value of the key "function" (cf. line 22) and $Q_{PS.metrics}$ is the key value pair associated with the key "metrics" (cf. line 23).

For the Score stage, it sets the tuple in Equation (5.3a). $Q_{S.function}$ is the value associated with the key "function" (cf. line 26). $Q_{S.metrics}$ is the key-value pair associated with the key "metrics" (cf. line 27).

Filter

Once a Pod reaches the Filter stage, the IoT Quality Aware plugin follows the objectives defined in Equations (5.1a) to (5.1d), i.e., it creates the set V^i of Nodes from the cluster N that meet all the quality requirements Q_F^i of the Pod p_i being scheduled.

$$N = \{n_1, n_2, ..., n_c\} \text{ is the set of Nodes in the cluster} \tag{5.1a}$$

$$Q_F^i = \{q_{F_1}^i, q_{F_2}^i, ..., q_{F_l}^i\} \text{ is the set of } l \text{ quality requirements of } p_i \text{ for the Filter stage} \tag{5.1b}$$

$$q_F^i = (q_{F.operator}^i, q_{F.metrics}^i, q_{F.refValue}^i) \tag{5.1c}$$

$$V^i = \{\forall n_j \in N, \forall q_{Fl}^i \in Q_F^i \mid q_{Fl.operator}^i(n_j, q_{Fl.metrics}^i, q_{Fl.refValue}^i) = True\} \tag{5.1d}$$

PreScore

The PreScore stage at the IoT Quality Aware Plugin only runs if it is configured in the configMap (Section 5.2.1). When configured, the IoT Quality Aware Plugin follows the objective defined in Equation (5.2c), which is to prepare the cluster-aware metrics PS_{out}^i of Pod p_i under scheduling. PS_{out}^i is later used by the optimisation function defined for the Score stage (cf., Equation (5.3c)).

$Q_{PS.function}^i$ is the PreScore function that calculates PS_{out}^i. It is defined by the user using QOIA's quality configuration and has as inputs the list of filtered Nodes V^i that result from the Filter stage (cf., Equation (5.1d)), the metrics $Q_{PS.metrics}^i$ defined in the configMap and the Pod p_i under scheduling. PS_{out}^i does not have a predefined type, i.e., in the IoT Quality-aware plugin implementation, it is defined as a Go *interface*. This means the users can define their $Q_{PS.function}^i$ with any output type, as long as they handle it correctly in the optimisation function at the Score stage.

$$Q_{PS}^i = (Q_{PS.function}^i, Q_{PS.metrics}^i) \tag{5.2a}$$

$$Q_{PS.metrics}^i = \{q_{PS.metrics_1}^i, q_{PS.metrics_2}^i, ..., q_{PS.metrics_k}^i\} \text{ is the set of } k \text{ quality metrics of } p_i \text{ for the PreScore stage} \tag{5.2b}$$

$$PS_{out}^i = Q_{PS.function}^i(V^i, Q_{PS.metrics}^i, p_i) \tag{5.2c}$$

Score

When the Pod reaches the Score stage, the IoT Quality Aware Plugin creates the set $\bar{V}_i$ (Equation (5.3c)) that extends the set of filtered Nodes V^i (Equation (5.1d)) attaching to each Node the corresponding output of the optimisation function $Q_{S.function}^i$. This function receives as inputs the Node being scored v_j^i of the set V^i, the outputs of the PreScore stage PS_{out}^i (Equation (5.2c)), the Pod p_i under scheduling, and the metrics $Q_{S.metrics}^i$ to be considered for optimisation.

$$Q_S^i = (Q_{S.function}^i, Q_{S.metrics}^i) \tag{5.3a}$$

$$Q_{S.metrics}^i = \{q_{S.metrics_1}^i, q_{S.metrics_2}^i, ..., q_{S.metrics_z}^i\} \text{ is the set of } z \text{ quality metrics of } p_i \text{ for the Score stage} \tag{5.3b}$$

$$\bar{V}_i = \{\forall v_j^i \in V^i \mid (v_j^i, Q_{S.function}^i(v_j^i, PS_{out}^i, p_i, Q_{S.metrics}^i))\} \tag{5.3c}$$

At the end of the scheduling cycle, the IoT Quality-aware plugin binds Pod p_i being scheduled to Node $\bar{v_j^i} \in \bar{V}_i$ that meets quality requirements Q_F^i and optimisation objective Q_S^i. Section 5.3 provides examples of how the equations of each scheduling stage map to QOIA's configuration.

5.2.3 Runtime Watcher

In addition to the scheduler, QOIA is composed of the "Runtime Watcher" (cf., Q3 in Figure 5.2), which is responsible for monitoring the life cycle of the running Pods. It does this by watching for modifications to the configuration of each Pod's quality requirements, the Node on which each Pod is running, and its network status. The objective is to guarantee that each running Pod still meets its quality requirements Q_F^i expressed in Equation (5.1b). This means our life cycle management strategy does not consider the optimisation objective defined by $Q_{S.function}^i$ (Equation (5.3c)). We opted for a strategy that keeps the Pod alive as long as it meets its requirements to work correctly, even if it no longer meets the scheduling optimisation objective. With this strategy, we reduce the number of evictions to the minimum required for the application to continue working properly.

Figure 5.3 presents the activity diagram of the Runtime Watcher. It starts by loading the Nodes available in the cluster (cf. (a)), the QOIA quality configurations (cf. (b)), and the Pods associated with QOIA quality requirements, along with their Diktyo AppGroup and Network topology information (cf. (c)). If there is an issue while loading such information (e.g., they are unavailable or not correctly configured), the Runtime Watcher exits (d). If that is not the case, the Runtime Watcher starts monitoring for changes in these resources by using the K8S Go Informer API (cf. (e) for network changes, cf. (f) for Pods, cf. (g) for QOIA configMap, cf. (h) for Nodes), which is a K8S mechanism that provides notifications when K8S resources are added, deleted, and updated.

If there are updates in the network, the Runtime Watcher checks which Pods have their network requirements no longer valid, evicting them (cf. (i)). We apply the same logic for the updates in QOIA configMaps and on the Nodes in the cluster. When a QOIA configMap is updated, the Runtime Watcher checks if the Pods associated with those quality requirements are still valid, evicting them if that is not the case (cf. (j)). When a Node is updated, the Runtime Watcher checks if the update affects the validity of the requirements of the Pods running on that Node, evicting the non-valid ones (cf. (k)). The evicted Pods are then added to the scheduling queue to be rescheduled.

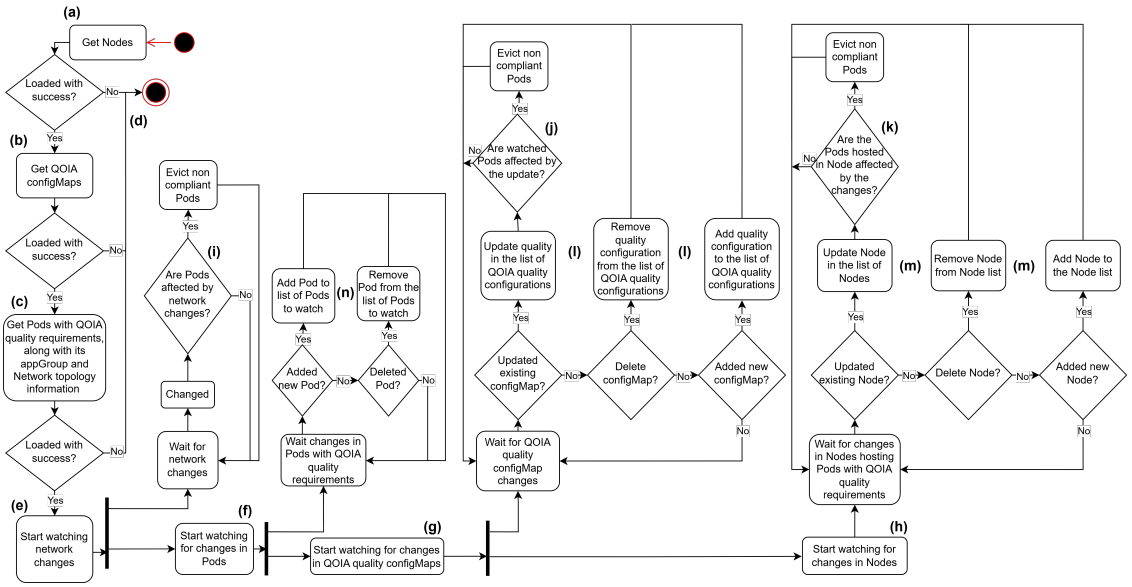


Figure 5.3: Runtime Watcher activity diagram.

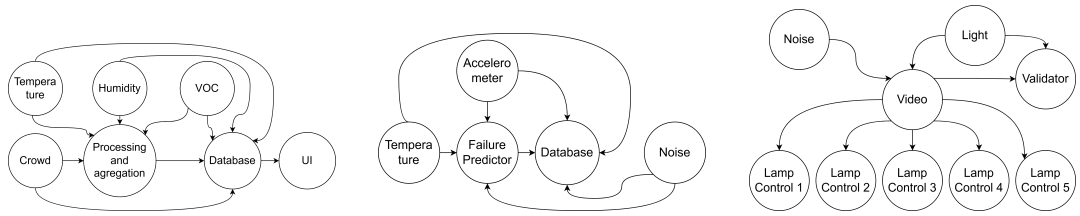
The update, addition, and removal of QOIA configMaps and Nodes, as well as the addition and removal of Pods, are reflected in the Runtime Watcher internal structures (cf. (l), cf. (m), and cf. (n), respectively). We use such structures to store information (such as names, namespaces, etc.) about those resources in memory. This strategy slightly increases the Runtime Watcher memory usage but reduces the number of requests performed to the K8S API server and the Runtime Watcher eviction time.

5.3 Using QOIA's Configurability to Model Different Use Cases

This section demonstrates how to use QOIA's flexible configurability in terms of scheduling requirements and optimisation objectives for three distinct use cases: Smart Building, Smart Industry, and Smart Lighting. Figure 5.4 presents the application graphs of these use cases, while the Equation systems (5.4a) to (5.6e) present the requirements and scheduling optimisation objectives for each application component of each example application. The last equation in each equation system represents the scheduling goals (composed of quality requirements and optimisation objective). The remaining equations in the system represent the quality models of the quality characteristics that compose the scheduling goals. In the remainder of this section, we further detail each application and how its quality models, requirements and optimisation objectives are configured using QOIA quality configuration.

5.3.1 Smart Building Use Case

Figure 5.4a presents a smart building use case that could be applied in a public setting, such as an airport. Three application components ("Temperature", "Humidity", and "Volatile Organic Compound" - "VOC") track the environment and air quality of the location where they are installed.



(a) Smart Building application graph. (b) Smart Industry application graph. (c) Smart Lighting application graph.

Figure 5.4: QOIA application examples.

The "Crowd" component collects statistical data about crowds in the monitored location. All these components share their data with the "Processing and Aggregation" component, which performs cross-evaluation of the crowd statistics with the environmental data. All the components save their data in a database, which can be consulted via a UI.

Equation systems (5.4a) to (5.4d) present the quality models, requirements and scheduling optimisation objectives for the Smart Building application components. It is worth noting that we omit the "Processing and Aggregation", "Database" and "UI" components, because the main quality requirements for these components are related to the Node processing and memory resources, which are supported by default by K8S, so QOIA does not need to orchestrate them.

We can model the scheduling requirements and optimisation objectives of the Smart Building application in the QOIA quality configuration file without using custom operators or functions. QOIA default functions and operators can handle this use case because:

1. The quality models used by the Smart Building application components are performing operations over a single metric. QoD equals accuracy (A) for the "Temperature" component; resolution (R) for the "Humidity" component; response time (RT) for the "VOC" component; and dynamic range (DR) and A for the "Crowd" component. QoDe equals location (L) for all components. QoSe&P equals trust score (TrSc) for all components (the trust score reflects how secure a Node is and can be measured using existing strategies [1]);

2. The quality requirements of all application components compare the quality characteristics of point 1 to a reference value using QOIA built-in operators ($\leq, =, \geq$);

3. The optimisation defined for the PreScore and Score stages of all application components is the maximisation ($\max()$) of a single metric (TrSc), which is also in the list of QOIA built-in functions.

Considering this, Listing 5.1 shows the YAML configuration file for the Temperature component. The quality requirements ($A \leq 1 \wedge L = zoneX \wedge TrSc \geq 0.8$) are added to the Filter stage. For example, the "operator" "LtEq" (cf. line 10) is used to compare the "referenceValue" "1" (cf. line 11) against the value of the metric "temp/accuracy" (cf. line 12). The metric value "temp/accuracy" maps to the Node label that hosts the values of this metric. The optimisation goal $\max(TrSc)$ is defined at the PreScore and Score stages. The value for the "function" is "Max" and the metric

used by the function is "TrSc". Again, the metric value "TrSc" maps to the Node label that hosts the trust score value. The same rationale is used to model the other application components.

$$Temperature \begin{cases} QoD = A \\ QoDe = L \\ QoSe\&P = TrSc \\ \max(TrSc) \text{ subject to } (A \leq 1 \wedge L = zoneX \wedge TrSc \geq 0.8) \end{cases} \tag{5.4a}$$

$$Humidity \begin{cases} QoD = R \\ QoDe = L \\ QoSe\&P = TrSc \\ \max(TrSc) \text{ subject to } (R \leq 1 \wedge L = zoneX \wedge TrSc \geq 0.8) \end{cases} \tag{5.4b}$$

$$VOC \begin{cases} QoD = RT \\ QoDe = L \\ QoSe\&P = TrSc \\ \max(TrSc) \text{ subject to } (RT \leq 30 \wedge L = zoneX \wedge TrSc \geq 0.8) \end{cases} \tag{5.4c}$$

$$Crowd \begin{cases} QoD = DR_{video} \wedge A_{noise} \\ QoDe = L \\ QoSe\&P = TrSc \\ \max(TrSc) \text{ subject to } (DR_{video} \geq 120 \wedge A_{noise} \leq 5 \wedge L = zoneX \wedge TrSc \geq 0.8) \end{cases} \tag{5.4d}$$

5.3.2 Smart Industry Use Case

Figure 5.4b shows a failure prediction application in an industrial setting. The application components "Temperature", "Accelerometer" and "Noise" collect sensor data and share the calculated features with the "Failure Predictor" component. The "Failure Predictor" component predicts the failure of the machine being monitored. All data is saved in a "Database" for historical assessment.

Equation systems (5.5a) to (5.5c) present the quality models and scheduling goals of each application component, except the "Failure Predictor" and "Database", which were excluded for

the same reason as the application components "Processing and Aggregation", "Database" and "UI" of the Smart Building use case (Section 5.3.1).

The distinctive feature in the Smart Industry use case is the quality model for QoD. In all application components, QoD is equal to sensor ageing (SenAge). This use case comprises a harsh environment prone to fast sensor ageing, so the QoD for the Node where the application components can be deployed is modelled by a sensor ageing strategy such as those in the literature [105]. This quality model would retrieve the features and statistical data for each sensor (drift, uncertainty, outliers, etc.) from the database to calculate sensor ageing. The user would implement such a quality model using a custom operator for the filter stage using the method described in Section 5.2.1. The resulting operator could be shared as a library with the community to extend the list of QOIA's supported operators.

Listing 5.3 shows how this custom operator is configured in the filter stage. The "operator" value is "custom-operator-plugin/SensorAgeing" (cf. line 6), where "custom-operator-plugin" is the name of the Go plugin where the sensor ageing model is implemented, and "SensorAgeing" is the name of the symbol that holds the sensor ageing implementation. One can define the number of metrics needed for the custom operator calculation, each with its own user-defined key. In this example, we show this using the metric keys "metqod1", "metqod2" and "metqod3", whose values are "qod/metric1", "qod/metric2" and qod/metric3", respectively (cf. lines 2 to 5). These values represent the database entry that host the values of the metrics "metqod1", "metqod2" and "metqod3", which the custom operator will use in its calculations. The reference value ("referenceValue") against which the result of the "SensorAgeing" operator would compare is "0.3" in our example (cf. line 7). We assume the sensor ageing calculation is normalised between 0 and 1, with 0 representing the lowest impact of ageing in the sensor data. The logical operator used for the comparison ($\leq$) is embedded in the custom operator implementation.

$$Temperature \begin{cases} QoD = SenAge \\ \min(SenAge) \text{ subject to } (SenAge \leq 0.3) \end{cases} \tag{5.5a}$$

$$Accelerometer \begin{cases} QoD = SenAge \\ \min(SenAge) \text{ subject to } (SenAge \leq 0.3) \end{cases} \tag{5.5b}$$

$$Noise \begin{cases} QoD = SenAge \\ \min(SenAge) \text{ subject to } (SenAge \leq 0.3) \end{cases} \tag{5.5c}$$

Listing 5.3: Custom Operator Example.

```
1 filter:
2   - metrics:
3     metqod1: "qod/metric1"
4     metqod2: "qod/metric2"
5     metqod3: "qod/metric3"
6     operator: "custom-operator-plugin/SensorAgeing"
7     referenceValue: 0.3
```

5.3.3 Smart Lighting Use Case

Figure 5.4c shows a Smart Lighting use case that manages the automated dimming of lampposts in smart cities. The "Noise" application component detects activity and informs the "Video" component about it. The "Light" application component measures luminosity levels and informs the "Video" component if the luminosity levels are below a threshold. When the "Video" application component is informed about the existence of activity and luminosity below the threshold, it starts processing the video stream to classify the activity and defines the lamp dimming level. The "Lamp Control" application components (1 to 5) receive the dimming instructions from the Video component and act accordingly. The "Validator" application component receives the dimming instructions and measures the luminosity to check if the dimming was applied. It raises an alarm if that is not the case.

Equation systems (5.6a) to (5.6e) present the quality characteristics and scheduling goals for the smart lighting application components. There are two distinctive features:

1. The optimisation objective for all application components is the minimisation of the power cost ($min(Pc)$). This is a complex optimisation function not supported by the QOIA's built-in optimisation functions. Algorithms 1 and 2 present the pseudo code of $min(Pc)$ for the PreScore and Score custom functions, respectively. We explain the algorithm in Section 5.4.1.

2. The QoD for the Noise application component is also not supported by QOIA's built-in operators for the Filter stage. This custom operator is defined by the weighted sum of the normalised Accuracy ($\frac{A_{ref}}{A}$), response time ($\frac{RT_{ref}}{RT}$) and resolution ($\frac{R_{ref}}{R}$), where the ref numerators are the reference values of the ideal sensor (0.5, 2, and 0.1 in our example, which considers the sensors of Table 5.1 and are further detailed in Section 5.4) and the denominators are the real sensor metrics. We defined the weights for the sum to value response time and accuracy over resolution. Their concrete values would need to be fine-tuned in an applied deployment context.

Listing 5.4 shows how these custom operators and functions are modelled in the QOIA quality configuration file of the "Noise" application component.

The operator for QoD is modelled in the Filter stage (cf. line 1). The values of the metric keys "metacc", "metrp" and "metr" (cf. lines 3 to 5) contain the Node labels that host the values of the accuracy, response time, and resolution of the noise sensor, i.e., the values of these keys inform

Listing 5.4: Custom Operator and Function Example.

```
1  filter:
2    - metrics:
3      metacc: "noise/accuracy"
4      metrp: "noise/response-time"
5      metr: "noise/resolution"
6      operator: "custom-operator-plugin/QoDNoise"
7      referenceValue: 0.79
8  (...)
9  prescore:
10   function: "custom-operator-plugin/MinTotalPower"
11   metrics:
12     metsenspw: "noise/pw-cost"
13     metrapw: "radio/pw-cost"
14 score:
15   function: "custom-operator-plugin/NormalizeNodeCost"
```

the custom operator about which labels to search in the Node under analysis to calculate QoD. The reference values of the ideal sensor (A_{ref}, RT_{ref} and R_{ref}) in our example are embedded in the custom operator, but a different version of the custom operator could be configurable regarding these values, which would be passed as extra metrics. The custom operator in our example is implemented in the Go plugin "custom-operator-plugin" with the symbol "QoDNoise" (cf. line 6). The reference value against which to compare is "0.79" (cf. line 7). The logical operator used for the comparison ($\geq$) is embedded in the custom operator implementation.

The PreScore configuration (cf. line 9 in Listing 5.4) contains the reference to the PreScore function detailed in Algorithm 1. This function is hosted in the Go plugin "custom-operator-plugin" under the symbol "MinTotalPower" (cf. line 10 in Listing 5.4). The metrics that feed it are "metsenspw" and "metrapw", which indicate the Node labels that host the sensor power consumption and radio power consumption relevant for the calculation (cf. lines 8 and 11 of Algorithm 1). The Score configuration (cf. 14 in Listing 5.4) only contains the reference to the custom function. The Score logic (Algorithm 2) does not require input metrics, since it operates only on PS_{out}. This logic is implemented in the Go plugin "custom-operator-plugin" under the symbol "NormalizeNodeCost" (cf. line 15 in Listing 5.4).

$$Noise \begin{cases} QoD = 0.3(\frac{A_{ref}}{A}) + 0.6(\frac{RT_{ref}}{RT}) + 0.1(\frac{R_{ref}}{R}) \\ A_{ref} = 0.5 \\ RT_{ref} = 2 \\ R_{ref} = 0.1 \\ QoDe = L \\ QoS = Lat_{video} \\ \min(Pc) \text{ subject to } (QoD \geq 0.79 \wedge L = BI_w.BF_z \wedge Lat_{video} \leq 40) \end{cases} \tag{5.6a}$$

$$Light \begin{cases} QoD = A \\ QoDe = L \\ QoS = Lat_{video} \wedge Lat_{validator} \\ min(Pc) \text{ subject to } (A \leq 5 \wedge L = BI_w.BF_z \wedge Lat_{video} \leq 40 \wedge Lat_{validator} \leq 60) \end{cases} \tag{5.6b}$$

$$Video \begin{cases} QoD = DR \wedge R \\ QoDe = L \\ QoS = \{\text{Lat}_{LampCont_k} \mid k = 1, \ldots, 5\} \\ \min(Pc) \text{ subject to } (\\ \quad DR \geq 120 \wedge R \geq 2 \wedge L = BI_w.BF_z \wedge \text{Lat}_{validator} < 60 \wedge \text{Lat}_{LampCont_k} < 40 \; \forall k \in \{1, \ldots, 5\}) \end{cases} \tag{5.6c}$$

$$LampControl \begin{cases} QoDe = L \\ (L = BI_w.BF_z) \end{cases} \tag{5.6d}$$

$$Validator \begin{cases} QoD = A \\ QoDe = L \\ \min(Pc) \text{ subject to } (A \leq 5 \wedge L = BI_w.BF_z) \end{cases} \tag{5.6e}$$

In conclusion, QOIA quality configuration strategy covers both simple quality requirements, which use single metric quality models and optimisation strategies (for example, the Smart Building use case of Section 5.3.1), as well as custom quality models and optimisation goals, which can be defined as Go plugins by users with programming knowledge (such as the examples, in the Smart Industry and Smart Lighting use case of sections 5.3.2 and 5.3.3, respectively). These Go plugins extend the list of operators and functions supported by QOIA. The users who develop them can aggregate their Go plugins into a set of libraries and share them, allowing other users to leverage the new operators and functions in their IoT installations.

The use case examples in this section demonstrate that, with QOIA's quality configuration, users simply need to specify their quality models and scheduling goals in YAML files, leaving the scheduling itself and runtime management to QOIA, as explained in Section 5.2.

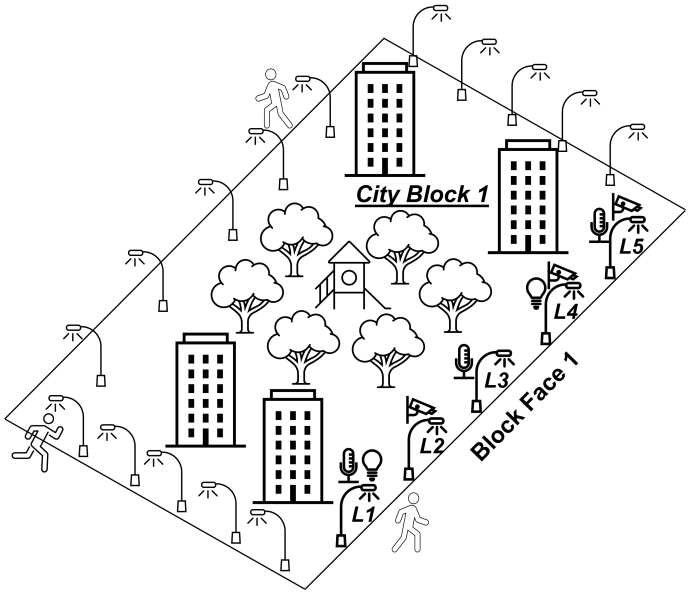


Figure 5.5: Smart Lighting use case infrastructure.

5.4 Evaluation

We evaluate QOIA performance against the Smart Lighting use case introduced in Section 5.3. We selected this use case because it is the most complex, covering both custom operators and functions, and it easily scales in terms of the number of Nodes and Pods in the cluster, allowing us to assess QOIA at scale. Additionally, Smart Lighting is a relevant application in the scope of IoT. Smart Lighting is one of the services of Smart Street Lighting, which enhances the functionality of the lighting infrastructure by providing, in addition to public lighting, a connectivity and IoT infrastructure composed of sensors that enhance city liveliness [18]. Smart Street Lighting is envisioned as the future of lighting in Smart Cities, with an annual market growth of 22.7% per year and an expectation to reach 63.8 million connected street lights in 2027, as well as a market of USD 50 billion by 2028[6]. Smart Lighting runs on top of this infrastructure, using the sensors embedded in the lampposts to automate light dimming according to the space and citizens' needs. The aim is to reduce energy consumption, with expected savings between 50% and 70% [18]. Section 5.4.1 further details the use case introduced in Section 5.3, namely the evaluation scenario and the PreScore and Score functions. Section 5.4.2 presents the evaluation tests and results.

5.4.1 Evaluation Scenario

Figure 5.5 presents the infrastructure available to run the use case application detailed in Figure 5.4c. It is composed of "Block Face 1" of the "City Block 1", which contains five smart lampposts (L1 to L5), each with its set of sensors (L1 is composed of noise and light sensors, L2 a camera, L3 a noise sensor, L4 a camera and a light sensor, and L5 a noise sensor and a camera).

[6] https://www.tuvsud.com/en/press-and-media/2024/january/smart-cities-and-intelligent-lighting

Table 5.1: Block face Node specification for the Smart City use case infrastructure.

Metrics		Nodes				
		L1	L2	L3	L4	L5
Noise	Accuracy (dB)	3		0,5		1,5
	Response time (s)	3		2,5		2
	Resolution (dB)	0,1		0,1		0,1
	Power consumption (W)	0,84		0,18		0,23
Video	Resolution (MP)		12		6	4
	Dynamic range (dB)		120		120	120
	Power consumption (W)		9,55		5,6	6,6
Light	Accuracy (%)	5			5	
	Power consumption (W)	2			1,5	
Location		B1.F1				
Radio power consumption (W)		2	1	1,5	1,5	2
Latency (ms)	L1	NA	36	36	35	35
	L2	36	NA	35	33	33
	L3	36	35	NA	34	33
	L4	35	33	34	NA	32
	L5	35	33	33	32	NA

Table 5.1 presents the lampposts sensor specification and the communication latency between them. The sensor specification corresponds to existing commercial sensors (netvox R718PA7, Rika RK300-06B, Bosch 5100i, Hanwha QND-8080R, FortiCamera Series, PHOTASGARD AHK F-I Q and Thermokon LI65+ LRW). The latency was specified within the range for cellular communications reported by the Portuguese communications authority *Autoridade Nacional de Comunicações* (ANACOM) in its annual report[7].

The characteristics of the Nodes presented in Table 5.1 are evaluated by QOIA against the quality requirement and scheduling optimisation objectives of Equations (5.6a) to (5.6e). The reference values for QoD of Equation (5.6a) are $A_{ref} = 0.5$, $RT_{ref} = 2$ and $R_{ref} = 0.1$ as presented in Section 5.3.3.

Algorithms 1 and 2 detail the $min(Pc)$ optimisation of Equations (5.6a) to (5.6e) for the PreScore ($Q_{PS.function}$) and Score ($Q_{S.function}$) stages, respectively. We explain these algorithms using the "Video" application component and the Nodes specified in Table 5.1 as an example.

$Q_{PS.function}$ calculates the power cost for all filtered Nodes in the cluster (V, which for the "Video" application component contains Nodes L2, L4, and L5, the ones that meet the quality requirements of the Filter stage specified in Equation (5.6c) - $DR \geq 120 \wedge R \geq 2 \wedge L = BI_w.BF_z \wedge \text{Lat}_{validator} < 60 \wedge \text{Lat}_{LampCont_k} < 40 \; \forall k \in \{1,\ldots,5\}$), considering the current cluster state. It does that by iterating over all the filtered Nodes V (cf. line 6) and over all the metrics defined in the quality configuration file of the Pod under scheduling ($Q_{PS.metrics}$ in line 7, which for the "Video" application component contains metsenspw: "video/pw-cost" and metrapw: "radio/pw-cost"). It checks for each Node v of the cluster (V) if it contains the quality metric $q_{ps.metric}$ defined in the

[7] https://www.anacom.pt/render.jsp?contentId=1776583

Algorithm 1 Power Cost Minimization Objective at the PreScore Stage.

Require: $V \neq \emptyset$
Require: $Q_{PS} \neq \emptyset$
1: $Cost \leftarrow 0$
2: $radioCost \leftarrow 0$
3: $PSout \leftarrow \emptyset$
4: $MaxCost \leftarrow MaxFloat64$
5: $CompsToComm \leftarrow getAppCompsToComm(pi_{Namespace}, pi_{AppGroupData})$
6: **for** $v \in V$ **do**
7: **for** $Q_{ps.metrics} \in Q_{PS.metrics}$ **do**
8: **if** $q_{ps.metric.value} \in dom(v.labels)$ **then**
9: **if** $q_{ps.metric.key} ==$ "$metrapw$" **then**
10: $radioCost \leftarrow v.labels[q_{ps.metric.value}]$
11: **else**
12: **if** $q_{ps.metric.key} ==$ "$metsenspw$" **then**
13: $Cost \leftarrow Cost + v.labels[q_{ps.metric.value}]$
14: **end if**
15: **end if**
16: **else**
17: $Cost \leftarrow MaxCost$
18: **end if**
19: **end for**
20: **if** $Cost == MaxCost$ **then**
21: $append(PS_{out}, (v.name, Cost))$
22: **else**
23: **for** $compToComm \in CompsToComm$ **do**
24: **if** $compToComm \notin v.Pods$ **then**
25: $Cost \leftarrow Cost + radioCost$
26: **end if**
27: **end for**
28: $append(PS_{out}, (v.name, Cost))$
29: **end if**
30: **end for**
31: **return** PS_{out}

configuration file (cf. line 8). If that is not the case, that Node gets the maximum node cost (cf. line 17, which is not reached in our setup because all the Nodes are correctly configured according to $Q_{PS.function}$). Otherwise, it checks if the key of the metric $q_{ps.metric}$ under analysis is "metrapw" (cf. line 9), which is the key associated with the radio power cost. If that is the case, we get the value of the Node label defined by $q_{ps.metric.value}$ (label is "radio/pw-cost" in our example, cf. line 13 of Listing 5.4) and save that value in the variable radioCost (cf. line 10, with radioCost equal to 1 for Node L2, 1.5 for Node L4 and 2 for Node L5). If the key of the metric under analysis is not "metrapw", we check if the key contains the text "metsenspw" (cf. line 12), which is the key associated with the sensor power cost. If that is the case, we get the value of that key ($q_{ps.metric.value}$, which corresponds to "video/pw-cost" for the "Video" application component) and get the label of Node v defined by that value. The value of that Node label is added to the variable Cost (cf. line 13).

After checking all the metrics in the Node v under analysis against all the metrics defined in the quality configuration file of the Pod under scheduling (cf. line 19), the variable "Cost" contains the power cost of all relevant sensors (the ones the user defined in the quality configuration file with a key that contains the text "metsenspw", which is "video/pw-cost" for the "video" application component and results in a Cost of 9.55 for Node L2, 5.6 for Node L4 and 6.6 for Node L5). At this stage, we check if the radio cost also needs to be added to the total Node power cost (cf. lines 23 to 27). This is done by checking the components with which the Pod under scheduling communicates (cf. line 23), where for the "Video" application component *CompsToComm* contains the "Lamp Control" and "Validator" (cf. Figure 5.4c). If the Pods associated with such components are already running in the Node v under analysis (cf. line 24), the radio cost is not considered. Otherwise, it is considered. Following our "Video" application component example, and considering that the components in *CompsToComm* ("Lamp Control" and "Validator") are already deployed. "Lamp Control" is deployed in Nodes L1 to L5, and the "Validator" is in Node L4. With this cluster setup, Node L2 for the "Video" component includes the communication cost for five components ("Lamp Control" in Nodes L1, L3, L4, and L5, plus "Validator" in Node L4), Node L4 includes the communication cost for four components ("Lamp Control" in Nodes L1, L2, L3, and L5) and Node L5 includes the communication cost for five components ("Lamp Control" in Nodes L1, L2, L3, and L4, plus "Validator" in Node L4). After checking that, $Q_{PS.function}$ sets PS_{out} (cf. line 28), which is a map with the key equal to the name of the Node v under analysis and the value equal to its total power cost. At the end of the PreScore state, each Node in the set V has an entry in PS_{out}. Considering the "Video" component example we are following, PS_{out} is equal to {"L2": $14.55 (= 9,55 + 5 \times 1)$, "L4": $11.6 (= 5,6 + 4 \times 1,5)$ and "L5": $16.6 (= 6,6 + 5 \times 2)$}.

In the Score stage (Algorithms 2), $Q_{S.function}$ gets the power cost for the Node v_j under analysis from PS_{out} (cf. line 3, where for the "Video" component example we are following PS_{out} contains {"L2": 14.55, "L4": 11.6 and "L5": 16.6}). The Score function runs for each Node in the set V, so v_j is the Node from set V being scored at run j. It then calculates the Node power cost inverse normalisation considering the maximum and minimum power costs available in PS_{out} (cf. lines 4 to 16, where in our example *max* is 16.6 and *min* is 11.6). $Q_{S.function}$ uses inverse normalisation

to score with higher values the Node with the lowest power cost (K8S requires score values to be normalised). The normalisation result is returned at the end of the Score stage (cf. line 17). Taking the example of the Video component used in the PreScore stage, $Q_{S.function}$ scores L2 with $0.41(= 1 - \frac{14.55-11.6}{16.6-11.6})$, L4 with $1(= 1 - \frac{11.6-11.6}{16.6-11.6})$ and L5 with $0(= 1 - \frac{16.6-11.6}{16.6-11.6})$. After calling the Score stage for all the Nodes v, K8S binds the Pod under scheduling to the Node from the set V with the highest score (L4 in the "Video" application component example).

Algorithm 2 Power Cost Minimization Objective at the Score Stage.

```
1:  max ← minPossibleValue
2:  min ← maxPossibleValue
3:  Cost ← PS_out[v_j.name]
4:  for ps_out.value ∈ PS_out do
5:      if ps_out.value > max then
6:          max ← ps_out.value
7:      end if
8:      if ps_out.value < min then
9:          min ← ps_out.value
10:     end if
11: end for
12: if max == min then
13:     costNormalized ← 1
14: else
15:     costNormalized ← (1 − (Cost−min)/(max−min))
16: end if
17: return costNormalized
```

Applying Algorithms 1 and 2 to the described evaluation scenario, considering its quality requirements, scheduling optimisation objective, and that the scheduling follows the topological order of the application graph (ordering is handled by the Diktyo, which uses topological sort to order the Pods in the scheduling queue), QOIA assigns one Lamp Control application component per block face Node, the Video, Light and Validator components to Node L4, and the Noise component to Node L3. This scheduling meets the requirements and guarantees a power cost of 17.28W, the lowest considering the $min(Pc)$ optimisation.

5.4.2 Evaluation Tests

We tested QOIA at scale by considering two approaches: 1) An increase in the number of blocks and block faces (Section 5.4.2.1); and 2) an increase in the number of Nodes in a single block face (Section 5.4.2.2).

In the first approach, there is an increase in the number of Nodes in the Cluster, as well as in the number of smart lighting applications (one per block face). This evaluates the scalability regarding the number of application components to schedule and recover, as well as the number of Nodes to consider in the Filter stage of the scheduling framework. At the Pre-Score and Score stages, this setup always considers the Nodes of a single block face. This happens because the

components of each smart lighting application are associated with a block face, whose Nodes are filtered in the Filter stage.

In the second approach, we tackle the scalability of Nodes to consider in the Pre-Score and Score stages by replicating the Nodes in a block face and scheduling a single Smart Lighting application considering the extended block face. In this approach, the scalability of the application components is minimal. We just added new lamp control components to the Smart Lighting application to guarantee that there is a lamp control component per Node. The use of a single Smart Lighting application is intentional because our focus in this setup is not on the scalability of application components, which is evaluated in the first approach.

In both approaches, we tested scheduling and recovery. For each scalability configuration of each approach, we initiate the scheduling tasks by submitting a single file containing the complete set of smart lighting applications for the setup under evaluation. The Recovery evaluation is initiated when scheduling is completed and is triggered by introducing a 50 ms delay in the communication link between Nodes L3 and L4 of all block faces under evaluation. This makes Node L3 non-compliant with the Noise QoS requirements on all smart lighting applications under test. We repeated the scheduling and recovery evaluation of each setup 50 times. At each repetition, we used a K8S Watcher to listen to Pod update events and measure:

1. The time K8S takes to add an application component to the scheduling queue after it is submitted for scheduling. We measured this by calculating the difference between the time a Pod is updated with an event of type "Added" and the time we submit the set of Smart Lighting applications for scheduling.

2. The time QOIA Quality-aware Scheduler takes to bind an application component to a Node after it is added to the scheduling queue. We measured this by calculating the difference between the time a Pod is updated because it was associated with a Node and the time K8S added the Pod to the scheduling queue.

3. The time QOIA Runtime Watcher takes to evict an application component after its quality requirements are no longer met. We measured this by calculating the difference between the time a Pod is updated with a mark for eviction and the time we introduced the 50ms delay in the Diktyo network topology.

4. The time K8S takes to add an application component to the scheduling queue after it is evicted by QOIA Runtime Watcher. We measured this by calculating the difference between the time a Pod is updated with an event of type "Added" and the time it was marked for eviction.

5. The CPU and memory used by QOIA Quality Aware Scheduler and QOIA Runtime Watcher during scheduling and recovery. We have used a Python script that utilises the "psutil" library to measure the CPU and memory usage of the target processes with precision.

As noted at the end of Section 2.3.2, there is no State-of-the-Art proposal that directly compares with QOIA. Thus, we assessed the overhead added by QOIA to handle configurable quality

requirements and recovery by comparing the scheduling performance with Diktyo. Diktyo is the K8S scheduling plugin we used to support the QoS requirements (Section 5.2.2.1). This overhead assessment is only performed for scheduling because Diktyo does not support recovery.

Our evaluation testbed comprises a realistic IoT installation composed of resource-constrained and resourceful devices. The Nodes with cameras (cf. L3, L4 and L5 in Figure 5.5) are resourceful Nodes. Typically, cameras require higher processing capabilities than those available on microcontrollers for handling high frame rates and executing image processing algorithms. The remaining Nodes (cf. L1 and L3 in Figure 5.5) are resource-constrained Nodes composed of microcontrollers. The resourceful Nodes are emulated using KWOK[8], a CNCF project that provides a tool for emulating K8S Nodes and testing schedulers at scale. For the resource-constrained Nodes, we used IoTNetEmu [79] to emulate embServe-based devices (the same emulator we used in Chapter 4 to evaluate FITA).

Using this emulation strategy does not impact the assessment of QOIA performance, i.e., the time it needs to bind a Pod to a Node and the time it needs to adapt at runtime when the quality requirements are no longer met. As highlighted in Figure 5.2, both scheduling (that happens between labels 2 and 3) and recovery (that happens between labels 4 and 3) are tasks performed at the K8S Control Plane. The two tasks within the orchestration flow that would be affected by this strategy are not relevant for our evaluation: (1) the deployment of Pods after binding is a native K8S and FITA task that happens after QOIA finishes its scheduling tasks (in grey in Figure 5.2), so it is not relevant for QOIA performance assessment; (2) The measurement and reporting of quality metrics, which, as indicated in the beginning of this chapter, is outside the scope of this Thesis and should be addressed in a dedicated effort focused on measurement and reporting optimisation.

The K8S distribution used in our setup is microK8S[9], which handles a cluster setup on a NUC14RVK with 93GB of RAM and 22 CPUs, i.e., a high-performance Edge computing device that could be managing several city zones or districts. The cluster is composed of: (1) the control plane Node that runs QOIA's scheduler, QOIA's Runtime Watcher, FITA and the KWOK controller; and (2) the virtual KWOK and IoTNetEmu Nodes that represent the block face lampposts and are created for each test setup.

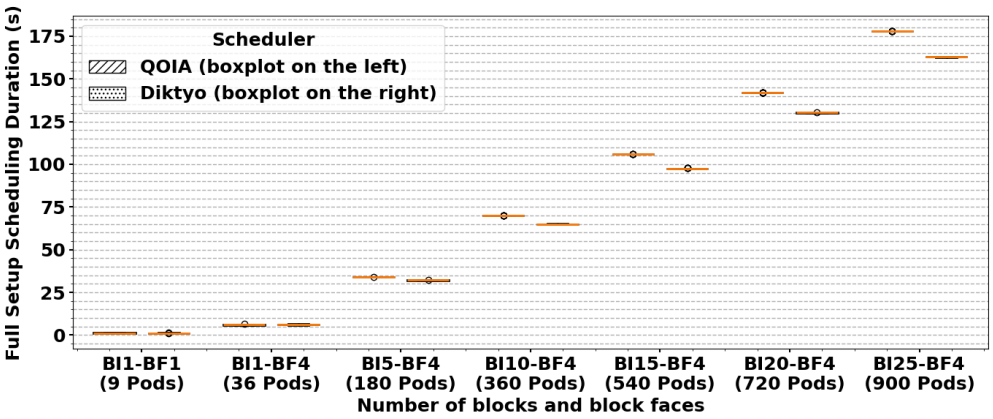
5.4.2.1 Scaling the Number of Blocks and Block Faces

In the approach that scales with the number of blocks and block faces, the evaluation begins with a single block face (BI1-BF1) and expands to 25 full blocks (BI25-BF4). This represents a total of 500 Nodes (300 resourceful Nodes and 200 resource-constrained Nodes) and 100 smart lighting applications (900 application components/Pods).

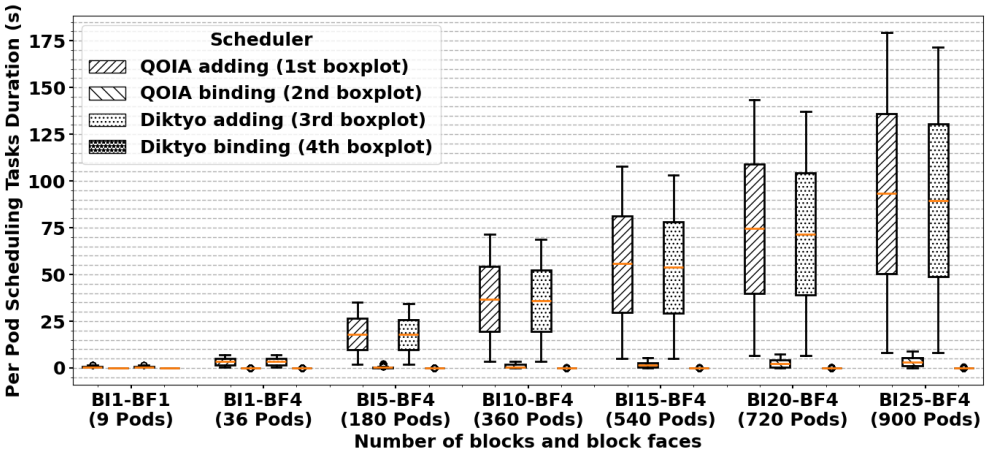
Scheduling

Figure 5.6 presents the scheduling evaluation for QOIA and Diktyo, which is represented by: (1) the scheduling duration for setting the complete group of applications that compose the set of blocks (BI) and block faces (BF) under evaluation (Figure 5.6a); (2) and the per-Pod duration of

[8] https://kwok.sigs.k8s.io/
[9] https://microk8s.io/



(a) Scheduling duration for the complete setup of block faces.



(b) Per Pod adding and binding duration.

Figure 5.6: QOIA and Diktyo scheduling evaluation when scaling the number of blocks and block faces.

the steps that compose the scheduling task (Figure 5.6b), i.e., the time K8S needs to add a Pod to the scheduling queue and the time QOIA and Diktyo take to bind a Pod to a Node.

The scheduling duration in Figure 5.6a is calculated by measuring the durations from when K8S adds the first application component to the scheduling queue to when QOIA or Diktyo bind the last application component to a Node. The results show that the scheduling duration for the complete setup increases approximately linearly with the number of blocks. Figure 5.6b provides further information for the analysis. The results show that the majority of the scheduling time is spent adding the Pod to the scheduling queue. The lowest values on the boxplots of Figure 5.6b represent the add or bind duration of the Pods that are dispatched first, and the highest represent the Pods that are dispatched last. This means that the full setup scheduling duration is determined by the time K8S takes to add the last Pod to the scheduling queue. The shape of the boxplot also indicates an approximately uniform distribution, which is consistent with a linear dependence.

Comparing the K8S queue add duration with the time QOIA and Diktyo take to bind the Pod

to a Node, the former is negligible. Nevertheless, the maximum Diktyo's binding time is between 0.094 and 0.599 seconds. QOIA's maximum binding time is between 0.096 seconds and 9.06 seconds for the BI1-BF1 and BI25-BF4 setups, respectively. QOIA binding time also has a higher variability. The difference between the maximum and minimum binding time is 0.095 seconds for the BI1-BF1 setup and 9.059 seconds for the BI25-BF4 setup. This means that in QOIA, there are more Pods accumulated in the scheduling queue waiting to be bound to a Node. The ones that are dispatched first from the scheduling queue are bound in nanoseconds, independently of the setup, but in setups with high load, the last Pod to be dispatched takes seconds to be bound to a Node. This binding overhead and variability introduced by QOIA, when compared with Diktyo, are related to the K8S API requests that QOIA performs to load the quality configuration files hosted in the cluster as configMaps. The API server is already handling a high number of simultaneous requests to create Pods, as reflected by the significant amount of time the K8S server takes to add the Pods to the scheduling queue. QOIA is adding more load to the server by requesting the configMaps. Additionally, in the setup with QOIA, the Runtime Watcher is also running, which introduces some load to the K8S API server due to the instantiation of informers that receive notifications about Node updates, and Pod creation and updates.

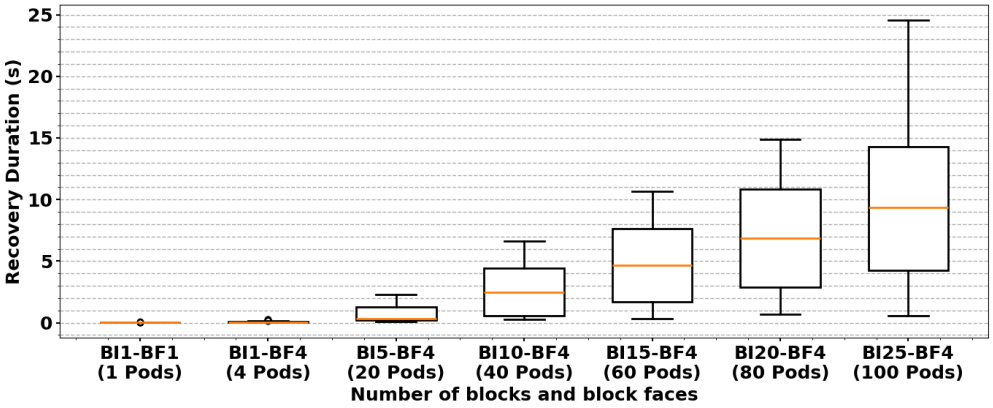
Despite the overhead introduced by QOIA, we should consider it in context. QOIA's maximum binding time of 9.06 seconds is obtained when simultaneously scheduling 900 Pods, which is an acceptable value for such a highly demanding request. For more fair requests, such as the creation of 36 simultaneous Pods, the maximum binding time is 0.105 seconds. These are acceptable binding durations considering the added value introduced by QOIA, i.e., the capability of handling user-defined quality requirements and scheduling optimisation objectives, as well as the recovery functionality added by the Runtime Watcher. Additionally, the majority of the scheduling time is used by K8S to add the Pods to the scheduling queue. This means that to improve the scheduling time, we should focus first on reducing the time taken to add the Pods to the queue. The objective should be to reach a more balanced distribution between the add and bind steps with the aim of reducing the total scheduling time.

Recovery

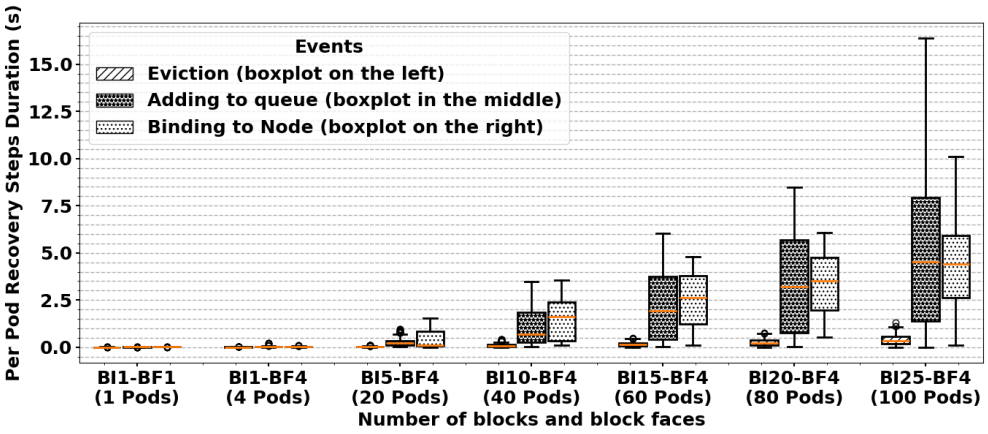
Figure 5.7 presents the recovery evaluation. Figure 5.7a presents the recovery duration for each test setup, which includes a maximum of 100 recovered Pods in the setup with the highest load (BI25-BF4). Figure 5.7b presents the duration of the steps that compose the recovery task, i.e., eviction of the Pod due to quality requirements non-compliance, addition of the evicted Pod to the scheduling queue and binding of the evicted Pod to a new Node.

The results in Figure 5.7b show that QOIA's Runtime Watcher is capable of detecting non-compliances and evicting the non-compliant Pod with a maximum value between 0.016 and 1.329 seconds for the BI1-BF1 and BI25-BF4 setups, respectively. Only the BI25-BF4 setup, which comprises 100 simultaneous non-compliances, has a maximum eviction duration higher than one second.

Looking at the total recovery duration in Figure 5.7a and considering the results in Figure 5.7b, we can conclude that the total recovery is mainly influenced by the sum of the time K8S takes to



(a) Per Pod recovery duration.



(b) Per Pod eviction, adding and binding Duration.

Figure 5.7: QOIA recovery evaluation when scaling the number of blocks and block faces.

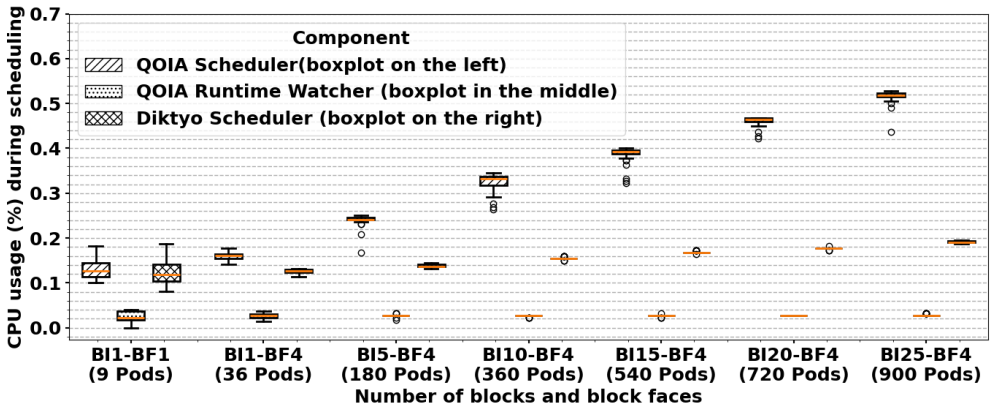
add the evicted Pod to the scheduling queue plus the time QOIA Quality-aware Scheduler takes to bind it to a Node.

Considering the use case under evaluation, Smart Lighting, QOIA's total recovery time is acceptable up to the BI1-BF4 setup, which includes four simultaneous faults. At this setup, the maximum recovery time is 0.286 seconds. In the next setup, which contains 20 simultaneous faults, the recovery time is already above one second. This may be suitable for some IoT and IIoT use cases, but it is not compliant with the specified Smart Lighting use case when instantiated in a setup that can experience 20 or more simultaneous faults. Although this is an unlikely and extreme scenario, improving QOIA recovery time to support it requires reducing both the time K8S takes to add the evicted Pods to the scheduling queue and also the time QOIA Quality-aware Scheduler takes to bind the Pods to a Node.

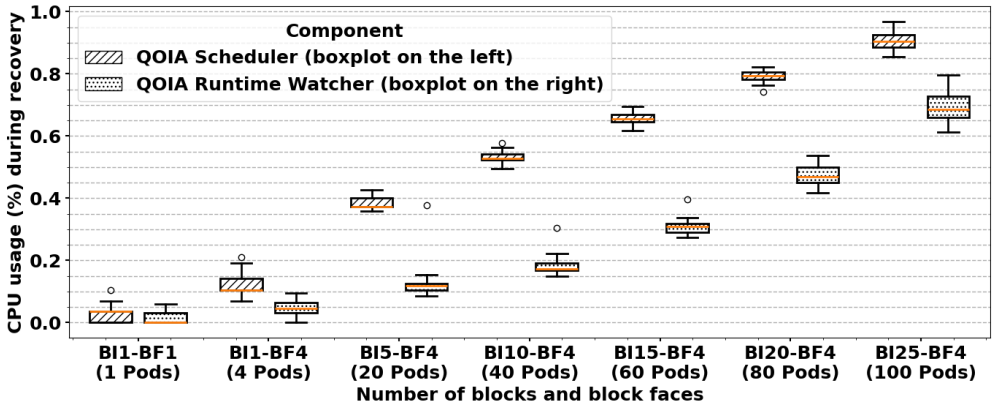
Resource Usage

Figure 5.8 shows the CPU usage during scheduling (Figure 5.8a) and recovery (Figure 5.8b).

During scheduling, the QOIA Quality-aware Scheduler CPU usage increases with the cluster load. The same phenomenon occurs with Diktyo, but at a considerably lower rate, resulting in a



(a) QOIA and Diktyo CPU usage during scheduling.

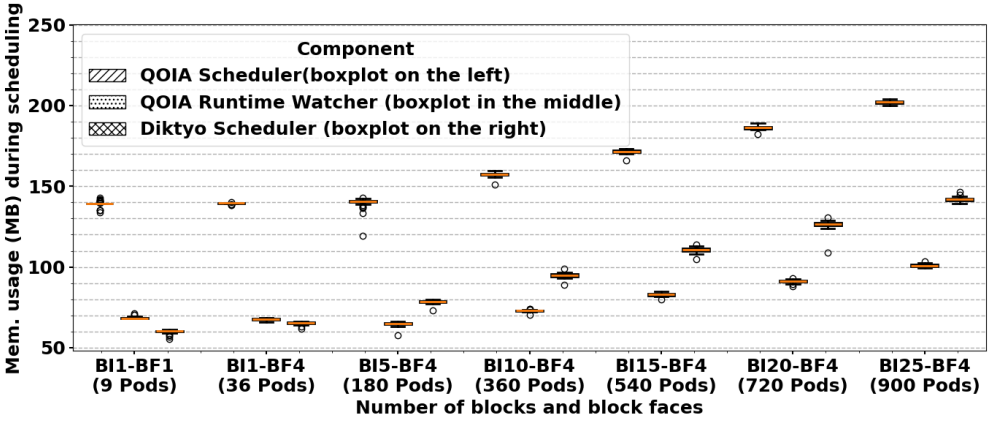


(b) QOIA CPU usage during recovery.

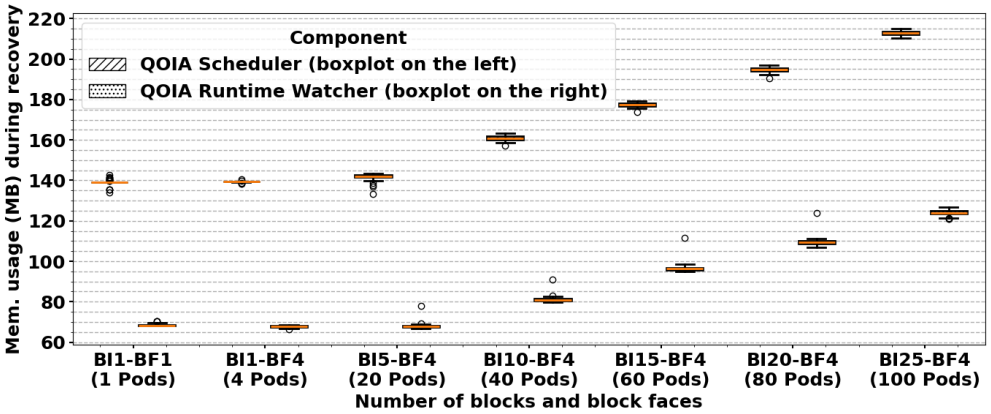
Figure 5.8: CPU usage during scheduling and recovery when scaling the number of blocks and block faces.

difference of 0.33% in the BI25-BF4 setup. This happens because of QOIA's extra work of assessing the quality requirements of the Pods. Nevertheless, both schedulers have a low CPU usage for the setups under analysis. Regarding the QOIA Runtime Watcher, the CPU usage remains low and unaffected by the increase in cluster load. This happens because, at this stage, there are no quality non-compliances to deal with.

During recovery, both the QOIA Quality-aware Scheduler and the QOIA Runtime Watcher increase their CPU usage in proportion to the cluster load. This is expected because recovery involves both detecting quality non-compliances and evicting such Pods (a task performed by the Runtime Watcher) and scheduling the evicted Pods (a task performed by the Quality-aware Scheduler). In the setup with the highest load (BI25-BF4), the QOIA Quality-aware Scheduler has a median CPU usage of 0.95% and the QOIA Runtime Watcher of 0.67%, which are low CPU usages. The highest CPU usage of the Quality-aware Scheduler is justified by the extra work performed during scheduling when compared to eviction.



(a) QOIA and Diktyo memory usage during scheduling.



(b) QOIA memory usage during recovery.

Figure 5.9: Memory usage during scheduling and recovery when scaling the number of blocks and block faces.

Comparing the results of Figure 5.8a with Figure 5.8b, we can see that the QOIA Quality-aware Scheduler has more CPU usage during recovery. This is justified by the time the QOIA Quality-aware Scheduler is waiting for the K8S API server to add the Pods to the scheduling queue. This duration is longer during scheduling than during recovery. During this waiting period, the QOIA Quality-aware Scheduler is not performing work, which reduces its CPU usage throughout the entire scheduling period.

Figure 5.9 shows the memory usage of QOIA Quality-aware Scheduler, Diktyo scheduler and QOIA Runtime Watcher during scheduling (Figure 5.9a) and recovery (Figure 5.9b). As expected, the memory usage increases for all of them with increasing Cluster load, both during scheduling and recovery.

The Diktyo scheduler loads the network topology into memory, which increases with the number of Nodes in the cluster. QOIA Quality-aware Scheduler also loads the network topology, which is complemented with the quality configuration of the Pods being scheduled that have already passed the Pre-Filter stage and are not yet bound to a Node. That is the reason behind the

QOIA Quality-aware Scheduler's higher memory usage when compared to Diktyo. This difference peaks at the BI1-BF4 setup with 80MB, reducing slowly for larger setups. The reason this difference is larger for smaller setups is that as cluster load increases, the network topology size also increases and begins to contribute more to total memory usage. QOIA Runtime Watcher increases memory usage with cluster load due to the information it loads on boot (see Figure 5.3), including the names and namespaces of Pods, Nodes, ConfigMaps, AppGroups, and network topology. The Runtime Watcher just loads into memory the complete Pod, Node, configMap, etc., of the Pod under eviction. That is why it consumes less memory than Diktyo and QOIA Quality-aware Scheduler. Between the BI1-BF1 and BI5-BF4 setups, there is a slight decrease of 1.5MB in QOIA Runtime Watcher memory usage with increasing cluster load. This small variation (2.3% of the total memory usage at BI5-BF4) is likely related to the measurement procedure rather than to a meaningful difference in memory consumption in these setups. At each repetition, we collect a memory usage sample every 0.1 seconds and then average the set at the end of the recovery task. Considering this, at the BI1-BF1 setup, we collect one memory sample at each repetition (the one collected at eviction start since the total recovery time is below 0.1 seconds). At the BI1-BF4 setup, we collect two samples at each repetition (total recovery time is below 0.2 seconds). At the BI5-BF4 setup, we collect between ten and twenty samples at each repetition (total recovery time considering the interquartile range is between approximately 1 and 2 seconds).

Despite increasing with the cluster load, the memory usage remains relatively low for all components (schedulers and Runtime Watcher), both during scheduling and recovery.

5.4.2.2 Scaling the Number of Nodes in a Single Block Face

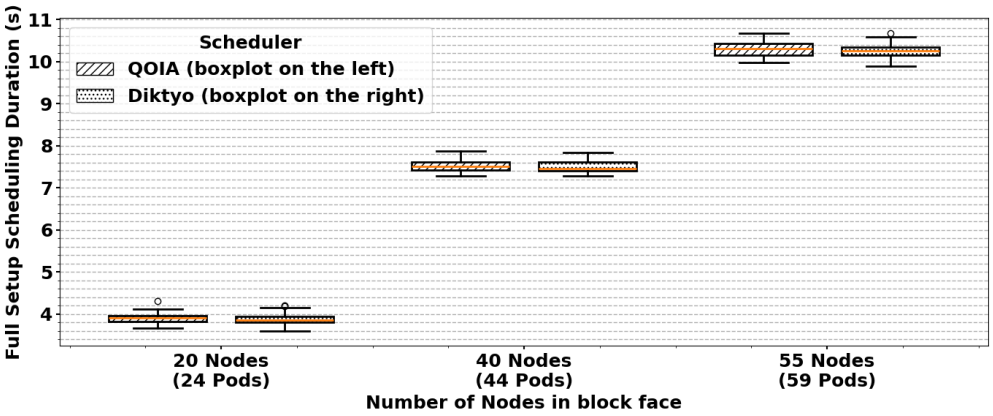
In the approach that scales with the number of Nodes in a single block face, we increase them by replicating all the Nodes of the block face. The evaluation starts with 20 Nodes (replicated four times the Nodes in the block face) and ends with 55 Nodes (replicated 11 times the Nodes in the block face). The evaluation scales up to 55 Nodes because this is the maximum number of replications supported in a single zone by Diktyo's network topology custom resource definition.

Scheduling

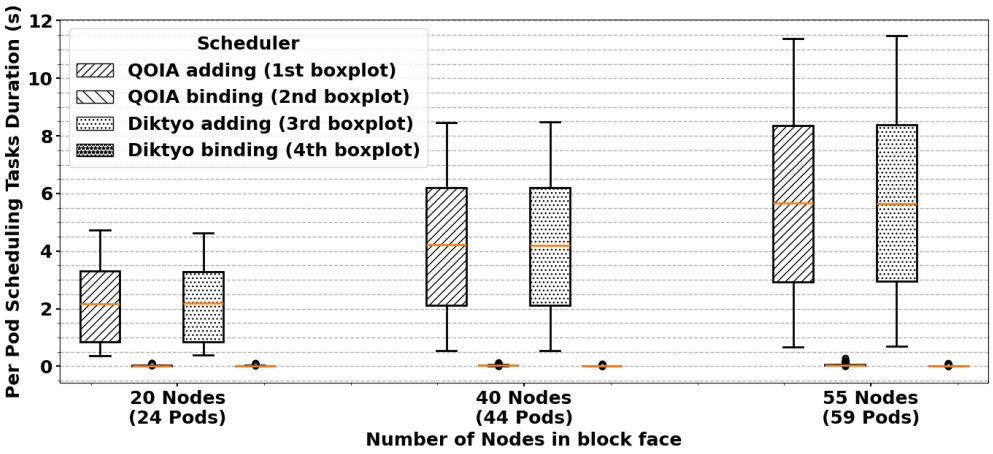
Figure 5.10 presents the scheduling evaluation when scaling the number of Nodes in a single block face. Figure 5.10a presents the scheduling duration for the complete application of each test setup. Figure 5.10b presents the duration of the steps that compose the scheduling task.

Considering the results of Figure 5.10a, the scheduling duration for the complete application has a behaviour approximately linear. With QOIA and Diktyo having approximately the same scheduling duration. Figure 5.10b adds more information to this analysis. The add time for QOIA and Diktyo is approximately the same, and is the step that determines the scheduling duration of each Pod, because compared with it, the binding duration is negligible. This is the reason behind the behaviour in Figure 5.10a.

Looking at the binding duration in Figure 5.10b, QOIA's maximum binding time is 0.11 and 0.275 seconds for the setup with 20 Nodes and 55 Nodes, respectively. Diktyo's maximum binding time is 0.098 and 0.099 for the same setups. These results reveal that QOIA user configurable



(a) Scheduling duration for the complete Smart Lighting application.



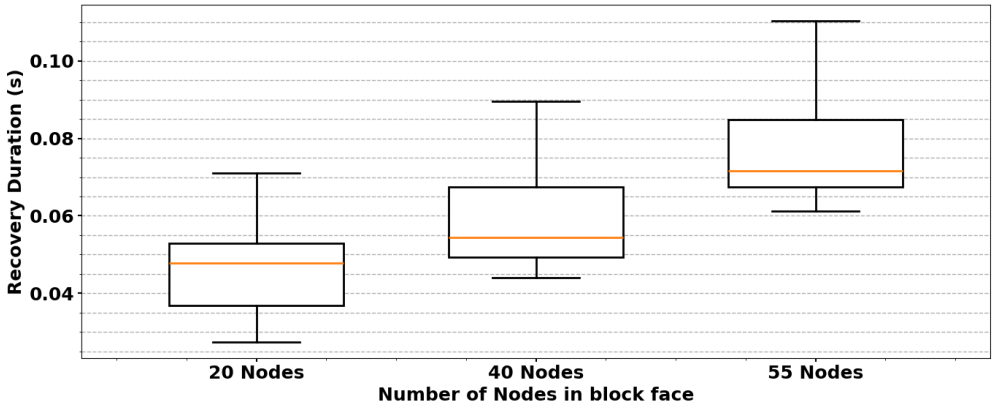
(b) Per Pod adding and binding duration.

Figure 5.10: QOIA and Diktyo scheduling evaluation when scaling the number of Nodes in a block face.

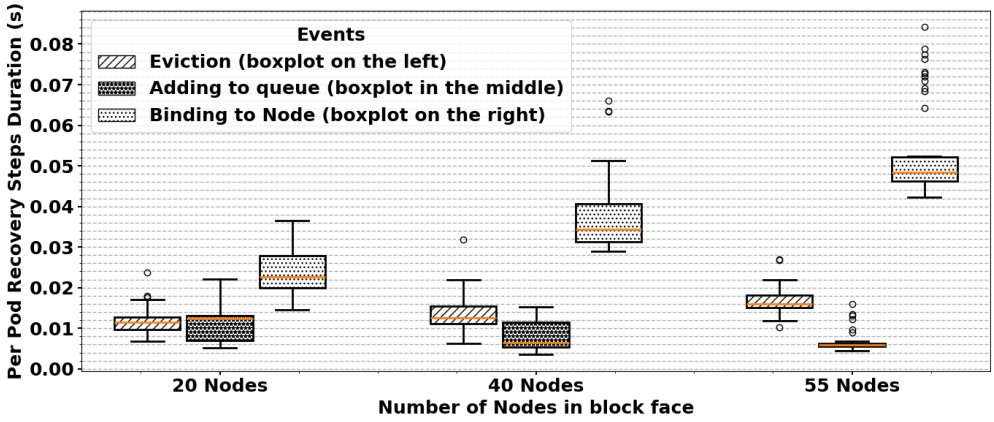
quality requirements and optimisation objectives, along with its recovery mechanisms, add an overhead of 57% on the maximum binding time of the setup with 55 Nodes. Despite such overhead, QOIA binding time for the setups analysed in this section is adequate for the majority of IoT and IIoT use cases, like the one under evaluation (Smart Lighting).

Comparing with the scheduling results in Section 5.4.2.1, it is possible to conclude that the increase in Nodes to consider in the Pre-Score and Score stages of the scheduling framework does not significantly impact the scheduling time. In fact, as we saw in Section 5.4.2.1, the binding bottleneck in QOIA is on the configMap read requests, which occur at the Pre-Filter stage. The results show that the single block face evaluation approach does not stress the K8S API server enough for such requests to be delayed.

In conclusion, if the scale of scheduling requests does not over-stress the K8S API server, the binding overhead introduced by the QOIA Quality-aware Scheduler is negligible, as it still produces short binding durations. In the setups of Section 5.4.2.1 that stress the K8S API server,



(a) Per Pod recovery duration.



(b) Per Pod eviction, adding and binding duration.

Figure 5.11: QOIA recovery evaluation when scaling the number of Nodes in a block face.

QOIA's Quality-aware Scheduler overhead is more pronounced. Nevertheless, the complete scheduling duration is still primarily defined by the time K8S API server takes to add the Pods to the scheduling queue. Improving such times has the potential to reduce the scheduling duration not only of high-demanding scenarios but also of more fair scenarios, such as those that do not stress the K8S API server.

Recovery

Figure 5.11 presents the recovery evaluation when scaling the number of Nodes in a single block face. Figure 5.11a presents the recovery duration for each test setup, which includes a maximum of 55 Nodes in the setup with the highest load. Figure 5.11b presents the duration of the steps that compose the recovery task.

The results in Figure 5.11a show that the recovery of a single failing Pod in clusters of 55 Nodes is, at most, 0.11 seconds, which is within acceptable values. Figure 5.11b shows that under the tested conditions, adding the evicted Pod to the scheduling queue is no longer the bottleneck, since the majority of time is used by QOIA's Quality-aware Scheduler to bind the Pod to a Node. This is the opposite of the behaviour analysed in the results of Section 5.4.2.1, and reflects the

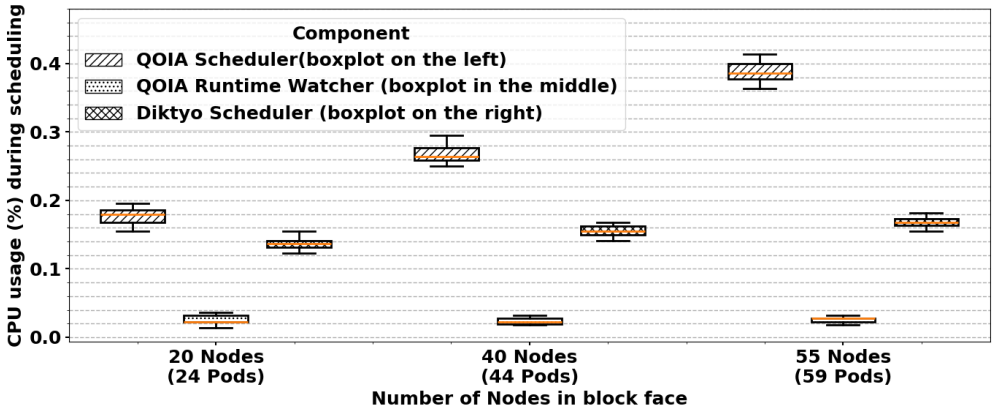
reduced load the recovery of a single Pod adds to the K8S API server. The small increase in the duration of the binding time - from 0.037 to 0.084 seconds, considering the maximum values - is related to the increase in the number of Nodes to analyse in the Filter, Pre-Score, and Score stages. Nevertheless, such an increase of 0.047 seconds is not significant for most IoT and IIoT use cases, including the one under analysis. Considering the time taken to add the evicted Pod to the scheduling queue, we observe a counterintuitive behaviour, as it decreases with an increase in cluster load. Looking at the maximum and minimum durations of each setup, we observe 0.022 seconds and 0.005 seconds for the setup with 20 Nodes, 0.015 seconds and 0.004 seconds for the setup with 40 Nodes, and 0.016 seconds and 0.005 seconds for the setup with 55 Nodes. The differences are minimal: 0.006 seconds when comparing the maximum adding duration for the setup with 20 Nodes against the setup with 55 Nodes; and 0.001 seconds when comparing the minimum duration for the setup with 20 Nodes against the setup with 40 Nodes. Such a minimum difference is likely associated with how K8S manages the work queues and REST API calls of the non-deterministic reconciliation tasks executed between eviction and re-queue for scheduling (Pod termination by the Kubelet, deletion of Pod object by the K8S API server, creation of a new Pod by the Deployment controller and add to queue by the K8S scheduller), as well as how they are influenced by external events. Considering this, we argue that we should consider the duration of adding the evicted Pod to the scheduling queue on the tested cluster loads under our test conditions to be within the measured range of 0.022 seconds and 0.004 seconds. Further understanding the different statistical distributions of each setup within this range requires a detailed analysis and evaluation of the K8S reconciliation tasks performed between eviction and re-queue, which is outside the scope of this Thesis because it does not influence the analysis of QOIA's complete recovery duration.

Resource Usage

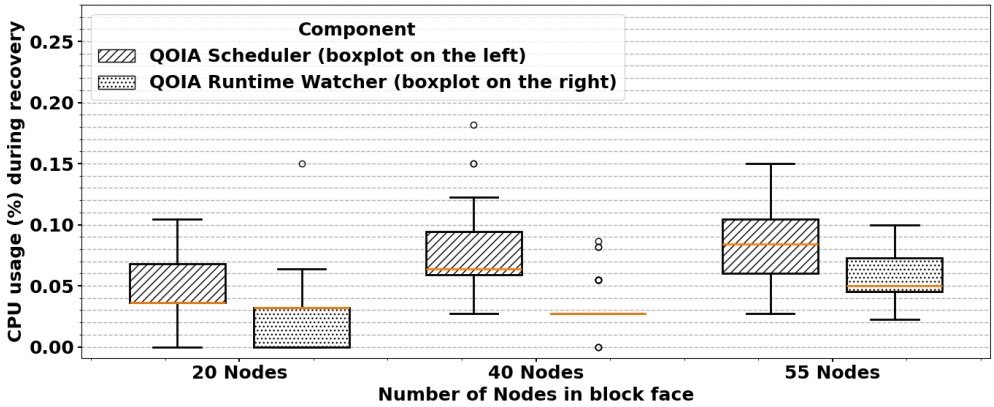
Figure 5.12 provides QOIA Quality-aware Scheduler, Diktyo scheduler and QOIA Runtime Watcher CPU usage during scheduling (Figure 5.12a) and recovery (Figure 5.12b). During scheduling, both the QOIA Quality-aware Scheduler and the Diktyo scheduler experience an increase in CPU usage as the number of Nodes in the block face increases. QOIA exhibits a CPU usage difference of 0.205% between the setup with 55 Nodes and the setup with 20 Nodes, while Diktyo has a difference of 0.03%. This behaviour is aligned with the one identified in Section 5.4.2.1 and is related to the extra work QOIA performs to bind Pods to Nodes, considering the user-defined quality requirements and optimisation objectives. The QOIA Runtime Watcher CPU usage behaviour is also aligned with the one identified in Section 5.4.2.1, i.e., it remains stable during scheduling and experiences a slight increase during recovery.

In contrast to what was verified in Section 5.4.2.1, the CPU usage during scheduling is higher than during recovery. There are two reasons for this:

1. On one hand, there is a reduction in the time K8S API server takes to add the Pods to the scheduling queue on the single block face approach (Figure 5.6 vs. Figure 5.10). This reduction reduces the amount of time the QOIA Quality-aware Scheduler is waiting for the Pods to be added to the scheduling queue.



(a) QOIA and Diktyo CPU usage during scheduling.



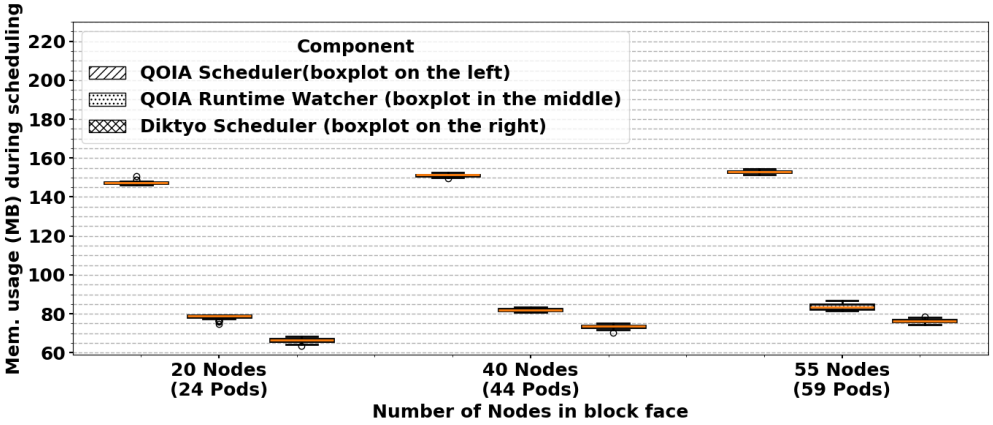
(b) QOIA CPU usage during recovery.

Figure 5.12: CPU usage during scheduling and recovery when scaling the number of Nodes in a block face.

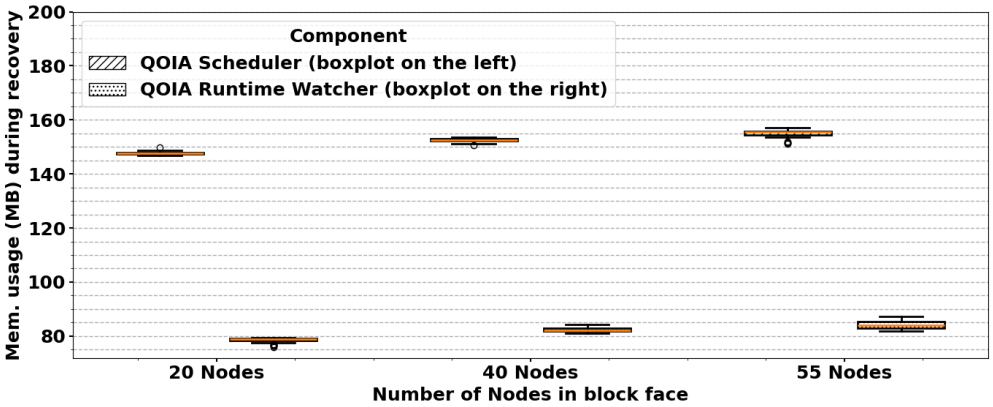
2. On the other hand, the recovery approach using a single block face always deals with only a single non-compliant Pod, whereas the multi-block face approach increases the number of non-compliant Pods with the increase in cluster load. This means there is much less work to do during recovery in the single block face setup.

CPU usage is low for both scheduling and recovery in all evaluated setups for the QOIA Quality-aware Scheduler, QOIA Runtime Watcher, and Diktyo scheduler.

Figure 5.13 provides QOIA Quality-aware Scheduler, Diktyo scheduler and QOIA Runtime Watcher memory usage. Both during scheduling (Figure 5.13a) and recovery (Figure 5.13b), the memory usage of all components increases slightly with the number of Nodes in the block face. This happens because increasing the number of Nodes increases the number of Pods in the cluster, the number of quality configuration files, and the size of the network topology, which affects QOIA Quality-aware Scheduler and QOIA Runtime Watcher memory usage. The increase in network topology size is also why the Diktyo scheduler has a slight increase in memory. Nevertheless, memory usage is low for both scheduling and recovery across all components under evaluation.



(a) QOIA and Diktyo memory usage during scheduling.



(b) QOIA memory usage during recovery.

Figure 5.13: Memory usage during scheduling and recovery when scaling the number of Nodes in a block face.

Table 5.2 provides a brief summary of QOIA's benefits and performance overhead compared to Diktyo. The summary shows QOIA's effective scheduling and life cycle management of IoT applications, meeting configurable quality requirements and scheduling optimisation objectives. These qualitative benefits come with an additional scheduling overhead that depends on the cluster load. Considering the maximum scheduling durations, the overhead varies between 1.45% when simultaneously scheduling 36 Pods in clusters with 20 Nodes and 9.19% when simultaneously scheduling 900 Pods in clusters with 500 Nodes. Life cycle management is only supported by QOIA, providing a recovery from non-compliances that takes a maximum of 0.286 seconds to recover 20 simultaneously failing Pods in a cluster with 100 Nodes and 24.574 seconds to recover 100 simultaneously failing Pods in a cluster with 500 Nodes.

The quantitative results demonstrate that QOIA's qualitative benefits are applicable in IoT and IIoT scenarios with fair cluster loads, even time-demanding ones, as scheduling and recovery are within acceptable values despite the added overhead.

For scenarios with demanding cluster loads, although the measured scheduling and recovery

Table 5.2: Summary of QOIA benefits and performance overhead compared to Diktyo

Evaluation			QOIA	Diktyo
Qualitative		K8S requirements	X	X
		QoS requirements	X	X
		User defined requirements	X	N.A.
Quantitative	Scheduling duration (s)	BI1-BF4: 4 Apps 36 Pods 20 Nodes	6.605	6.511
		BI5-BF4: 20 Apps 180 Pods 100 Nodes	34.043	32.492
		BI25-BF4: 100 App 900 Pods 500 Nodes	178.091	163.104
	Recovery duration (s)	BI1-BF4: 4 Pods 20 Nodes	0.286	N.A.
		BI5-BF4: 20 Pods 100 Nodes	2.265	N.A.
		BI25-BF4: 100 Pods 500 Nodes	24.574	N.A.

durations might be acceptable for some use cases, they are not compliant with time-demanding ones. The majority of time used by QOIA during scheduling and recovery in demanding cluster loads is associated with the time K8S takes to add the Pods to the scheduling queue. Considering this, we can explore reducing the scheduling and recovery duration by improving the latency of adding Pods to the scheduling queue. This improvement could be achieved by:

1. Exploring scaling the K8S control plane horizontally. In this context, the K8S special interest group project kOps[10] is an interesting tool to explore.

2. Fine-tuning the K8S API server resource limits.

5.5 Summary

This chapter detailed QOIA, our proposal for addressing the placement and management of IoT application components, considering their quality requirements and optimisation objectives. We started the chapter by providing an overview of the K8S configMaps, scheduler framework, and

[10] https://kops.sigs.k8s.io/

Diktyo, which are the technologies and tools we used to build QOIA. The configMaps are the K8S tool we used to build QOIA's quality configuration language, which defines the quality requirements and scheduling optimisation objectives of IoT application components. The scheduler framework is the K8S extension mechanisms we used to build our quality-aware scheduler. Diktyo is a K8S scheduler that handles QoS requirements, which we incorporated in our solution. Throughout this chapter, we detailed QOIA architecture and how it takes advantage of these technologies and tools to provide its configurable IoT Quality-aware Scheduler and Runtime Watcher modules, which enforce compliance with user-defined quality requirements and optimisation objectives during scheduling and runtime without being tied to a particular set of quality metrics and models.

After describing QOIA, we explained through three different use cases - Smart Building, Smart Industry and Smart Lighting - how to use QOIA's quality configuration to specify quality requirements and optimisation objectives. We highlighted in this explanation the types of operators currently supported by QOIA for quality requirements evaluation, and how users can build and integrate their own quality models in QOIA using the extension mechanism we provide through Go plugins.

Before closing the chapter, we evaluated QOIA's scheduling and recovery performance in the Smart Lighting use case, which we scaled in the number of applications and Nodes in the cluster. To assess the overhead added by QOIA's configurable quality-aware scheduling and runtime management, we compared its scheduling performance with Diktyo. The results showed that QOIA's scheduling and recovery mechanisms are applicable in IoT and IIoT scenarios with fair cluster loads, even time-demanding ones (sub-second unavailability). In demanding cluster loads, the performance might be acceptable for use cases without time-demanding requirements. We concluded from the result analysis that the performance bottleneck is in the K8S API server, so for demanding loads and time-demanding use cases, we should explore improving its response time by scaling the K8S control plane horizontally and fine-tuning the K8S API server resource limits.

Chapter 6

Conclusion and Future Directions

This chapter reviews the contributions developed throughout this Thesis, how they are integrated in DOT to validate it, and our immediate next steps (Section 6.1). Additionally, considering the advancements of this Thesis towards the vision of IoT continuum orchestration, we outline what we consider to be future research directions that have not yet been fully addressed by the community (Section 6.2).

6.1 Conclusions

In this document, we addressed the IoT continuum orchestration and stated the following Thesis:

"Improving existing Cloud orchestration tools, such as Kubernetes, to consider IoT devices heterogeneity and IoT applications quality requirements, allows such devices to be considered legitimate resources for application component placement and life cycle management, enabling IoT continuum orchestration."

We validated this Thesis by answering three research questions:

- RQ1: *Can standards-based approaches be used to address IoT device heterogeneity and avoid vendor lock-in?*
- RQ2: *Can Cloud orchestration systems handle the heterogeneity of IoT devices runtime environments?*
- RQ3: *Can Cloud orchestration systems be extended to consider the quality characteristics of IoT applications?*

We answered these research questions with three independent contributions: 1) NextGenGW (cf. N in Figure 6.1 and detailed in Chapter 3) provides an answer to RQ1; 2) FITA (cf. F1 to F3 in Figure 6.1 and detailed in Chapter 4) provides an answer to RQ2; and finally 3) QOIA (cf. Q1 to Q3 in Figure 6.1 and detailed in Chapter 5) provides an answer to RQ3. When integrated as

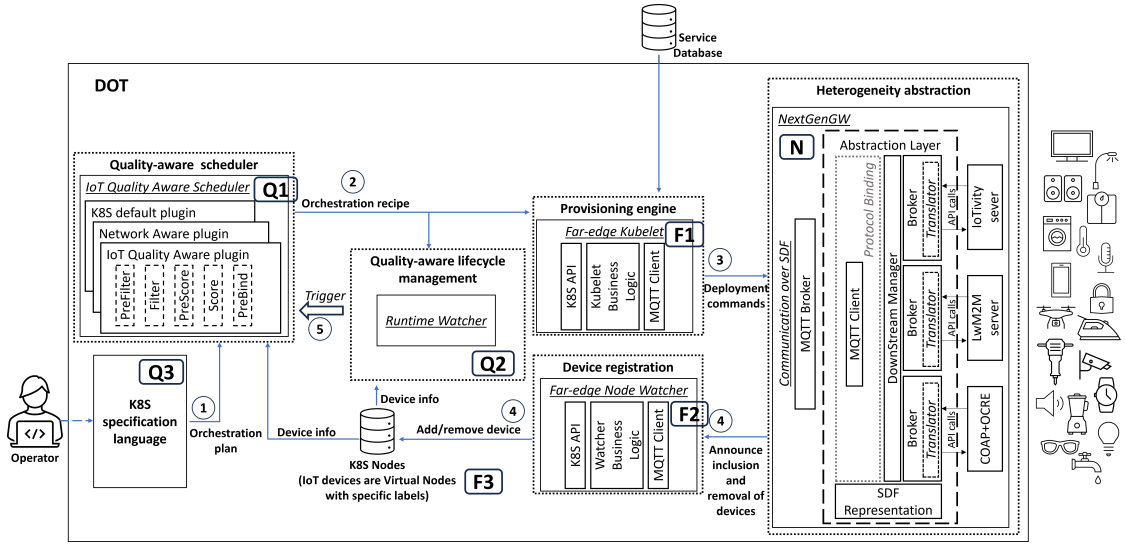


Figure 6.1: DOT architecture overview

shown in Figure 6.1, these contributions form DOT, which is our proposal for addressing the IoT continuum orchestration using K8S, thereby validating our Thesis.

These independent contributions advance the State-of-the-Art in their respective topic, namely:

- NextGenGW addresses IoT interoperability by abstracting IoT devices heterogeneous interaction and data models using IETF SDF bound with MQTT. It instantiates the endpoints that communicate with the IoT devices (IoTivity server, LwM2M server and COAP+OCRE, in the scope of this Thesis), which are paired with one translator each, responsible for converting the data models of IoT devices into IETF SDF and binding them to MQTT. This abstraction eases the integration of new devices in the IoT installation and avoids vendor lock-in through the use of standards. In the scope of IoT orchestration, NextGenGW provides a unified data representation and communication interface to the orchestration framework.

- FITA addresses the IoT devices orchestration challenge by integrating IoT devices in K8S. It provides the device registration (cf. F2), device representation (cf. F3) and provisioning mechanisms (cf. F1) through the Far-edge Node Watcher, the K8S Virtual Nodes and the Far-edge Kubelet, respectively. The Far-edge Node Watcher subscribes to the NextGenGW announce and unregister topics, creating or deleting the Far-edge Kubelets according to the messages received on these topics. The Far-edge Kubelet is the proxy between the K8S API and the NextGenGW. It is paired with one K8S Virtual Node, and this combination is the instantiation of the IoT device in the K8S cluster. We have one pair of Far-edge Kubelet and K8S Virtual Node per IoT device. The K8S Virtual Node represents the IoT device in the cluster through a set of labels that hold the specific characteristics of IoT devices. The Far-edge Kubelet receives the application component deployment commands from the K8S API and deploys the application component on its IoT device independently of the device

runtime environment, through the NextGenGW. It also monitors the IoT device CPU and memory usage, updating the K8S Virtual Node accordingly.

- QOIA addresses the quality aspects of IoT orchestration by providing a configurable scheduler (cf. Q1) and life cycle management (cf. Q2) module. QOIA's Quality-aware Scheduler determines the placement of application components, considering their quality requirements and optimisation objectives without being tied to a fixed set of quality characteristics and metrics. Users configure the quality metrics, scheduling requirements, and optimisation objectives of each application component in a configuration file (cf. Q3) that is loaded by the scheduler and Runtime Watcher, thereby configuring them. The Runtime Watcher uses that information to monitor the status of the K8S Nodes and the scheduled application component, ensuring they still meet their quality requirements. If non-compliances are found, it evicts them and requests a reschedule.

DOT integrates these individual contributions enabling IoT continuum orchestration. The orchestration flow begins with the operator specifying their orchestration plan (cf. 1) using the K8S specification language and the QOIA proposed extensions. The orchestration plan feeds the QOIA Quality-aware Scheduler, which inspects the K8S Nodes available in the cluster to produce the orchestration recipe that identifies the K8S Node that meets the orchestration plan specification (cf. 2). This plan is also shared with QOIA's Quality-aware life cycle manager (Runtime Watcher) that monitors the scheduled application component and the Node to which it was bound. The deployment continues with the transmission of the orchestration recipe to the Kubelet associated with the K8S Node identified by the scheduler. If such K8S Node is a K8S Virtual Node managed by the Far-edge Kubelet, then it is this entity that places the application component in the Node. Otherwise, it follows the default K8S deployment procedure using the default K8S Kubelet. When the Far-edge Kubelet is used, it sends the deployment commands to the NextGenGW (cf. 3) that translates them and forwards them to the corresponding IoT device, completing the deployment process. At runtime, IoT devices can be added or removed from the IoT installation. NextGenGW shares this information on its MQTT topics, which are subscribed to by the Far-edge Node Watcher that adds or removes them from the K8S cluster (cf. 4). Additionally, the Runtime Watcher detects non-compliances of the monitored application components with the operator-defined quality requirements. If non-compliances are found, the Runtime Watcher de-schedules the non-compliant application component and triggers a new scheduling cycle (cf. 5).

The evaluation of the proposed solutions demonstrates their effectiveness with an acceptable overhead, making them suitable for use cases involving a high number of heterogeneous devices, without stringent timing requirements. Through QOIA, DOT is capable of handling quality non-compliances at runtime, though the associated timing overhead must be taken into account in harsh environments with a high probability of many simultaneous faults. Considering the measured results, DOT is well-suited for many Smart Cities, Smart Buildings, and Industrial IoT use cases.

Our next steps involve evaluating DOT in a Smart City installation to assess its performance and fine-tune it in a real-world deployment. We also want to explore timing overhead mitigation

strategies, such as the usage of redundant replicas for critical application components that require swift recovery when non-compliances are detected, as well as the improvement of K8S API server concurrent requests handling, which is one of the main contributors to DOT's total deployment (including scheduling) and recovery latency.

Focusing on the heterogeneity abstraction brought by NextGenGW, we are updating the "SDF Representation" module and its binding with MQTT to consider the latest version of the standard [57], which was recently submitted to the Internet Engineering Steering Group for publication. We are also exploring the binding with CoAP. In this regard, we are looking at the IETF efforts around CoMI, whose status was recently submitted to the Internet Engineering Steering Group as a proposed standard, revealing its current maturity. Bounding SDF, with CoMI might further enhance the heterogeneity abstraction standardisation effort. In this regard, there are also three more incipient drafts, currently not endorsed by IETF, that are worth keeping track of and, if appropriate, contribute to:

- The "Protocol Mapping for SDF" [73] is exploring how to formally represent the bound between SDF and a communication protocol. This can improve how we document our proposed bounds between SDF and MQTT, and we could contribute with the knowledge we gained from our proposed bounding.

- The "Semantic Definition Format (SDF): Mapping files" [13] explores how to formally represent the mapping between SDF and other data formats, following the intents of OneDM liaison organisation, which was the SDF proponent. This work could improve how we formally represent the mapping performed by our proposed translators, i.e., SDF to LwM2M, IoTivity and our defined CoAP protocol. We could also contribute to this effort, considering our experience with the creation of the proposed translators.

- The "Instance Information for SDF" [12] is working on how to model instances of devices and their objects using SDF, and how to map such instances with the SDF model that defines them. In the current draft status, it is not yet clear how instances are formally represented, nor how, or if, such formal representation is related to the SDF protocol bindings. Nevertheless, we should keep track of the work on this draft to clarify these points and understand if it relates to the work done on NextGenGW.

6.2 Future Directions

While developing DOT, we came across two main research lines that are worth exploring further in future work. One relates to the continuous effort to align the IoT orchestration framework with existing standards to avoid vendor lock-in (Section 6.2.1), and the other one relates to the synchronisation of IoT devices with their digital counterparts (Section 6.2.2). Figure 6.2 presents in grey the envisioned impact of these future research lines on the current version of DOT.

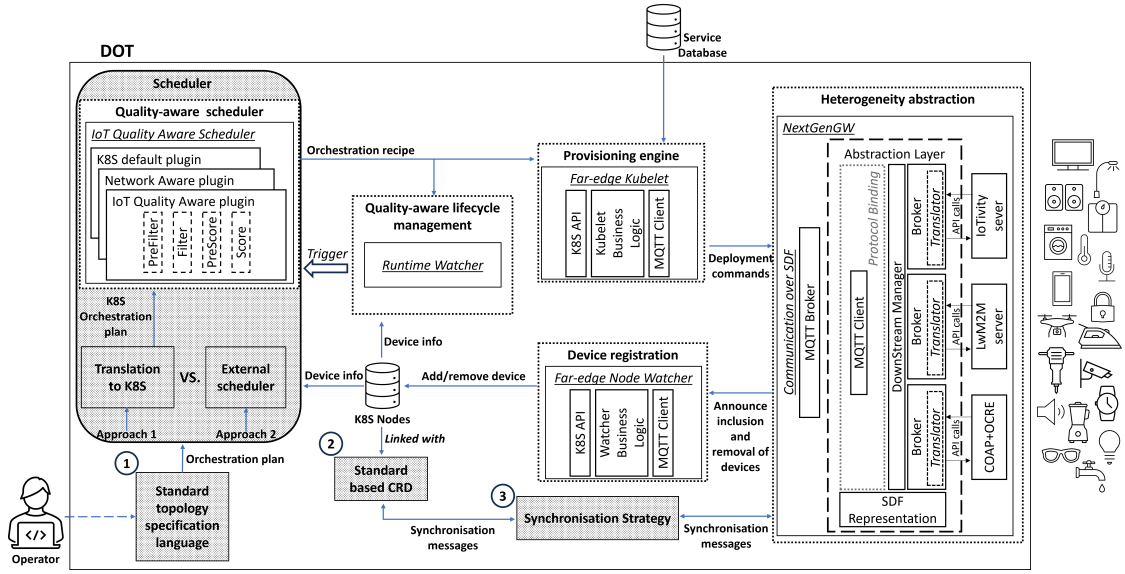


Figure 6.2: DOT future directions

6.2.1 Continuous Alignment with Standards

DOT was designed with standard alignment in mind to address vendor lock-in. This is a continuous effort, mainly due to the current ongoing work on the different standard organisations. With DOT, we addressed the interoperability issue on the interaction with the IoT devices through the heterogeneity abstraction. However, there are two other interaction points that were outside the scope of DOT and should be taken into consideration: the (1) topology specification language used by the operator to define its deployment, and (2) the digital representation of IoT devices and their resources in the cluster.

Regarding the **specification language** (cf. 1), DOT currently uses K8S specification language, which we extended for the definition of IoT applications component quality requirements and scheduling objectives. An interesting improvement would be the integration of a standard topology specification language as the system interface with the operator. The standard topology specification language would allow the operators to define their orchestration plan, while in the background, two approaches could be followed:

1. The standards-based orchestration plan is translated to the K8S specification language along with our proposed extensions to support quality requirements and optimisation objectives. The translation result would then feed QOIA, following the procedure currently being used.

2. Use an external scheduler that understands the standard topology specification language and outputs the orchestration recipe in the format understood by the K8S control plane. This recipe would also need to include the extension we proposed to identify quality requirements and optimisation objectives, which feed the quality-aware life cycle management module.

As explored in section 2.1.2, OASIS TOSCA is a relevant standard topology specification language. However, currently it does not support IoT devices and their application requirements. Its

current status and industry adoption are also unclear. Nevertheless, keeping monitoring TOSCA Ad Hoc workgroup on IoT, Edge and Fog[1], as well as the evolution of the latest standard specification [59] might be worth it.

In what concerns the **digital representation of IoT in the cluster** (cf. 2), DOT uses K8S nodes, which comprise specific labels to represent the specific characteristics of IoT devices. While this is an interesting approach that allows QOIA to reason about specific IoT characteristics and quality requirements, following a standards-based approach would further improve its generalisation and vendor lock-in aspects. Considering this, using IETF SDF for the representation of IoT devices in the K8S cluster would be worth exploring. The IoT devices would continue to have their virtual representation in the cluster in a K8S Node, but this Node would be associated with a K8S Custom Resource Definition, which would follow the SDF definition to enhance its representation. To achieve this, one would need to explore the compatibility between the SDF syntax and K8S Custom Resource Definition, and propose a mapping between the two. The recent IETF informational draft "Semantic Definition Format (SDF) modelling for Digital Twin" [61] could be explored for this purpose.

This approach for IoT devices representation following IETF SDF feeds a data model-based scheduler that reasons about the specific characteristics and capabilities of each device, independently of the application domain. However, for domain-specific schedulers that reason about domain knowledge, processes and context, the information provided by IETF SDF might not be enough. For such an application, one could explore the existing standard ontologies, ideally as a complement to the information provided by IETF SDF. For example, considering the information provided in Table 2.2:

- SSN and SOSA could be used to feed an ontology-aware scheduler that reasons over sensing and actuation procedures, as well as the device deployment context. For example, the scheduling objective "deploy application component A on a device with a wind sensor whose feature of interest is ambient wind, deployed in street A, an observation procedure with height equal to 10 meters and a maintenance schedule higher than 30 days" would probably be more efficient if built using the SSN and SOSA ontology;

- SAREF could be used to further fine-tune the orchestration to the specific application domains it supports. For example, the objective "deploy application component on all waste treatment devices of type compost bin, which contain a temperature sensor whose feature of interest is compost" fits SAREF environmental domain.

In this scope, the mappings between SSN and SOSA with other ontologies listed in Section 2.2.2 should be explored to ease the adaptation of the representation between multiple domains.

[1] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca

6.2.2 Synchronisation of IoT Devices and their Virtual Counterparts

In the scope of this Thesis, DOT addresses the orchestration and runtime management of IoT devices using the solutions proposed in Sections 4 and 5, respectively. These solutions use the status and information of the IoT device represented digitally inside the K8S cluster as a Virtual Node. It was left outside the scope of this Thesis to address the synchronisation of the status between the physical IoT device, whose behaviour might change over time due to wear, ageing and faults [116], and the Virtual Node that represents it (cf. 3, Figure 6.2). This section presents some possible research directions regarding this challenge.

From our perspective, the synchronisation between the physical IoT devices and their virtual counterparts in the scope of orchestration is related to the digital twin synchronisation challenge being actively discussed in the literature. As debated in section 6.2.1, we think that the representation of IoT devices in the K8S cluster should evolve into a standards-based representation. Ultimately, this would be a digital twin of the IoT device in the scope of orchestration[2].

The digital twin synchronisation main challenge stems from the high number of devices being modelled and the amount of data produced in this process [63]. This creates a tradeoff between an optimal use of resources to keep the digital twins updated versus how accurately the digital twin captures the several aspects of the device [51, 16]. Cakir et al. [16] performed an interesting simulation on how these two aspects affect each other, considering different network and synchronisation configurations.

This network aspect of the synchronisation problem is also studied by Khan et al. [53], who focus on using digital twins to build self-configuring wireless systems. They propose a taxonomy covering twins for wireless and wireless for twins. The former relates to the use of digital twins to model wireless systems, and the latter to the efficient use of the wireless network resources to support the signalling and communication with the digital twins. The proposed tutorial is a good source of information regarding key strategies and technologies for digital twin synchronisation in the scope of wireless systems. Huang et al. [63] explored the digital twin synchronisation problem through the Mobile Edge Computing perspective, proposing a multi-objective optimisation framework that considers resource usage and synchronisation frequency. The aim is to offload resource demands to the Edge to reduce latency and network usage, while considering the synchronisation frequency, which, according to the authors, is dynamic and depends on the device operational state.

The use of different synchronisation frequencies relates to another important aspect of the synchronisation of digital twins: measuring the performance of digital twins. In this scope, Loubany et al. [67] review the current synchronisation metrics for IoT networks. More recently, Zhang et al. [115] propose a generalised model to measure the synchronisation performance considering that the IoT devices use different synchronisation strategies. They also built a scheduler that

[2] As discussed by Sjarov et al. [104], there is no unified concept of digital twin in the literature. Considering their work, the virtual representation of the IoT device in the scope of orchestration could be a digital shadow and not a digital twin. It depends on the existence or non-existence of a functional layer on top of the virtual representation of the device. Nevertheless, the synchronisation challenge discussed in this section is valid for both concepts.

optimises resource usage and synchronisation performance based on the proposed metric.

With the ability to measure the synchronisation performance, one can consider both the different IoT devices synchronisation strategies and the synchronisation performance required by each application. One can use this information to consciously perform work on top of the digital twin, being aware of how approximate its representation of the physical device is. In this scope, Jia et al. [48] propose an application-agnostic synchronisation scheme that handles synchronisation considering the requirements of each application. Kalasapura et al. [51] follow a different approach and propose a synchronisation protocol that considers a user-defined application-specific notion of error for the purpose of approximate synchronisation.

Although not focused on IoT orchestration, we think the approaches, solutions and issues identified in the digital twin literature referenced above are valid within the orchestration scope. This brief summary highlights the importance of the synchronisation problem and its complexity. It comprises different sub-problems, such as the allocation of network resources for digital twin synchronisation, the measurement of the synchronisation performance and the synchronisation protocol that optimises the tradeoff between resource usage and performance. In our opinion, the community should consider these research paths within the scope of IoT orchestration to resolve the synchronisation of the IoT devices with their digital counterparts. This would allow schedulers and runtime management modules to make informed decisions considering not only the information exposed through the digital twin but also its accuracy.

References

[1] Muhammad Aaqib, Aftab Ali, Liming Chen, and Omar Nibouche. IoT trust and reputation: a survey and taxonomy. *Journal of Cloud Computing*, 12(1):42, 2023.

[2] Muhammad Alam, Joao Rufino, Joaquim Ferreira, Syed Hassan Ahmed, Nadir Shah, and Yuanfang Chen. Orchestration of microservices for iot using docker and edge computing. *IEEE Communications Magazine*, 56(9):118–123, 2018.

[3] Gianluca Aloi, Giuseppe Caliciuri, Giancarlo Fortino, Raffaele Gravina, Pasquale Pace, Wilma Russo, and Claudio Savaglio. Enabling IoT interoperability through opportunistic smartphone-based mobile gateways. *Journal of Network and Computer Applications*, 81:74–84, 2017.

[4] Christian Amsüss, Zach Shelby, Michael Koster, Carsten Bormann, and Peter Van der Stok. CoRE Resource Directory. Internet-Draft draft-ietf-core-resource-directory-28, Internet Engineering Task Force, March 2021. Work in Progress.

[5] Rob Atkinson, Raúl García-Castro, Joshua Lieberman, Claus Stadler, Armin Haller, Krzysztof Janowicz, Simon Cox, Danh Le Phuoc, Kerry Taylor, and Maxime Lefrançois. Semantic Sensor Network Ontology. W3C Recommendation REC-vocab-ssn-20171019, W3C, October 2017.

[6] Mahmoud M Badawy, Zainab H Ali, and Hesham A Ali. QoS provisioning framework for service-oriented internet of things (IoT). *Cluster Computing*, 23(2):575–591, 2020.

[7] Sharu Bansal and Dilip Kumar. IoT ecosystem: A survey on devices, gateways, operating systems, middleware and communication. *International Journal of Wireless Information Networks*, 27(3):340–364, 2020.

[8] Gunjan Beniwal and Anita Singhrova. A systematic literature review on IoT gateways. *Journal of King Saud University-Computer and Information Sciences*, 2021.

[9] Maria Bermudez-Edo, Tarek Elsaleh, Payam Barnaghi, and Kerry Taylor. IoT-Lite Ontology. W3C Member Submission SUBM-iot-lite-20151126, W3C, November 2015.

[10] Andy Bierman, Martin Björklund, and Kent Watsen. RESTCONF Protocol. RFC 8040, January 2017.

[11] Stefano Borgo, Roberta Ferrario, Aldo Gangemi, Nicola Guarino, Claudio Masolo, Daniele Porello, Emilio M. Sanfilippo, Laure Vieu, Stefano Borgo, Antony Galton, and Oliver Kutz. DOLCE: A descriptive ontology for linguistic and cognitive engineering1. *Appl. Ontol.*, 17(1):45–69, January 2022.

[12] Carsten Bormann and Jan Romann. Instance Information for SDF. Internet-Draft draft-bormann-asdf-instance-information-05, Internet Engineering Task Force, July 2025. Work in Progress.

[13] Carsten Bormann and Jan Romann. Semantic Definition Format (SDF): Mapping files. Internet-Draft draft-bormann-asdf-sdf-mapping-07, Internet Engineering Task Force, July 2025. Work in Progress.

[14] Mike Botts, Alexandre Robin, and Eric Hirschorn. OGC SensorML: Model and XML Encoding Standard. OGC Standard 12-000r2, OGC, August 2020.

[15] Antonio Brogi, Stefano Forti, and Ahmad Ibrahim. Optimising QoS-Assurance, Resource Usage and Cost of Fog Application Deployments. In *Cloud Computing and Services Science*, pages 168–189. Springer International Publishing, 2019.

[16] Lal Verda Cakir, Sarah Al-Shareeda, Sema F Oktug, Mehmet Özdem, Matthew Broadbent, and Berk Canberk. How to synchronize digital twins? a communication performance analysis. In *2023 IEEE 28th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 123–127. IEEE, 2023.

[17] Carmen Carrión. Kubernetes scheduling: Taxonomy, ongoing issues and challenges. *ACM Computing Surveys*, 55(7):1–37, 2022.

[18] Graham Colclough, Leen Peeters, Christina Protopapadaki, and Judith Borsboom. Smart Lighting in Cities: Factsheet. Technical report, Smart Cities Marketplace, June 2021. Accessed: 2025-04-12.

[19] OpenFog Consortium. OpenFog Reference Architecture for Fog Computing. Technical Report OPFRA001.020817, Industry IoT Consortium, February 2017.

[20] Mauro AA da Cruz, Joel José PC Rodrigues, Jalal Al-Muhtadi, Valery V Korotaev, and Victor Hugo C de Albuquerque. A reference model for internet of things middleware. *IEEE Internet of Things Journal*, 5(2):871–883, 2018.

[21] Mauro AA da Cruz, Joel JPC Rodrigues, Pascal Lorenz, Valery V Korotaev, and Victor Hugo C de Albuquerque. In. IoT—a new middleware for internet of things. *IEEE Internet of Things Journal*, 8(10):7902–7911, 2020.

[22] Rustem Dautov and Hui Song. Towards Agile Management of Containerised Software at the Edge. In *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*, volume 1, pages 263–268. IEEE, 2020.

[23] Mathias Santos de Brito, Saiful Hoque, Thomas Magedanz, Ronald Steinke, Alexander Willner, Daniel Nehls, Oliver Keils, and Florian Schreiner. A service orchestration architecture for fog-enabled infrastructures. In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 127–132. IEEE, 2017.

[24] Pratikkumar Desai, Amit Sheth, and Pramod Anantharam. Semantic gateway as a service architecture for iot interoperability. In *2015 IEEE International Conference on Mobile Services*, pages 313–319. IEEE, 2015.

[25] Bruno Donassolo, Ilhem Fajjari, Arnaud Legrand, and Panayotis Mertikopoulos. Fog based framework for IoT service provisioning. In *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6. IEEE, 2019.

[26] Rob Enns, Martin Björklund, Andy Bierman, and Jürgen Schönwälder. Network Configuration Protocol (NETCONF). RFC 6241, June 2011.

[27] Mehmet Ersue, Dan Romascanu, Jürgen Schönwälder, and Ulrich Herberg. Management of Networks with Constrained Devices: Problem Statement and Requirements. RFC 7547, May 2015.

[28] ETSI. Context Information Management (CIM); NGSI-LD API. Group Specification RGS/CIM-0009v142, ETSI, April 2021.

[29] ETSI. SmartM2M; SAREF: SDT interoperability and oneM2M base ontology alignment. Technical Report DTR/SmartM2M-103783, ETSI, May 2022.

[30] Mark Fedor, Martin Lee Schoffstall, James R. Davin, and Dr. Jeff D. Case. Simple Network Management Protocol (SNMP). RFC 1157, May 1990.

[31] Nicolas Ferry, Phu Nguyen, Hui Song, Pierre-Emmanuel Novac, Stéphane Lavirotte, Jean-Yves Tigli, and Arnor Solberg. Genesis: Continuous orchestration and deployment of smart iot systems. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 870–875. IEEE, 2019.

[32] Kaneez Fizza, Abhik Banerjee, Prem Prakash Jayaraman, Nitin Auluck, Rajiv Ranjan, Karan Mitra, and Dimitrios Georgakopoulos. A survey on evaluating the quality of autonomic internet of things applications. *IEEE Communications Surveys & Tutorials*, 25(1):567–590, 2022.

[33] Kaneez Fizza, Prem Prakash Jayaraman, Abhik Banerjee, Nitin Auluck, and Rajiv Ranjan. IoT-QWatch: A novel framework to support the development of quality-aware autonomic IoT applications. *IEEE Internet of Things Journal*, 10(20):17666–17679, 2023.

[34] Open Connectivity Foundation. OCF Device Specification 2.2.5. Technical report, Open Connectivity Foundation, January 1022.

[35] Ana Cristina Franco da Silva and Pascal Hirmer. Models for Internet of Things Environments—A Survey. *Information*, 11(10):487, 2020.

[36] Maria Ganzha, Marcin Paprzycki, Wiesław Pawłowski, Paweł Szmeja, and Katarzyna Wasielewska. Towards common vocabulary for IoT ecosystems—preliminary considerations. In *Asian Conference on Intelligent Information and Database Systems*, pages 35–45. Springer, 2017.

[37] Markus Graube, Stephan Hensel, Chris Iatrou, and Leon Urbas. Information models in OPC UA and their advantages and disadvantages. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2017.

[38] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. W3C Recommendation REC-sparql11-query-20130321, W3C, March 2013.

[39] Klaus Hartke. Observing Resources in the Constrained Application Protocol (CoAP). RFC 7641, September 2015.

[40] Klaus Hartke. The Constrained RESTful Application Language (CoRAL). Internet-Draft draft-ietf-core-coral-03, Internet Engineering Task Force, March 2020. Work in Progress.

[41] Hamdan Hejazi, Husam Rajab, Tibor Cinkler, and László Lengyel. Survey of platforms for massive IoT. In *2018 IEEE International Conference on Future IoT Technologies (Future IoT)*, pages 1–8, 2018.

[42] Saiful Hoque, Mathias Santos De Brito, Alexander Willner, Oliver Keil, and Thomas Magedanz. Towards container orchestration in fog computing infrastructures. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 294–299. IEEE, 2017.

[43] ISO/IEC. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. Published ISO/IEC 25010:2023, International Organization for Standardization, November 2023.

[44] ITU-T. Series Y: Global information infrastructure, Internet protocol aspects, next-generation networks, Internet of Things and smart cities. ITU-T Recommendation Y.4455, ITU, October 2017.

[45] Michael Jacoby and Thomas Usländer. Digital twin and internet of things—Current standards landscape. *Applied Sciences*, 10(18):6519, 2020.

[46] Krzysztof Janowicz, Alex Robin, and Hylke van der Schaaf. Semantic Sensor Network Ontology - 2023 Edition. W3C Editor's Draft OGC 25-022, W3C, December 2025.

[47] Cullen Jennings, Zach Shelby, Jari Arkko, Ari Keränen, and Carsten Bormann. Sensor Measurement Lists (SenML). RFC 8428, August 2018.

[48] Pengyi Jia, Xianbin Wang, and Xuemin Shen. Accurate and Efficient Digital Twin Construction Using Concurrent End-to-End Synchronization and Multi-Attribute Data Resampling. *IEEE Internet of Things Journal*, 10(6):4857–4870, 2023.

[49] Yuxuan Jiang, Zhe Huang, and Danny HK Tsang. Challenges and solutions in fog computing orchestration. *IEEE Network*, 32(3):122–129, 2017.

[50] Sebastian Kaebisch, Michael McCool, and Ege Korkan. Web of Things (WoT) Thing Description 1.1. W3C Recommendation REC-wot-thing-description11-20231205, World Wide Web Consortium, December 2023.

[51] Deepti Kalasapura, Jinyang Li, Shengzhong Liu, Yizhuo Chen, Ruijie Wang, Tarek Abdelzaher, Matthew Caesar, Joydeep Bhattacharyya, Jae Kim, Guijun Wang, Greg Kimberly, Josh Eckhardt, and Denis Osipychev. TwinSync: A Digital Twin Synchronization Protocol for Bandwidth-Limited IoT Applications. In *2023 32nd International Conference on Computer Communications and Networks (ICCCN)*, pages 1–1, 2023.

[52] Carlos Kamienski, Ivan Zyrianoff, Luiz Fernando Bittencourt, and Marco Di Felice. IoTinuum: The IoT Computing Continuum. In *2024 20th International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, pages 732–739, 2024.

[53] Latif U. Khan, Zhu Han, Walid Saad, Ekram Hossain, Mohsen Guizani, and Choong Seon Hong. Digital Twin of Wireless Systems: Overview, Taxonomy, Challenges, and Opportunities. *IEEE Communications Surveys & Tutorials*, 24(4):2230–2254, 2022.

[54] Jay Kiruthika and Souheil Khaddaj. Software quality issues and challenges of internet of things. In *2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*, pages 176–179. IEEE, 2015.

[55] Takashi Kono, Yasuhiko Taito, and Hideto Hidaka. Essential roles, challenges and development of embedded MCU micro-systems to innovate edge computing for the IoT/AI age. *IEICE Transactions on Electronics*, 103(4):132–143, 2020.

[56] Michael Koster and Carsten Bormann. Semantic Definition Format (SDF) for Data and Interactions of Things. Internet-Draft draft-ietf-asdf-sdf-11, Internet Engineering Task Force, February 2022. Work in Progress.

[57] Michael Koster, Carsten Bormann, and Ari Keränen. Semantic Definition Format (SDF) for Data and Interactions of Things. Internet-Draft draft-ietf-asdf-sdf-25, Internet Engineering Task Force, October 2025. Work in Progress.

[58] Viacheslav Kulik and Ruslan Kirichek. The heterogeneous gateways in the industrial internet of things. In *2018 10th International congress on ultra modern telecommunications and control systems and workshops (ICUMT)*, pages 1–5. IEEE, 2018.

[59] Chris Lauwers and Calin Curescu. TOSCA Version 2.0. OASIS Standard, OASIS OPEN, July 2025.

[60] Chris Lauwers and Paul Lipton. TOSCA Simple Profile in YAML Version 1.3. OASIS Standard, OASIS OPEN, February 2020.

[61] Hyunjeong Lee and Jungha Hong. Semantic Definition Format (SDF) modeling for Digital Twin. Internet-Draft draft-ietf-asdf-digital-twin-00, Internet Engineering Task Force, August 2025. Work in Progress.

[62] 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.

[63] Yaohua Li, Linyu Huang, Quan Yu, and Qian Ning. Optimization of Synchronization Frequencies and Offloading Strategies in MEC-Assisted Digital Twin Networks. *IEEE Internet of Things Journal*, 12(15):29203–29216, 2025.

[64] Yonghua Li, Xuxin Huang, and Siye Wang. Multiple protocols interworking with open connectivity foundation in fog networks. *IEEE Access*, 7:60764–60773, 2019.

[65] Steve Liang and Tania Khalafbeigi. OGC SensorThings API Part 2 – Tasking Core. Implementation 17-079r1, Open Geospatial Consortium, January 2019.

[66] Steve Liang, Tania Khalafbeigi, and Hylke van der Schaaf. OGC SensorThings API Part 1: Sensing Version 1.1. Standard 18-088, Open Geospatial Consortium, August 2021.

[67] Ali Loubany, May Itani, and Sanaa Sharafeddine. From Age of Information to Age of Digital Twin: A Review on Synchronization Metrics for IoT Networks. *IEEE Access*, 13:131102–131119, 2025.

[68] Open Mobile Alliance Ltd. OMA Device Management Protocol. Approved Version OMA-TS-DM_Protocol-V2_0-20160209-A, Open Mobile Alliance Ltd., February 2016.

[69] Open Mobile Alliance Ltd. Lightweight Machine to Machine Technical Specification: Core. Approved Version OMA-TS-LightweightM2M_Core-V1_2-20201110-A, Open Mobile Alliance Ltd., November 2020.

[70] Alex Mavromatis, Aloizio Pereira Da Silva, Koteswararao Kondepu, Dimitrios Gkounis, Reza Nejabati, and Dimitra Simeonidou. A software defined device provisioning framework facilitating scalability in Internet of Things. In *2018 IEEE 5G World Forum (5GWF)*, pages 446–451. IEEE, 2018.

[71] Yasser Mesmoudi, Mohammed Lamnaour, Yasser El Khamlichi, Abderrahim Tahiri, Abdellah Touhafi, and An Braeken. A middleware based on service oriented architecture for heterogeneity issues within the internet of things (MSOAH-IoT). *Journal of King Saud University-Computer and Information Sciences*, 32(10):1108–1116, 2020.

[72] Jozef Mocnej, Adrian Pekar, Winston K.G. Seah, Peter Papcun, Erik Kajati, Dominika Cupkova, Jiri Koziorek, and Iveta Zolotova. Quality-enabled decentralized IoT architecture with efficient resources utilization. *Robotics and Computer-Integrated Manufacturing*, 67:102001, 2021.

[73] Rohit Mohan, Bart Brinckman, and Lorenzo Corneo. Protocol Mapping for SDF. Internet-Draft draft-mohan-asdf-sdf-protocol-mapping-00, Internet Engineering Task Force, August 2025. Work in Progress.

[74] Roberto Morabito and Jaime Jiménez. IETF protocol suite for the Internet of Things: Overview and Recent Advancements. *IEEE Communications Standards Magazine*, 4(2):41–49, 2020.

[75] Donatien Koulla Moulla Moulla, Ernest Mnkandla, and Alain Abran. Systematic literature review of IoT metrics. *Applied Computer Science*, 19(1):64–81, 2023.

[76] Phu Nguyen, Nicolas Ferry, Gencer Erdogan, Hui Song, Stéphane Lavirotte, Jean-Yves Tigli, and Arnor Solberg. Advances in deployment and orchestration approaches for IoT-a systematic review. In *2019 IEEE International Congress on Internet of Things (ICIOT)*, pages 53–60. IEEE, 2019.

[77] Aditya Nugur, Manisa Pipattanasomporn, Murat Kuzlu, and Saifur Rahman. Design and development of an IoT gateway for smart building applications. *IEEE Internet of Things Journal*, 2018.

[78] João Oliveira, Filipe Sousa, and Luís Almeida. embServe: Embedded Services for Constrained Devices. In *2023 IEEE 19th International Conference on Factory Communication Systems (WFCS)*, pages 1–8, 2023.

[79] João Oliveira, Filipe Sousa, and Luís Almeida. IoTNetEMU - A Framework to Emulate and Test IoT Applications. In *Proceedings of the 19th ACM International Symposium on QoS and Security for Wireless and Mobile Networks*, Q2SWinet '23, page 33–37, New York, NY, USA, 2023. Association for Computing Machinery.

[80] oneM2M Partners. Base Ontology. oneM2M Technical Specification TS-0012-V3.7.3, oneM2M, February 2019.

[81] oneM2M Partners. oneM2M Technical Specification: Functional Architecture. oneM2M Technical Specification TS-0001-V3.24.0, oneM2M, June 2021.

[82] Open Connectivity Foundation. OCF Core Specification 2.2.8. Technical report, Open Connectivity Foundation, June 2025.

[83] Open Mobile Alliance Ltd. Lightweight Machine to Machine Technical Specification - Software Management. Approved OMA-TS-LWM2M_SwMgmt-V1_0_2-20210119-A, Open Mobile Alliance Ltd., January 2021.

[84] Claus Pahl, Sven Helmer, Lorenzo Miori, Julian Sanin, and Brian Lee. A container-based edge cloud paas architecture based on raspberry pi clusters. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pages 117–124. IEEE, 2016.

[85] Keyur Patel, Chinmay Mistry, Rajesh Gupta, Sudeep Tanwar, and Neeraj Kumar. A systematic review on performance evaluation metric selection method for IoT-based applications. *Microprocessors and Microsystems*, 101:104894, 2023.

[86] Ayesha Qamar, Muhammad Asim, Zakaria Maamar, Saad Saeed, and Thar Baker. A Quality-of-Things model for assessing the Internet-of-Things' nonfunctional properties. *Transactions on Emerging Telecommunications Technologies*, 33(8):e3668, 2022.

[87] Zeineb Rejiba and Javad Chamanara. Custom scheduling in Kubernetes: A survey on common problems and solution approaches. *ACM Computing Surveys*, 55(7):1–37, 2022.

[88] Carlos Resende. DOT - Digital Orchestration of Things. In *2023 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 247–248, 2023.

[89] Carlos Resende, Waldir Moreira, and Luís Almeida. NextGenGW: a Software-based Architecture Targeting IoT Interoperability. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4, 2022.

[90] Carlos Resende, Waldir Moreira, and Luís Almeida. NextGenGW - a Software Framework Based on MQTT and Semantic Definition Format. In *2023 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 141–148, 2023.

[91] Carlos Resende, João Oliveira, Filipe Sousa, Waldir Junior, and Luís Almeida. Managing Far-Edge Devices using Kubernetes. In *2024 International Conference on Embedded Wireless Systems and Networks (EWSN)*, 2024.

[92] Carlos Resende, João Oliveira, Filipe Sousa, Waldir Moreira, and Luís Almeida Sousa. Improving Far-Edge Device Management in IoT Applications Using Kubernetes. *IEEE Open Journal of the Industrial Electronics Society*, 6:1027–1049, 2025.

[93] Jan Romann. Bridging the Gap to the Web of Things. On the Conversion between WoT Data Models and the Semantic Definition Format. In *First International Workshop on Semantic Web on Constrained Things*, pages 27–46, 2023.

[94] Andreas Rossberg. WebAssembly Core Specification. W3C Candidate Recommendation Draft CRD-wasm-core-2-20250616, W3C, June 2025.

[95] Felix Maximilian Roth, Christian Becker, German Vega, and Philippe Lalanda. XWARE—A customizable interoperability framework for pervasive computing systems. *Pervasive and Mobile Computing*, 47:13–30, 2018.

[96] Peter Ruckebusch, Eli De Poorter, Carolina Fortuna, and Ingrid Moerman. GITAR: Generic extension for Internet-of-Things ARchitectures enabling dynamic updates of network and application modules. *Ad Hoc Networks*, 36:127–151, 2016.

[97] José Santos, Chen Wang, Tim Wauters, and Filip De Turck. Diktyo: Network-Aware Scheduling in Container-Based Clouds. *IEEE Transactions on Network and Service Management*, 20(4):4461–4477, 2023.

[98] Khaldoun Senjab, Sohail Abbas, Naveed Ahmed, and Atta ur Rehman Khan. A survey of Kubernetes scheduling algorithms. *Journal of Cloud Computing*, 12(1):87, 2023.

[99] Aggeliki Sgora and Periklis Chatzimisios. Defining and Assessing Quality in IoT Environments: A Survey. *IoT*, 3(4):493–506, 2022.

[100] Zach Shelby, Klaus Hartke, and Carsten Bormann. The Constrained Application Protocol (CoAP). RFC 7252, June 2014.

[101] Jonathan de C. Silva, Joel J. P. C. Rodrigues, Jalal Al-Muhtadi, Ricardo A. L. Rabêlo, and Vasco Furtado. Management Platforms and Protocols for Internet of Things: A Survey. *Sensors*, 19(3), 2019.

[102] Soraya Sinche, Duarte Raposo, Ngombo Armando, André Rodrigues, Fernando Boavida, Vasco Pereira, and Jorge Sá Silva. A survey of IoT management protocols and frameworks. *IEEE Communications Surveys & Tutorials*, 22(2):1168–1190, 2019.

[103] Manisha Singh and Gaurav Baranwal. Quality of service (qos) in internet of things. In *2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU)*, pages 1–6. IEEE, 2018.

[104] Martin Sjarov, Tobias Lechler, Jonathan Fuchs, Matthias Brossog, Andreas Selmaier, Florian Faltus, Toni Donhauser, and Jörg Franke. The Digital Twin Concept in Industry – A Review and Systematization. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 1789–1796, 2020.

[105] Hui Yie Teh, Andreas W Kempa-Liehr, and Kevin I-Kai Wang. Sensor data quality: A systematic review. *Journal of Big Data*, 7(1):11, 2020.

[106] Andreas Tsagkaropoulos, Yiannis Verginadis, Maxime Compastié, Dimitris Apostolou, and Gregoris Mentzas. Extending TOSCA for Edge and Fog Deployment Support. *Electronics*, 10(6):737, 2021.

[107] Michel Veillette, Peter Van der Stok, Alexander Pelov, Andy Bierman, and Ivaylo Petrov. CoAP Management Interface (CORECONF). Internet-Draft draft-ietf-core-comi-11, Internet Engineering Task Force, January 2021. Work in Progress.

[108] Karima Velasquez, David Perez Abreu, Marcio RM Assis, Carlos Senna, Diego F Aranha, Luiz F Bittencourt, Nuno Laranjeiro, Marilia Curado, Marco Vieira, Edmundo Monteiro, et al. Fog orchestration for the Internet of Everything: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 9(1):1–23, 2018.

[109] U. Warrier, L. Besaw, L. LaBarre, and B. Handspicker. Common Management Information Services and Protocols for the Internet (CMOT and CMIP). RFC 1189, October 1990.

[110] Gary White, Vivek Nallur, and Siobhán Clarke. Quality of service approaches in IoT: A systematic mapping. *Journal of Systems and Software*, 132:186–203, 2017.

[111] Cheng Xie, Beibei Yu, Zuoying Zeng, Yun Yang, and Qing Liu. Multilayer internet-of-things middleware based on knowledge graph. *IEEE Internet of Things Journal*, 8(4):2635–2648, 2020.

[112] Diana C Yacchirema, Manuel Esteve, and Carlos E Palau. Design and implementation of a gateway for pervasive smart environments. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 004454–004459. IEEE, 2016.

[113] Sami Yangui, Pradeep Ravindran, Ons Bibani, Roch H Glitho, Nejib Ben Hadj-Alouane, Monique J Morrow, and Paul A Polakos. A platform as-a-service for hybrid cloud/fog environments. In *2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–7. IEEE, 2016.

[114] Ashkan Yousefpour, Ashish Patil, Genya Ishigaki, Inwoong Kim, Xi Wang, Hakki C Cankaya, Qiong Zhang, Weisheng Xie, and Jason P Jue. FOGPLAN: A lightweight QoS-aware dynamic fog service provisioning framework. *IEEE Internet of Things Journal*, 6(3):5080–5096, 2019.

[115] Qiuyang Zhang, Ying Wang, and Zhendong Li. Scheduling of Digital Twin Synchronization in Industrial Internet of Things: A Hybrid Inverse Reinforcement Learning Approach. *IEEE Internet of Things Journal*, 12(5):5137–5147, 2025.

[116] Holger Zipper and Christian Diedrich. Synchronization of Industrial Plant and Digital Twin. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1678–1681, 2019.