ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN
COMPUTER SCIENCE AND ENGINEERING
Ciclo XXXIII

Settore Concorsuale: 01/B1 - INFORMATICA
Settore Scientifico Disciplinare: INF/01 - INFORMATICA

Managing Device and Platform Heterogeneity through the Web of Things

Presentata da:
Luca Sciullo

Supervisore:
Prof. Marco di Felice

Coordinatore Dottorato:
Prof. Davide Sangiorgi

Co-supervisore:
Prof. Luciano Bononi

Esame finale anno 2021

Haba na haba hujaza kibaba

Abstract

The chaotic growth of the Internet of Things (IoT) determined a fragmented landscape with a huge number of devices, technologies, and platforms available on the market, and consequential issues of interoperability on many system deployments. The Web of Things (WoT) architecture recently proposed by the W3C consortium constitutes a novel solution to enable interoperability across IoT platforms and application domains. At the same time, in order to see an effective improvement, a wide adoption of the W3C WoT solutions from the academic and industrial communities is required; this translates into the need of accurate and complete support tools to ease the deployment of W3C WoT applications, as well as reference guidelines about how to enable the WoT on top of existing IoT scenarios and how to deploy WoT scenarios from scratch. In this thesis, we bring three main contributions for filling such gap: (1) we introduce the WoT Store, a novel platform for managing and easing the deployment of Things and applications on the W3C WoT, and additional strategies for bringing old legacy IoT systems into the WoT. The WoT Store allows the dynamic discovery of the resources available in the environment, i.e. the Things, and to interact with each of them through a dashboard by visualizing their properties, executing commands, or observing the notifications produced. In addition, similarly to popular app stores, the WoT Store allows the search and execution of third-party WoT applications that interact with the available Things again in a seamless way. (2) We map three different IoT scenarios to WoT scenarios: a generic heterogeneous environmental monitoring scenario, where the goal is to orchestrate the sensing of different Wireless Sensor Networks (WSNs), a structural health monitoring (SHM) scenario, where civil structures are monitored by purposely-designed sensors, and an Industry 4.0 scenario, where industrial devices are deployed and orchestrated for a production process pipeline. For each of them, we describe the challenges addressed for this kind of operation to validate such effort. (3) We make concrete proposals to improve both the W3C standard and the *node-wot* software stack design: in the first case, new vocabularies are needed in order to handle particular protocols employed in industrial scenarios, while in the second case we present some contributions required for handling the dynamic instantiation and the migration of Web Things and WoT services in a cloud-to-edge continuum environment.

Table of contents

List of figures **ix**

List of tables **xiii**

I Background **1**

1 Introduction **3**

2 State of the art **7**

2.1 Internet of Things 7

2.1.1 IoT application domains 8

2.2 IoT Interoperability 12

2.3 Web of Things 14

2.3.1 Requirements 16

2.4 Web of Things: definitions 19

2.4.1 Web Thing 19

2.4.2 Mashup application 22

2.5 Architecture 23

2.5.1 Access Layer 25

2.5.2 Find Layer 26

2.5.3 Share layer 28

2.5.4 Compose Layer 30

2.5.5 W3C WoT 32

II Contributions **37**

3 Tools **39**

3.1 WoT Store . . . 39
3.1.1 Overview . . . 39
3.1.2 Service Components . . . 43
3.1.3 Implementation . . . 47
3.1.4 Components Validation . . . 49
3.2 Web of Things as REST APIs and communication with legacy IoT Systems 61
3.2.1 Overview . . . 61
3.2.2 Architecture . . . 63
3.2.3 Implementation and Validation . . . 68

4 Use cases validation 71
4.1 Heterogeneous environmental monitoring . . . 74
4.1.1 Heterogeneous Sensing Scenario . . . 74
4.1.2 Testbed and Architecture . . . 75
4.1.3 Mashup sensing Policies . . . 78
4.1.4 Evaluation . . . 80
4.2 Structural Health Monitoring (SHM) . . . 83
4.2.1 SHM Scenario . . . 83
4.2.2 Architecture . . . 84
4.2.3 Implementation and Validation . . . 88
4.2.4 Deploy . . . 89

5 Improvements for WoT 95
5.1 Industrial WoT on the Edge . . . 96
5.1.1 Scenario . . . 96
5.1.2 Time-Sensitive Networking . . . 98
5.1.3 Design . . . 100
5.1.4 Protocol Bindings: OPC-UA . . . 103
5.1.5 Protocol Bindings: NETCONF . . . 106
5.1.6 Proof of Concept . . . 109
5.2 Web Things migration from Cloud To Edge . . . 112
5.2.1 Context and Motivations . . . 112
5.2.2 Architecture . . . 118
5.2.3 Migration Policy . . . 125
5.2.4 Proposed Heuristic . . . 127
5.2.5 Implementation . . . 131
5.2.6 Validation . . . 133

III Conclusions 143

6 Conclusions 145

6.1 Current and future research directions on the WoT 146

References 149

List of figures

2.1 Taxonomy of IoT application domains. 9
2.2 WoT evolution from [1]. 15
2.3 W3C Web Thing architecture proposed in [2]. 19
2.4 Web Thing architecture proposed in [3]. 25
2.5 W3C Web of Things Architecture [2] 32
2.6 Implementation of a Servient using the WoT Scripting API [2] 34

3.1 WoT Store functionalities and sequence of operations: Things discovery and deploy of a Mashup application. 41
3.2 WoT Store functionalities and sequence of operations: reconfiguration of the scenario through the instantiation of new Things and the update of the Thing Applications. 42
3.3 The operations of the Things Discovery Service (TDS). 43
3.4 A portion of the TD of the Device Thing of Table 3.3. The Thing is associated to a wireless sensor producing temperature values. 44
3.5 The rendering of the actions of the TD of Figure 3.4 within the WoT Store. 44
3.6 The WoT Store internals. 48
3.7 The IoT/WoT monitoring system deployed in this study. 50
3.8 The per-sensor RTT and PDR metrics are shown respectively in Figures 3.8(a) and 3.8(b). 52
3.9 The impact of the Thing Discovery Service on the PDR and RTT performance indexes in a scenario with a varying number of available Things/devices is shown in Figure 3.9(a). The Thing/device utilization over time is depicted in Figure 3.9(b). 53
3.10 The RTT and PDR values when switching the MA in use over time. 54
3.11 The crowdsensing system considered in this study is depicted in Figure 3.11(a). The abstraction of the WoT deployment with the WoT Store and the real/simulated entities is represented in Figure 3.11(b). 55

3.12 The average sensing value and the number of Things detecting the event for the *Random* MA are shown in Figure 3.12(a). The same metrics for the *Adaptive* MA are shown in Figure 3.12(b). 58
3.13 The geodata stream visualization for the *Random* MA before the occurrence of the event (Figure 3.13(a)) and during the occurrence (Figure 3.13(b)). . 59
3.14 The geodata stream visualization for the *Adaptive* MA before the occurrence of the event (Figure 3.14(a)) and during the occurrence (Figure 3.14(b)). . 59
3.15 The resource (CPU, RAM) utilization of the WoT Store for increasing number of deployed Things in the crowdsensing scenario. 60
3.16 The System Architecture 65
3.17 Sequence diagram presenting the interactions of all the components of the three-layer architecture 67
3.18 Figure 3.18(a) shows the Online Things vs Discovered Services, while Figure 3.18(b) shows the mean detected value over all the sensors. 69

4.1 The WoT Store Architecture mapped into a layered architecture 72
4.2 The IoT/WoT monitoring system deployed in this study. 75
4.3 The average per-device RTT and PDR is shown in Figures 4.3(a) and Figure 4.3(b), respectively. The per-device RTT for the CoAP protocol is shown in Figure 4.3(c). 81
4.4 The RTT and PDR values for the four mash-up policies are shown in Figures 4.4(a) and 4.4(c). The device utilization ratio for the P_1 policy is shown in Figure 4.4(b). 82
4.5 The device utilization ratio for the P_2 policy is shown in Figure 4.5(a). The RTT and PDR values when replacing the active policy at run-time are shown in Figure 4.5(b). The RTT when enabling/disabling the WoT approach is shown in Figure 4.5(c). 82
4.6 Proposed layered SHM architecture. 85
4.7 The MODRON software platform with the edge/cloud components. 86
4.8 A set of four subfigures. 90
4.9 The monitored metallic truss structure and relative sensor equipment: two logical SANs (red and green colors, respectively) are connected to as many GW interface devices (black boxes). 92
4.10 The MODRON Data Manager allowing the selection, visualization, and exportation of sensor time-series. 92
4.11 The MODRON Data Manager detail of sensor data. 93

5.1 Different TSN concepts at the egress port 99
5.2 Industrial IoT environment with TSN-enabled Ethernet connectivity and end-devices with OPC UA interfaces, orchestrated through a W3C Web of Things Servient 101
5.3 TSN Testbed with WoT-based supervisory logic on edge node and field devices with OPC UA interface 111
5.4 The M-WoT migration environment. 115
5.5 Two possible M-WoT use cases: the data processing service migration (Figure 5.5(a)) and the Digital Twin migration (Figure 5.5(b)). 116
5.6 The M-WoT software architecture. 119
5.7 The internal structure of the Orchestrator. 120
5.8 The M-WoT Servient internal structure. The new Monitoring API module is highlighted in solid green. 122
5.9 Sequence diagram of a WT migration event. 124
5.10 The NO, TF and CF metrics for the six policies when varying the number of active WTs are shown respectively in Figures 5.10(a), 5.10(b) and 5.10(c). 134
5.11 The average utilization of each computational node is shown in Figure 5.11(a). The IL metric when varying the number of active WTs is shown in Figure 5.11(b). The NO metric as a function of the WT degree is reported in Figure 5.11(c). 134
5.12 The CF and IL metrics when varying the WT degree are shown respectively in Figures 5.12(a) and 5.12(b). The NO over time slots in a dynamic WoT deployment where the number of WTs is varied over time is reported in Figure 5.12(c). 136
5.13 The TF over time slots in a dynamic WoT deployment where the number of WTs is varied over time is reported in Figure 5.13(a). The NO over time in the IoT monitoring use case is shown in Figure 5.13(b); the processing latency for the same scenario is reported in Figure 5.13(c). 137
5.14 CPU load (Figure 5.14(a)) and RAM consumption (Figure 5.14(b)) of the Orchestrator for different numbers of deployed WTs. 139

List of tables

2.1 Summary of the studies presented in Section 2.5 31

3.1 Example of RDF Description of a MA application available in the WoT Store. 47
3.2 List of technologies used for the implementation of the WoT Store service components of Section 3.1.2. 50
3.3 List of Properties, Actions, and Events of a Device Thing. 52
3.4 List of Properties, Actions, and Events of a Virtual Smartphone Thing in the crowdsensing scenario. 56
3.5 List of Properties, Actions, and Events of the WAE Thing. 68

4.1 Example of Properties, Actions, and Events described in a Thing Description of a Device Thing. 77
4.2 List of Affordances of a Controllable Sensor WT. 89

5.1 QoS requirement terms are assigned values by the mashup application and define the class `FloW`. They need to be parameterized through QoS capabilities (`cycleTime` between `minCycle` and `maxCycle` of the Thing, `maxBytes` given in Form). The combined parameters are passed as inputs to an `ScheduleFlow` Action of a scheduler Thing to request network configuration. 100
5.2 QoS capability terms, which are defined by the Thing. Their assignment is optional for TDs, but they are required if the Thing shall be used with QoS. The working clock must be identical to the one of the application (e.g., same PTP domain). 103
5.3 OPC UA Binding Template vocabulary 105
5.4 NETCONF Binding Template vocabulary 108
5.5 List of variables and parameters introduced in Section 5.2.3. 128

Part I

Background

Chapter 1

Introduction

Since the beginning, the Internet of Things (IoT) has been presented as a novel networking paradigm consisting of a huge base of connected devices that are able to produce and exchange data, and to enable new services thanks to seamless interaction among physical and virtual components [4][5]. At the same time, the presence on the market of heterogeneous software platforms, network protocols and Machine-to-Machine (M2M) technologies [6], as well as the creation of proprietary silos often lead by big vendors, have partially changed the vision of the IoT as a global interconnected and self-organizing system. Indeed, the lack of interoperability among heterogeneous platforms and devices has been indicated as one of the main issues of the IoT [7], since it might introduce additional costs for the system implementation and additional complexity for the re-use of existing solutions in different contexts. Furthermore, guaranteeing the interoperability among heterogeneous devices is becoming a key issue for current Industrial IoT (IIoT) systems, as well as an open field for novel scenarios and hence for novel business opportunities. For instance, a proper design and deployment of such systems play a fundamental role in the success of the transition to Industry 4.0, where efficiency, self-organization, and information-transparency strictly depend on achieving full interoperability. It is easy to believe that the presence of machines using different communication technologies, programming languages and data format can significantly increase the complexity and costs of existing IIoT deployment and integration [8] since, for instance, data collected in such scenarios can remain largely inaccessible in an integrated way unless significant manual effort is invested [9]. From another perspective, interoperability can be considered a remunerative research challenge [10]: a recent report from McKinsey quantifies in 40% the additional IoT value that can be unlocked when achieving full interoperability among heterogeneous IoT systems [11]. Interoperability can be interpreted in different ways, but according to the IEEE it is defined as "the ability of two more systems or components to exchange information and to use the information that has been exchanged" [12]. Research on interoperability

solutions for IoT systems has followed several approaches [13][4], some of them focused on reaching interoperability at the bottom layers, i.e. mainly at the networking layer - through the introduction of middleware or custom frameworks [14][15][16] -, and some of them at the upper layers, especially at the data and application ones. Nevertheless, this thesis mainly focuses on the latter approaches: on the one hand, semantic approaches have proposed to achieve interoperability among heterogeneous devices and platforms at the data layer [17][18][19], through the utilization of shared ontologies for IoT contexts like SOSA [20], data models like RDF, and languages like SPARQL. On the other hand, novel paradigms like the Web of Things (WoT) have investigated how to envision interoperability by re-using well-accepted standards and technologies adopted in the World Wide Web (WWW). Differently from other stack-oriented solutions (e.g. 6LoWPAN), WoT-based approaches propose to achieve system interoperability at the application layer, abstracting from the sensing and communication technologies: in a first approximation, Things are represented as Web resources, and all the interactions toward and between Things are mapped over Representational State Transfer (REST) services [3]. However, given the lack of a reference architecture, this approach has also led to a proliferation of architectures proposed in the literature (e.g. [21] [22][23]), mapped on different technologies (e.g. HTTP, MQTT, WebSockets) and with few elements in common besides the adoption of the RESTful interface [18], introducing further fragmentation and the consequential need of devising ad-hoc solutions for the system integration. Breaking the deadlock, the World Wide Web Consortium (W3C) has recently proposed some reference standards for the WoT [24] that formally describe the interfaces allowing IoT devices and services to communicate with each other, regardless of their underlying implementation. In the W3C WoT vision, everything can be considered a Thing and, to this purpose, each Thing is associated to a Thing Description (TD) providing general metadata as well as the interactions, data models, and security mechanisms of a Thing. Part of such metadata (named "Binding templates") describes the communication mode used to interact with the Thing, hence abstracting from the specific IoT protocol and data format. In addition, a TD can be serialized and semantically annotated via the JSON-LD language, hence representing a uniform model to enable Machine-to-Machine (M2M) communication toward a Thing and enabling several semantic features such as the Thing Discovery.

The W3C architecture greatly simplifies the deployment of software applications for the IoT, since they must not cope with details about the communication strategies implemented by the Things. Moreover, since each Thing exposes its capabilities through well-defined and uniform interfaces, the costs and effort for devising mashup applications gathering data

from large-scale and heterogeneous sensor installations (e.g. IoT monitoring systems) can be greatly reduced.

The success of the W3C WoT initiative strongly depends on its wide acceptance from the academic and industrial communities, as well as from the end-users. For this reason, it is extremely important to introduce proper instruments and solutions that leverage this new standard and to give feedback on its utilization in real scenarios. Hence, the main research questions of this thesis can be summarized into the following: (i) How to map heterogeneous IoT systems into the WoT? (ii) How to easily design and deploy WoT scenarios? (iii) Is the W3C standard effectively addressing every key point? How to improve and contribute to it? This leads us to the definition of our main macro-contributions:

1. regarding the question (i), we focus on different IoT scenarios where heterogeneous devices contribute to collect data that has to be processed to extract some kind of useful information or where heterogeneous devices need to be orchestrated. For each of these scenarios, first we analyze the use case, with particular attention to the proper devices to be used, then we propose a mapping of the scenario to the WoT, and finally we deploy one or multiple orchestration policies through the WoT Store, as explained in the next point.

2. in order to address the question (ii), we carried out the design and the implementation of the *WoT Store*, a tool for the easy deploy and use of WoT scenarios. More in detail, the WoT Store is a novel software platform that supports the distribution, discovery and subsequent installation of applications for W3C-compliant Things. In particular, it allows the dynamic discovery of the resources available in the environment, i.e. the Things, and to interact with each of them through a dashboard by visualizing their properties, executing commands, or observing the notifications produced. In addition, similarly to popular app stores, the WoT Store allows the search and execution of third-party WoT applications that interact with the available Things again in a seamless way.

3. finally, for question (iii) we propose some improvements both for the standard and the official framework implementation for WoT [25], based on two main research directions: first, we study how to bring Deterministic Networks - in detail the Time-Sensitive Networking (TSN) - with QoS requirements in the W3C WoT. This is particularly interesting for several Industry 4.0 scenarios, like a production line, where different devices must be perfectly synchronized and have to deal with delicate operations, like the movement of a sophisticated robot. This leads to the introduction of new vocabularies, as well as the design and implementation of new protocol bindings for

the W3C WoT standard. Second, we analyze a dynamic WoT scenario, where Web Things and WoT services can be migrated at run-time following precise *migration policies*. To this aim, we augment the *node-wot* software stack with an additional *monitoring* layer needed by the service orchestrator to collect usage statistics about every service.

The rest of this dissertation is organized as follows: in Chapter 2, an introduction of IoT and the interoperability issues are given. Then, the Web of Things is presented with a special focus on the W3C WoT standard. We introduce the WoT Store in Chapter 3, with its design and architecture. After that, we validate the architecture components in two different scenarios, and we also provide a contribution to ease the porting of legacy IoT systems into the WoT Store environment. Instead, the effective validation of the whole WoT Store software ecosystem is investigated in Chapter 4 through two studies. In the first, we deploy the WoT Store in a heterogeneous environmental monitoring, where the sensing made by different Wireless Sensor Networks (WSNs) is directly orchestrated by the WoT Store. In the second, we apply the WoT Store to a Structural Health Monitoring (SHM) context, where custom sensors are deployed and information collected is processed in the WoT Store. Finally, in Chapter 5, we introduce the contributions about bringing the TSN into the WoT and about the migration of Web Things and WoT services, with a particular focus on the edge-cloud continuum. In Chapter 6 we conclude and present the future directions of our research.

Chapter 2

State of the art

2.1 Internet of Things

Nowadays, Internet of Things (IoT) is a word that indicates a consolidated paradigm that has rapidly gained importance in the last two decades. It encloses the idea of a pervasive presence of a multitude of interconnected *smart devices* - or rather *Things* - in almost every context. A Thing can be a tag, a sensor, an actuator, a mobile phone, or a simple `object` somehow computationally augmented. According to Szilagyi and Wira [26], an object does not necessarily need native computational and communication capabilities to be defined as a *Thing*. Instead, these capabilities can be provided later on by a chip attached and integrated into the object, in fact turning it into a *Thing*. This means that Things that compose the IoT could not originally be created with the purpose to be connected to the Internet, but that they can achieve this result later in time. Furthermore, a Thing, in order to be defined as such, must possess some interest for some services or applications: basically, it must do something useful for an IoT system. Clearly, there exist several kinds of Things, like the *Tags* (QR Code, RFID), *Devices* (for example Arduino, Raspberry Pis), *Machines* (Smart Bulb, smart car), or entire *Environments* (Smart building, smart city): all of them have different capabilities, especially in terms of computation and energy capacity. Once integrated into an IoT System, a Thing is typically characterized by (i) *communication*, (ii) *programmability*, and (iii) *sensing and/or actuating* capabilities. IoT has a concrete impact on several aspects of our lives and in different contexts, from domotics to Industry 4.0, from smart agriculture to healthcare, as better described later in Section 2.1.1. The literature about IoT is quite vast, and according to Atzori et al. [27], three main visions follow the IoT paradigm. The first vision can be defined as "Things oriented" and it is strictly related to the introduction of the "Internet of Things" word coined by Auto-ID Labs [28] at the end of the 90s. More in detail, Things are considered as very simple items, like Radio-Frequency IDentification

(RFID) tags. Their main goal is to improve the objects' visibility and traceability. In this sense, RFID is recognized as a catalyst for the realization of this vision [29] together with the use of Wireless Sensor and Actuator Networks (WSAN), mainly because of low costs and the maturity of these technologies. The same holds for the authors of [30], who in addition envisage a network of *everyday objects*. This is achieved by augmenting the Things' intelligence, defining the so-called *Smart Items*. These are not only sensors with the addition of usual wireless communication, memory, and elaboration capabilities, but also with autonomous and proactive behaviour, context awareness, elaboration capabilities. This perfectly matches the idea of [31], according to which: "from anytime, anyplace connectivity for anyone, we will now have connectivity for anything". This concept represents the joining link between a simple "Thing oriented" and an "Internet-oriented" vision of the IoT. In particular, according to the latter, the Internet Protocol (IP) represents the proper network technology that already lets a huge amount of different devices communicate and is also available for tiny and battery-supplied devices. This claim was originally made by the IP for Smart Objects (IPSO) Alliance [32], that since 2008 has been working on an IP adaptation for such devices, in particular with the focus on incorporating IEEE 802.15.4 into the IP architecture, in the view of 6LoWPAN [33]. The last main vision for the IoT is the *Semantic oriented* one [34] [35] [36]. The main idea is to take advantage of the semantic technologies to tackle the issues related to how to represent, store, interconnect, search, and organize information produced by the huge amount of devices spread everywhere. The main contributions in this sense are the possibility to meaningfully describe Things and to enable semantic reasoning over data previously collected. The Web of Things can be considered as the evolution of this last vision, as better explained in section 2.3.

2.1.1 IoT application domains

The great potentialities offered by IoT can be used to improve several aspects of our lives and our society. Furthermore, the characteristics of such paradigm can be easily adapted and instantiated in almost every scenario. The deployment of sensors that communicate with each other enables the possibility to collect data and hence to perceive the surrounding environment. This data can then be used for any kind of analysis, like real-time or predictive, and if actuators are deployed as well, this implies that actions can be immediately and automatically performed as a consequence of some kind of reaction. Clearly, on top of this basic pipeline, thousands of different applications can be implemented. As shown in Figure 2.1, we can divide them into eight macro domains: (i) transportation domain, (ii) vehicles domain, (iii) industry domain, (iv) healthcare domain, (v) agriculture domain, (vi) smart home domain, (vii) smart cities domain, (viii) commercial domain.

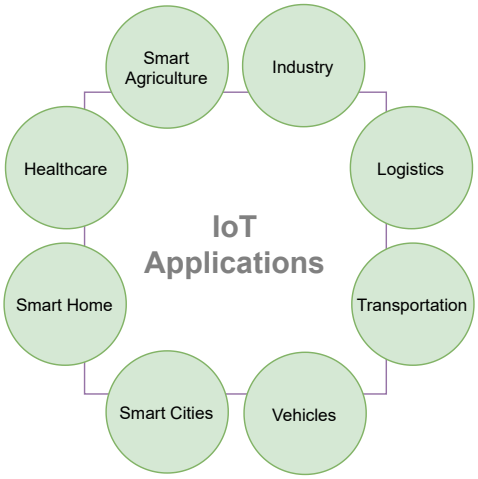


Fig. 2.1 Taxonomy of IoT application domains.

Industry and logistics domains

The goal of applying the IoT paradigm to *logistics and Industry* domains is to improve the workflow [37]. This can be realized by keeping track of every step of a process in order to analyze it and understand where it can be enhanced. In industrial scenarios, tags can be associated with what needs to be monitored. After each step of the production, the tag is read by a sensor that can collect information about that specific object. Another interesting scenario is the monitoring of goods transportation. Goods are associated with tags that can be read by sensors put on the vehicles used for transportation. Through the vehicle's GPS, products can be monitored in real-time during their travel, as well as in the moment they reach the destination.

Transportation and vehicles domains

Cars, trucks, trains, bicycles, and buses can be equipped with sensors, actuators, and processing units to monitor the status of transported goods, improving driving comfort, or suggesting better routes in real-time. *Smart Transports* [38] are radically changing the way of thinking about transports. The chance to spread sensors and actuators over roads and rails means that people can control transportation vehicles to better route the traffic, provide tourists with real-time transportation information, help in the management of the depots, and monitor the status of transported goods, especially if they are equipped with some tracking tags. But Smart Transports also include the assisted-driving world and the automatic one. Research in these fields is extremely important nowadays, with companies like AlphaBet and Tesla, for instance, investing billions of dollars in self-driving cars.

Healthcare domain

In the healthcare environment, there are many benefits [39] that can be provided by IoT and that - according to [27] - can be grouped mostly into:

- *Tracking*: like in all the other domains, the most useful benefit of tracking is related to the possibility to monitor and improve the workflow. In such sense, it means being able to follow patients in hospitals, or checking the access to designated areas, as for instance operating theaters.
- *Identification and authentication*: by the patients' point of view, the identification is useful for reducing incidents that can be very harmful to them, like wrong drug prescription, dosage, or procedure. From the staff's point of view, identification and authentication grant access to specific areas in a more comfortable and efficient way.
- *Data collection*: automatic data collection reduces processing time and helps improve the goods supply, by making it faster and more efficient.
- *Sensing*: one of the most interesting fields where IoT is improving healthcare is sensing. Since sensors have become very small and less invasive, some studies investigate the possibility to place them inside a human body in order to provide real-time information on patient health indicators. These values can be monitored remotely and/or by the patient using, for instance, a smartphone.

Smart Cities domain

The IoT paradigm can also be applied to *cities*, introducing several benefits [40]:

- *Structural Health of Buildings*: different kinds of sensors can be used in order to monitor the health of buildings, especially historical ones. Vibration and deformation sensors are suitable for monitoring a building's stress level, atmospheric agent sensors in the surrounding areas for pollution levels, and temperature and humidity sensors for the environmental conditions [41]. This particular case is also deeply investigated in Section 4.2.
- *Waste Management*: in order to improve the quality of recycling and to reduce the cost of waste, cities can use smart waste containers, which detect the level of load and allow for an optimization of the collector trucks route. Sensors can detect how much a container is full and can communicate it to the people in charge of the waste service.

- *Monitoring*: some important factors can be monitored in cities, like the pollution level (or the quality of the air), and the noise rate. Also in this case, sensors are used in specific positions of the city in order to collect reasonable data about what is monitored. Data collected together can give an overview of such levels, letting the city government know right away about a problem and supporting the decision-making process.

- *Smart Lighting*: since the optimization of costs is fundamental for public administration, the smart management of resources like electricity can save a lot of funds. Smart lighting reduces electricity consumption by switching lights on and off or optimizing their intensity depending on some precise factors and smart policies. For instance, lights can be immediately turned off if there is no presence of people nearby, or in the morning after a specific value of natural light has been reached.

- *City Energy Consumption*: related to energy optimization, cities [42] can monitor the instant energy required by different services (public lighting, transportation, traffic lights, and so on) in order to set priorities in resource management.

Smart Home domain

Sensors and actuators spread over the house/office can help people make their lives more comfortable, most of the times by automating several processes that are usually done by hand. Room lighting can be automatically adjusted depending on the time of day, the heating system can behave according to the residents' preferences, and possibly everything can be managed remotely by a smartphone. By using this kind of systems, people are also able to save money and energy since they can manage their resources according to some optimized policies. In the last few years, some of the biggest vendors in the ICT world, like Google [1], Apple [2], Amazon [3], Microsoft [4], released products for the IoT market, often focusing on the domotics field. Most of them are plug-and-play but require important investments at the beginning, while homemade solutions are quite cheap but require more effort and technical knowledge.

Smart Agriculture domain

Smart Agriculture leverages IoT solutions to implement the so-called Third Green Revolution. In this context, ICT technologies face unique challenges like, among others and just to name

[1] https://developers.google.com/weave
[2] www.apple.com/lae/ios
[3] www.amazon.com/iot
[4] https://azure.microsoft.com

a few [43], the lack of a stable power supply, the need for calibration procedures customized for every type of soil, the security concerns from farmers. Various sensors and actuators can be deployed to enhance the productivity and the economical gain of the farm, like for instance soil moisture sensors, drones, automatic sprinklers, and cameras. Using the data flow analysis, agronomists or soil experts can create calibration models for soil moisture sensors to better fit the real volumetric water content and hence saving the total amount of water used for irrigation.

2.2 IoT Interoperability

A key role in the success of IoT is played by the full interoperability among interconnected devices, still guaranteeing them a high *degree of smartness*, together with the respect of privacy and security principles. Interoperability is intended as the ability to exchange and make use of information among heterogeneous devices, i.e., that use different protocols and technologies. As pointed out by Van Der Veer and Wiles [44] and Serrano et al. [45], interoperability can be classified into the following categories:

- **Technical Interoperability**: it is usually reached with hardware/software components, systems, and platforms that directly enable a *machine-to-machine* communication. This interoperability is based on protocols and infrastructures in order to take place.
- **Syntactical Interoperability**: it is usually focused on data formats. In particular, since many protocols carry data or content, this should be represented using high-level transfer syntax such as HTML, XML or JSON
- **Semantic Interoperability**: it is usually based on sharing the same meaning for exchanged data. This concerns both the machine and human layers
- **Organizational Interoperability**: it is usually based on the fact that organizations effectively share meaningful data and information even if this implies several information systems over widely different infrastructures. This kind of interoperability strongly depends on the technical, syntactical, and organizational ones.

Solutions to support interoperability on IoT scenarios have been largely investigated by the academic research as well as by European projects: we cite, among others, the projects Arrowhead [46], BIG IoT [47] and Wise-IoT [48] that proposed reference platforms to connect and deploy cross-domain IoT applications. As extensively surveyed in [7], interoperability can be seen from different points of view, in particular by the (i) device, (ii)

network, (iii) semantic and (iv) cross-domain perspectives. Device and network solutions (e.g. the 6LoWPAN stack [49]) face the existing fragmentation by defining common addressing, routing, and data-exchange rules. Vice versa, semantic solutions focus on the definition of a common data model used by the IoT interacting components; to this aim, several IoT-related ontologies have been proposed [50]. Cross-domain solutions (e.g. [51]) represent the most general way of supporting interoperability on the IoT: rather than focusing on protocols and data, they aim to define common interfaces that should be implemented by the IoT components, in order to be discoverable and queryable.

García Mangas and Suárez Alonso [52] make an interesting separation between solutions for interoperability that use a classic approach (non-WoT) and the ones that instead are based on the Web. The Open Connectivity Foundation (OCF) Core specification [53] provides explicit descriptions (profiles) for resources and devices depending on the different vertical markets they are used on, like for instance *smart home*, *industrial*, *healthcare*, just to cite a few. From a technical point of view, it is based on a REST architecture with CoAP (with both UDP or TCP) and CBOR serialization. Both REST and CoAP are also proposed by LwM2M [54], despite only UDP is taken into account. All LwM2M clients also act as servers, since they expose object instances containing their set of resources. A similar approach is also considered by the *servient* architecture in the W3C Web of Things [2]. The oneM2M [55] standard is based on three main functional entities: *Application Entity (AE)*, *Common Services Entity (CSE)*, and *Network Services Entity (NSE)*. The first is an entity in the application layer that represents the M2M application service logic. The second entity contains a collection of oneM2M-specified common service functions that AEs are able to use. These service functions are then exposed to be used by other entities. The third one is the entity that provides services for the network layer. OneM2M follows REST as architectural style and uses several application layer protocols (HTTP, CoAP, MQTT, WebSockets). Kamienski et al. [56] introduce IMPReSS Systems Development Platform (SDP) whose goal is to accelerate and facilitate the development of IoT applications. The core of this proposal is the layered architecture: the top layer is represented by a reusable User Interface (UI) set of components, the middle one consists of a set of middleware APIs useful for implementing typical IoT services (like storage), while the last layer includes several adaptors for concrete IoT devices and platforms, providing in this sense the possibility to interconnect heterogeneous systems and hence enabling IoT interoperability.

2.3 Web of Things

Although no standard definition exists [57], the Internet of Things is considered as the enabler of intercommunication among heterogeneous devices, while the Web of Things aims at bringing and solving the interoperability problem at the application layer. Despite this separation line in their principles, these two worlds often overlap, making their distinction not clear enough and generally creating some confusion. As highlighted by Negash et al. [58], in contrast to the current Internet - also called *Internet of people* [59] -, the Internet of Things, and hence the Web of Things, enhances the real world by adding connected devices capable of sensing and acting through the Internet. Nevertheless, since the beginning the Web of Things has been considered as an extension of the IoT: the main idea is the possibility to adopt Web standards and Web technologies in order to map smart things to web services, hence building new kinds of applications and services based on devices' capabilities. Continuous improvements both in electronics and in software have led to a new generation of small, cheap, and low-power devices. As pointed out by Raggett [60], this paves the way for a Web of Things world, where "Things" can be considered as proxies for physical or abstract realities and the "Web" refers to the possibility that these "Things" communicate via Web technologies, like for instance HTTP.

The concept of the Web of Things starts to appear at the beginning of the 2000s, when Kindberg et al. [61] presented the idea of *Web presence*, an extension of the *Home page* concept to also include all physical entities. *Web presence* can be considered as a description model of a Thing and its entry point for interacting with it. This is obtained by embedding web servers into the Things - like printers, projectors, whiteboards - or by hosting their *web presence* within a web server. Authors also set the basis for a location-aware system by proposing an infrastructure where URLs are used to address the *Web presences*, localized web servers function as directories for URL sensing, and beaconing is used for the discovery. In 2007, another study was made by Wilde [62] where the author proposed to assign a URI to each device - for instance to each sensor of a sensors network - and to interact with it through the *basic verbs* of REST, i.e., *PUT, GET, POST, DELETE*. Each device can be represented as Web resources by using *HTML, XML or JSON* format. Following the same principle, Guinard and Trifa [63] defined the Web of Things as "a refinement of the Internet of Things by integrating smart things not only into the Internet (network), but into the Web Architecture (application).". The same authors in [64] claimed that the realization of the WoT needs to extend the existing Web by involving real-world objects and embedded devices. Instead of just using Web protocols as transport protocol - like most of WS-* Web services -, they intended to make devices an integral part of the WoT by using HTTP as application layer, as also analyzed in [65]. This can be achieved by making devices' functionalities

available through RESTful APIs over HTTP and this can be done in two possible ways: *Direct integration* or *Indirect integration*. In the first approach, devices directly embed a web server - becoming already part of the Web -, while in the second one an intermediate *Smart Gateway* is in charge of translating requests/responses across protocols. This last vision is also considered by Karim et al. [66], where a monitoring system for smart agriculture is presented: a smart gateway is responsible of collecting data from a Wireless Sensors Network (WSN) and share it over the Internet in order to be analyzed and visualized by means of a Web application. However, in this work authors seem to limit the role of the Web in WoT to be only a nice GUI for an Internet of Things system.

Social Web of Things *Social Web of Things* can be considered as the convergence of the Social Web and the Web of Things. More in detail, it enables users to manage, access, share and integrate smart Things/objects through Social Network Sites (SNS) [67]. SNSs are online platforms where people publish, collaborate, and share information with other individuals or groups and build social relationships. Ciortea et al. [68] define the Social WoT as the convergence of three dimensions: *pervasiveness*, meaning that the Web extends to the physical world by integrating everyday objects and things. *Pro-activeness*, since the Web is composed of several proactive entities that, exactly like normal users, produce and consume content by interacting with each other. Finally, *Socialness*, because the Web centers around its entities both humans and non-humans.

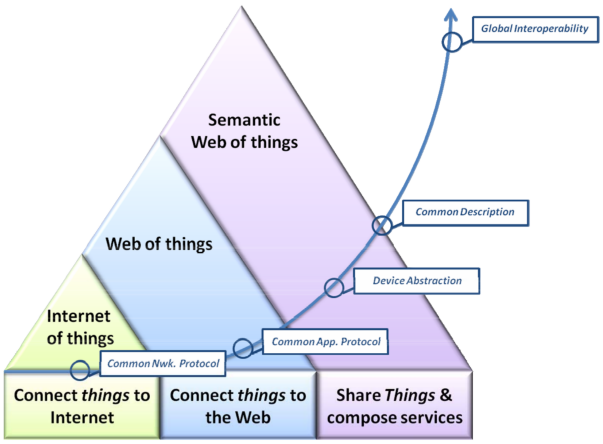


Fig. 2.2 WoT evolution from [1].

Semantic Web of Things Technologies for Semantic Web [69] can play an important role in the Web of Things, and in fact they can be considered as the interoperability enablers for some WoT environments. Researchers have formulated several proposals for including

the Semantic Web in the WoT architecture, forming the so-called *Semantic Web of Things (SWoT)*.

As shown in figure 2.2 and highlighted by Jara et al. [1], integrating the Semantic Web into the Web of Things is the last step for reaching what is defined as *global interoperability*. More in detail, this can be obtained if information is semantically enriched and systems can achieve high degrees of autonomic capability to discover, manage and store information and to provide transparent access to information sources in a given area. Clearly, data in this way must be machine-understandable. The first phase can be seen as the fundamental step of interconnecting everything to the Internet and can be mapped into the Internet of Things growth, while the second one had the goal to enable seamless interoperability among heterogeneous entities by the adoption of common application protocols, i.e., web protocols. Despite the fact that the WoT allows almost all kinds of devices to communicate with each other thanks to web technologies, shared data can be represented in very different ways, bringing different meanings and hence basically disabling interoperability at the data layer. For this reason, the main goal of SWoT is to face this problem by defining a common description that allows data to be universally understandable, creating extensible annotations (from minimal to more elaborate ones), ad agreeing on a catalogue of semantic descriptions (ontologies). It is then important to set up ontologies to regulate the relations among Web-enabled devices (or better *Web resources)*.

2.3.1 Requirements

In the vision of Heuer et al. [70], applying Web Technologies to the information exchange among Things and creating *Thing-to-Thing mashups*, should enable an analogous potential offered by the WWW in sharing information. They found four main questions to which researchers need to answer in order to enable such innovation: (i) how interactions happen between Things and the physical world, (ii) how to take advantage of constrained devices - that mainly compose the IoT layer - that have limited resources, both in terms of power and capabilities, (iii) how to manage sporadic user interactions or event-triggered updates, and (iv) how to handle continuous multiple dataflows. These questions can then be summarized into three main challenges that need to be faced to bring the IoT into the WoT:

1. the real world needs to be mapped and described through existing devices, i.e., smart things, in an effective way and possibly covering all the aspects that can be of interest

2. mashup applications need to take into account the device constraints and hence they should be tuned depending on the scenario

3. communication strategies (protocols and technologies) need to be designed and adapted according to new kinds of demands

Thing classification is proposed by Mathew et al. [71]: considering the high degree of variation, Things need to be abstracted and classified in order to be represented in the Web. For this reason, they identify four main dimensions that characterize the Things' capabilities: **Identity**, **Processing**, **Communication**, and **Storage**. The first one refers to the need of a Thing to be uniquely identifiable through the use of an appropriate identification system, like *Barcode*, *RFID*, or an *IP address*, that can be used to address and access the Thing as a unique resource. Clearly, this is a mandatory minimum requirement to be respected in order to be integrated into the Web world. The second aspect represents the processing capability of a Thing, i.e., a system that defines the functionalities of the Thing. The third one identifies the communication interfaces available for communicating and interacting with other Things. A Thing exposed in the Web (as a web service) should also provide a minimal set of APIs to interact with it. Each interface requires precise inputs/outputs and has its own set of requirements to be respected in the communication, like for instance the *medium*, *protocol* or *privacy policy*. According to Kamilaris et al. [72], a Web platform can be identified by the following elements:

- **Integration of things to the web**: Things should be accessible at the Web layer. Several architectures have been proposed (see section 2.5), mainly divided into architectures based on gateways/proxies for bringing Things on the Web, or architectures where Things are already web-enabled by embedding a web server
- **Device discovery**: Things can be automatically discovered by agents by means of specific architectural patterns or protocols
- **RESTful interaction with the things**: Things can be accessed through a RESTful interface
- **RESTful interaction with the platform**: Platforms can be accessed by clients through a REST interface
- **Data formats**: which well-known Web data formats are used
- **Multiple representations**: multiple data formats are offered by the platform and chosen by the client
- **Security**: well-understood web security protocols are used to enable the security level (for example HTTPS, OAuth, Bearer)

- **Service semantics**: semantics can be offered to enhance the description of services provided by the platform
- **Data semantics**: semantics can be offered to enhance the meaning of exchanged data
- **Physical Mashups**: new services and functionalities can be enabled by *mashups* application, i.e., applications that can be written in any programming language supporting Web protocols and that involve multiple Web Things
- **Sharing**: Access to Web Things and services can be shared through several instruments, like social networks
- **Syndication techniques and/or web messaging**: Interactions with Web Things and services can be done on a publish-subscribe matter

A smart object classification is made by Mrissa et al. [73]. More in detail, they distinguish the objects into the following categories. *Resourceful objects* already host all the services they provide by means, for instance, of a WoT platform that they can embed. The installation is pretty simple, since they are standalone and hence do not require additional infrastructures. *Resource-constrained objects* are characterized by restricted resources and must rely on external components that can help them to cover all the WoT platform stack. Finally, *Resourceless objects* are basically passive objects that can be identified by unique identifiers like QR codes or RFID tags. Given the lack of any computation, storage, and memory capability, they need to be augmented through specific software deployed on cloud or local network gateways. Because of the lack of standard specifications for developing WoT applications, authors also highlight which design requirements are important for WoT software platforms. In particular, they identify the following issues: *interoperability* should be guaranteed in order to let applications interact with heterogeneous physical objects. The *Live reactive* requirement is intended to make the platform dynamically adapt its behaviour and structure to the environment at run-time. This can be achieved only if the platform implements a strong *resource management* and is able to react to possible connectivity disruptions that could happen among devices (*disconnection tolerance*). The platform must be reliable and secure and hence responding to the *safety* requirement. Furthermore, it should be carefully chosen where to deploy and execute each task (*delegation*), instead of using only cloud infrastructures. A WoT platform should provide useful services for users, i.e. *user-understandable* services, and these services/applications should allow a set of objects to *collaborate*. In the Raggett [60]'s proposal, the idea is to enable *worldwide discovery and interoperability* by exposing the already existing IoT platforms through the Web, by respecting the following requirements. URIs are used for those things that serve as proxies for

physical and virtual entities; there should be a way to retrieve things' metadata in a standard format, like for instance JSON-LD. Things' owner, purpose, access control, and relations to other things should be explicitly described, and all the things should be modeled according to the Properties, Actions, and Events (PAE) paradigm. Each property represents a discrete value that can smoothly change between data points. Finally a variety of communication patterns, like request-response or publish-subscribe, should be available for the interactions with the Things.

2.4 Web of Things: definitions

2.4.1 Web Thing

The concept of the Web Thing definition comes together with its description model. In particular, several models have been proposed over the years, but the most important can be considered the one proposed by Trifa et al. [74], and the one by the W3C Working Group [2]. In the first case, a Web Thing - or simply Thing - is "a digital representation of a physical object accessible via a RESTful Web API". The core part of the definition is represented by the *RESTful Web API*, that they consider as hosted on the Thing itself or on an intermediate host in the network such as a Gateway or a Cloud service (for those Things that cannot communicate through the Internet). In the second case, a Web Thing, also referred to simply as a Thing, is defined as whatever entity can be semantically described. More precisely, and reporting the working group's words: a Thing is "an abstraction of a physical or a virtual entity whose metadata and interfaces are described by a WoT Thing Description, whereas a virtual entity is the composition of one or more Things." [3]. A Thing can be a device, a logical component of a device, a local hardware component, or even a logical entity such as a location (e.g., room or building).

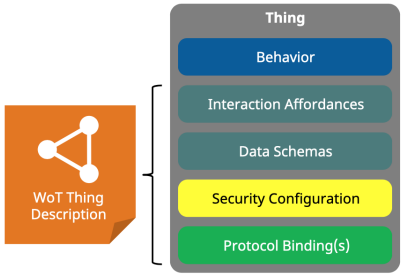


Fig. 2.3 W3C Web Thing architecture proposed in [2].

Models and Thing description In order to interact with existing IoT devices and services of various siloed ecosystems, W3C WoT leverages descriptive metadata, following the architectural style of the Web, i.e. REST [75], with a focus on *hypermedia controls* (cf. HATEOAS), and consolidated Web technologies like JSON [76] and Linked Data [77]. Metadata is serialized into a machine-understandable, but still human-readable WoT Thing Description (TD [78]), a Web *representation format* based on JSON-LD [79]. One of the key parts of the Thing Description is represented by the concept of *Affordance*. Norman [80] states that *"Affordance refers to the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used."*. Hence, the description of the Thing Affordances into a TD makes the metadata self-descriptive, so that consumers (users of a Thing) are able to identify *what* capabilities a Thing provides and *how* to use them. Furthermore, in the context of REST [81], the term was adopted in the definition of hypermedia: *"the simultaneous presentation of information and controls such that the information becomes the affordance through which the user obtains choices and selects actions."*. Affordances of Web Things consist of the information encoded in the high-level interaction endpoints *Properties*, *Actions*, and *Events* (*what* capabilities) and the controls encoded in their forms (*how*), defining the so-called PAE paradigm. The Thing Description is hence the entry point of a Web Thing, describing four of its five main architectural blocks of a Thing depicted in figure 2.3. The fundamental blocks of a Thing are:

- the *Behaviour*: it coincides with the Thing Application (TA), i.e., the application where both the autonomous behaviour of the Thing and the handlers for the Thing affordances are implemented.
- *Interaction Affordances*: it is basically the interaction model of the Thing, specifying how consumers can interact with it through abstract operations and without referring to a specific network protocol or data encoding. As previously introduced, it follows the so-called Properties, Actions, and Events (PAE) paradigm. Each *property* is considered as a state of the Thing and can be retrieved and possibly updated through a writing operation. A *property* can also be `observable`, and in this case the Thing is responsible to push the new state after the change to each consumer. An *action* basically allows to invoke a function of the Thing, that typically manipulates a Thing state or launches a process. An *event* describes an event source that can asynchronously push data to a consumer.
- *Data Schemes*: they represent the information model (with the related payload structure and data items) to be used in the interaction between Things and Consumers of the Thing.

- *Security Configuration*: it contains the security mechanisms provided by the Thing in order to control access to its Interaction Affordances. The security configuration includes the *Public Security metadata* and the *Private Security Metadata*. The first is a component that describes the mechanisms and the rights for accessing a Thing, but without including any secret or sensitive data. Therefore, it does not provide access to the Thing by itself, but instead it only describes how to grant access to the Thing, including any authentication mechanism. The second is a component of a security configuration that is kept secret since it contains sensitive data to get/obtain access to the Thing. This data is not shared with other devices or users.

- *Protocol Bindings*: they map each Interaction Affordance to messages with a specific protocol and they are serialized as *hypermedia controls*

Apart from the *Behaviour*, all the other blocks are described in the WoT Thing Description (TD). The TD is a fundamental architectural block of the entire W3C Web of Things architecture, as better described in section 2.5.5. It is considered as the entry point of a Web Thing, like the *index.html* for a website.

Listing 2.1 shows a TD sample for an Air Conditioner system. The conditioner has a property called *on/off* - to represent the current state of the conditioner -, an action called *toggle* - to toggle the conditioner -, and the *temperatureThreshold* event that is fired whenever the desired temperature of the room is reached. Based on the fact that *Forms* in HTML are controls for constructing dynamic requests on the client side (based on choices given by the server), W3C WoT also takes this approach and makes forms also machine-understandable following the attribute concept of Web Linking by providing a form context, an operation type, a submission target, a request method, and optionally form fields. *Protocol Bindings* define the mapping between affordance and concrete protocol message and, practically, they must provide a vocabulary for the form attributes of a specific protocol together with default assumptions and behavioral assertions for that protocol.

```
{
    "@context": "https://www.w3.org/2019/wot/td/v1",
    "id": "urn:dev:ops:32473-WoT-AC",
    "title": "MyAC",
    "securityDefinitions": {
        "bearer_sc": {
            "in": "header",
            "scheme": "bearer",
            "format": "jwt",
            "alg": "ES256",
            "authorization": "https://servient.example.com:8443/"
        }
    },
    "security": ["bearer_sc"],
    "properties": {
        "on": {
            "title" : "On/Off",
            "type": "boolean",
            "forms": [{"href": "https://myac.example.com/on"}]
        },
    },
    "actions": {
        "toggle" : {
            "forms": [{"href": "https://myac.example.com/toggle"}]
        }
    },
    "events":{
        "temperaturebThreshold":{
            "data": {"type": "number"},
            "forms": [{
                "href": "https://myac.example.com/threshold",
                "subprotocol": "longpoll"
            }]
        }
    }
}
```

Listing 2.1 Thing Description sample for an Air Conditioner

2.4.2 Mashup application

A *Mashup* application is considered as an application capable of enabling new kinds of services and providing new kinds of functionalities in a certain system by combining data

from multiple sources into a new single output based on specific criteria. For the WoT, this translates into having an application that takes data from multiple Web Things and arranges it in order to obtain a specific output useful for some purpose. A typical example is an application that queries multiple Web Thing sensors spread over a large environment and simply returns the average. Mashup applications can of course involve actuators: in this case, depending on the mashup logic, actions can be performed on target Things. For instance, let us assume that we have instantiated a Web Thing thermometer and several smart heat controllers in a restaurant for adjusting the temperature. The rationale behind the temperature regulation can be implemented into a *mashup application*, that first needs to retrieve the current temperature value by probing the Web Thing thermometer and then sends to each controller the right adjustment. According to Guinard and Trifa [64] and [82], *mashup applications* can be mainly divided into two categories: *physical-virtual mashups*, also called *cyber-physical systems* [83], and *physical-physical mashups*. In the first case, applications are meant to manage data coming from computation and physical processes. In the second case, applications combine real-world services provided only by physical devices.

2.5 Architecture

The architecture proposed by Guinard et al. [84] can be considered as a reference for all the others. The main contribution in this sense is to arrange the already existing Web architecture to also include the WoT. In particular, instead of focusing only on reaching an Internet-level connectivity (often in terms of TCP and/or UDP)- as mainly done in the Internet of Things -, they propose to re-use the REST architectural style to build interactions with smart things around universally supported methods [85]. Devices do not necessarily need to have natively RESTful interfaces, but instead they can take advantage of *intermediaries* like proxies or reverse proxies to be wrapped in RESTful abstractions. Clearly, since REST is based on the concept of *Resource*, the first step is to identify the smart things themselves as resources. Despite REST being an architectural style, therefore not bound to specific technologies, the authors propose to respect the five constraints of REST by adopting the same technologies used for making the Web a RESTful system. More in detail, they identify the following constraints:

- *Resource Identification*: resources are identified by *URI*, thus links to resources can be established using a well-known identification scheme
- *Uniform interface*: resources are accessible through a uniform interface with a well-defined interaction semantics, like Hypertext Transfer Protocol (HTTP). HTTP offers

a very small set of methods with different semantics that make interactions optimized. Almost all web applications have a RESTful interface (while the back-end is implemented following other interaction models), and the same idea can be employed for the Web of Things

- *Self-Describing Messages*: using standardized representation formats avoids individual negotiations between servers and clients. On the WEB, media type support in HTTP and the Hypertext Markup Language (HTML) allow peers to cooperate without individual agreements, while instead formats like JSON (and JSON-LD) are widely used for machine-oriented services

- *Hypermedia Driving Application State*: clients of RESTful services can explore the services without the need for specific discovery capabilities, but just by following links they found in resources. In this sense, the *Resource Identification* and the *Self-Describing Messages* constraints play a fundamental role for this kind of operation

- *Stateless Interactions*: all the information needed for the request must be part of it, meaning that requests are self-contained. This concept is part of HTTP since it only considers the request/response interaction pattern. It is important to note that other mechanisms - like cookies - can be used to obtain stateful interactions between client and server.

Starting from this idea, a four-layer architecture has been proposed, as shown in 2.4. The first layer is meant to turn every Thing into a Web Thing, granting interactions with it through HTTP requests or Web Sockets, i.e., basically making each Web Thing a REST API to allow interactions in the real world. The second layer has the goal to ensure that Web Thing can be automatically discovered and used by other WoT entities. Authors propose to re-use Web Semantic standards to describe Things and Services in order to enable searching for things through search engines and other web indexes as well as the automatic generation of user interfaces or tools to interact with Things. At this layer, different Thing models can be considered for defining abstract sets of interfaces and resources that Things should expose, as previously explained in section 2.4.1. The third layer is in charge of sharing data across Web Things. In particular, it defines how data produced by Things should be shared in a secure and efficient way. Clearly, at this layer all the Web security protocols, like TLS, or security mechanisms, like OAuth, can be re-used to ensure secure communications. Social networks can play an important role in order to share data, as previously described in section 2.3. Finally, the fourth layer helps to build large-scale and meaningful applications for the Web of Things by providing useful tools and mashup platforms (see 2.4.2).

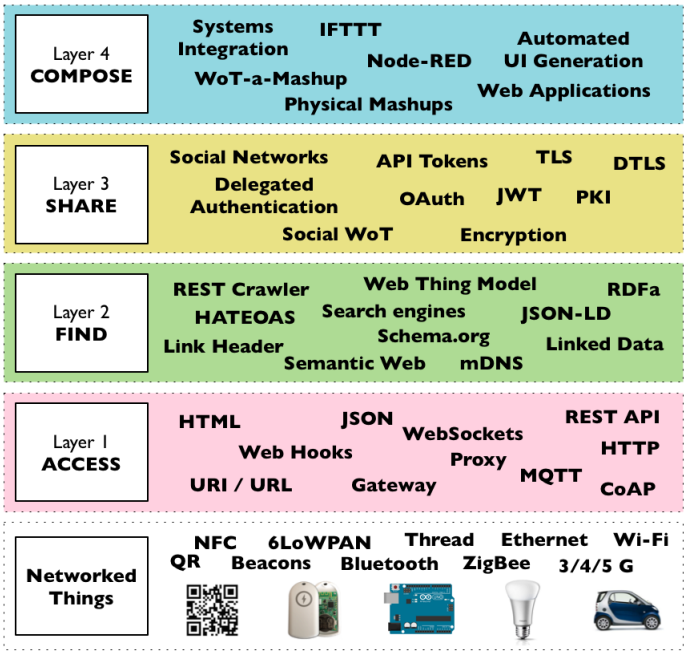


Fig. 2.4 Web Thing architecture proposed in [3].

2.5.1 Access Layer

In order to enable Web Things to communicate over the Internet, two main possibilities are highlighted by Guinard and Trifa [64]: *Direct Integration* and *Indirect Integration*. In the first kind of integration, Web Things are meant to be already IP-enabled and to own enough resources to host a web server. The web server is in charge of providing capabilities to understand and directly speak Web languages and protocols. At the same time, they can expose the APIs needed to interact with the Web Things. Each Thing has an IP address and on top of the Web server the RESTful APIs are used by the mashup applications. In the second integration, because most of the times devices do not own enough resources to host a web server, an intermediary gateway is needed in order to bring devices at the Web layer. Smart Gateways have the main goal of exposing the communication interfaces of Things through RESTful APIs, by abstracting proprietary communication protocols or custom APIs of the embedded devices. In particular, requests coming to the RESTful APIs are mapped by the gateway to the proprietary API that are then transmitted by using communication protocols that devices can understand. Gateways can also be used to arrange several data sources of different devices into a higher-level web service, so that *mashup applications* can easily use device-level services. In this sense, a gateway can also serve as an orchestrator of services. Trifa et al. [86] propose an architecture for smart gateways composed of three major layers: the presentation layer, the control layer, and the device abstraction layer. The first

layer is in charge of making the gateway components accessible from the web. It manages requests coming from clients through a REST interface and handles different datatypes for resources. Furthermore, in this layer each device's name is mapped to a URI in order to make every device connected to the gateway accessible on the Web. The second layer is composed by several sub-components that are defined as *plugins*. Plugins are loaded at the boot phase and custom plugins can be written in order to extend the functionalities of the gateway. Two important plugins are always included: the *Device management* and the *Eventing* plugins. The first one is meant to maintain a high-level view on devices registered at the gateway by using the device abstraction, while the second one is used to map sensors' updates to publish/subscribe events, hence avoiding polling made by the gateway. The third architectural layer is responsible for creating an abstraction for each device, for making them look the same at the higher levels, although their underlying implementations might differ. On these bases, Guinard et al. [87] bring RFID tags to the Web by adapting and designing a RESTful architecture for the Electronic Product Code Information Service (EPCIS). In this way, each tagged object, location or RFID reader gets a unique URL that can be used by web applications - like mashup applications - or easily shared across the Internet. Basically, a *RESTful EPCIS module* translates the incoming requests into WS-* requests, since in the EPCIS standard most features are accessible through a WS-* interface.

2.5.2 Find Layer

Nadim et al. [88] face the challenge of WoT service discovery by proposing a distributed WoT service semantic discovery architecture that leverages three services filtering mechanisms: clustering, indexing, and ranking, which are semantic annotation-based. Furthermore, the architecture also deals with the dynamism of WoT services - thanks to an incremental clustering algorithm - and the mobility of IoT gateways - thanks to WoT gateways. More in detail, the main strategy is to reduce the number of services by aggregating them: geographic location, measured property, the unit of measurement can be used as semantic features for aggregation. Most of the discovery services for the Web of Things rely on Semantic Web technologies. Nadim et al. [88] propose a distributed three-layer architecture to enable semantic discovery for dynamic environments in the Web of Things. In particular, the architecture can handle the dynamism of WoT Services thanks to a clustering algorithm and the mobility of gateways through the use of *WoT Gateways*. The first architectural layer can be considered as the same presented in Section 2.5.1, with a *smart gateway* whose main tasks are: (i) letting devices communicate with the Web layer, (ii) registering, managing, and controlling the connected objects, (iii) aggregating data coming from devices, (iv) publishing and provide devices' capabilities like web services, with specific Open APIs. The second

and the third layers can instead be considered as a set of functionalities belonging to the *Find layer*. In particular, the second layer is responsible of hosting the *WoT gateways*, the discovery functions, the IoT gateways management and the service composition functions. Each WoT Gateway contains the semantic description of the associated IoT service. Finally, the third layer contains discovery and indexing servers, whose goal is to formulate the query that can be submitted to the WoT Gateways, manage geo-spatial index of WoT gateways to discover suitable WoT Gateways based on geographical features, and to rank/visualize the results according to user preferences. Mayer and Guinard [89] propose DiscoWoT, an extensible semantic discovery service for Web-enabled smart things where multiple discovery strategies can be applied to a Web resource representation. Furthermore, users can create and update their custom strategies in an easy way, since the final aim of the study is to facilitate the discovery, selection, and utilization of smart things. From an architectural point of view, DiscoWoT has been designed to be a standalone RESTful service, with a *representation layer* that retrieves the Web resource representation and a *semantic layer* that is in charge of applying and implementing the discovery strategies. The WOT Semantic Search Engine (WOTS2E) is the proposal of Kamilaris et al. [72] that aims at realizing a scalable and flexible way to discover almost in real-time web connected embedded devices and their semantic data. WOTS2E is a search engine for the Semantic Web of Things that uses a web crawler to discover Linked Data endpoints and, through them, web-enabled devices. The main component of the architecture is the *Discovery Module*, that supports four steps: firstly, it continuously scans the web to discover Linked Data endpoints through several web crawlers. Secondly, it examines each discovered endpoint to understand if they contain IoT/WoT datasets and ontologies and hence can be considered as Web things. Thirdly, it analyzes the endpoints to acquire information about which IoT/WoT services are provided by the devices. Fourthly, IoT/WoT services discovered are recorded along with their semantic description. Ruta et al. [90] propose a general framework for the Semantic Web of Things that bases on an evolution of classic Knowledge Base models. The adopted architecture also supplies solutions for information storage, communication, and processing. A Knowledge base is the combination of an ontology and a set of asserted facts, from which additional knowledge can be inferred. Although it is usually considered a fixed and centralized component, in this case it becomes a pervasive element, since ontology files can be managed by multiple nodes in a Mobile Ad-hoc NETwork (MANET). Individual resources are instead spread across the environment, being physically bound to the microservices deployed in the scenario. The proposed architecture consists of two levels: the *field layer*, and the *discovery layer*. The first one is in charge of interconnecting the micro-devices of the environment with hosts that are able to extract their data. The second one provides means for knowledge dissemination

and retrieval, hence enabling inter-host communication among devices. More in detail, each network host acts as a cluster head for all the devices in its communication range, using all the available network interfaces (RFID, ZigBee). All the resources acquired in the first layer through different protocols are then shared with the second layer in a uniform way. In particular, the following steps are proposed for the whole process of collecting and then exposing semantically enriched resources: (i) extraction of resource parameters (objects characteristics are shared with the *discovery layer*), (ii) resource information dissemination (diffusion of resource characteristics at the discovery layer), (iii) peer-to-peer collaborative resources discovery, and (iv) extraction of selected resource annotations, that can be used for semantic-based queries and reasoning. The proposed framework provides services for accessing information embedded into semantic-enhanced micro-devices, while information processing and reasoning tasks are executed on local hosts or by a remote entity through a gateway exposing a high-level interface, like for instance in the web. In the Semantic web stack for the IoT proposed by Szilagyi and Wira [26], the *Semantic layer* is expanded into three sub-layers: the *modeling level*, the *data processing level*, and the *IoT Services and Application*. In the first case, semantic web technologies are used to provide a common understanding of Things' capabilities and characteristics, in particular by employing shared vocabularies and ontologies to guarantee the integration of heterogeneous data generated by different systems. The second case level is meant to enable reasoning and inference over data by using description logic and OWL semantics. The last one instead enables service publication, discovery, composition, and adaptation thanks to specialized description and ontologies. In the system architecture of Mathew et al. [71], the ontology of the related knowledge base acts also as a service directory for ubiquitous context-aware applications. A knowledge base server is used to register all the Things on the Web and maintain their profiles, hence acting also as a directory of services for applications that need to interact with things on the Web and providing information about things' attributes. The Ambient Space Manager (ASM) component is in charge of acting on or probing ambient physical things and provides a gateway to things on the Web to build ubiquitous applications.

2.5.3 Share layer

Shoaib et al. [91] proposes a system where users can share data produced by their sensors by offering REST APIs which can be queried by any other web application in order to retrieve recorded values. Access to sensors is guaranteed thanks to a SNS open API authentication mechanism. An interesting approach is the one in which smart objects directly communicate with APIs provided by SNS itself; obviously, these objects require capabilities to natively communicate over the Internet and to be programmed according to formats required by

the APIs. This is the case of Baqer [92]'s work, where each sensor has its own dedicated social page where to publish its data. An *Inter-portal communication* is then required to see published data from another social account. In the same direction, there is also the SenseShare application [93] that proposes to use *Facebook* as the main front end by taking advantage of its API. In particular, it uses authentication, privacy, and security settings offered directly by the SNS to share the sensed data. Guinard et al. [94] propose Social Access Controller (SAC), an application that creates a link between web-enabled devices and SNS through RESTful API. The architecture requires a special component that acts as authentication layer and a social access controller, i.e., its main role is to retrieve data from devices' API and to publish it on different Social Networks. A similar approach is also considered by Kamilaris and Pitsillides [95] to share smart home data across social networks. Web-enabled devices communicate directly or by a smart gateway to a central web server that hosts a Web application that sends data to SNSs through their REST APIs. In [96], authors propose a three-layer architecture for enabling social interactions not only between humans, but also between humans and devices or between devices only. The first layer, called *External resource layer*, consists of several types of devices (sensors, actuators, and smart devices provided by different vendors). These devices communicate with the *platform layer*, which is in charge to abstract resources from data and capabilities of devices. Its goal is hence to store all the received data by adding semantic meaning to it. In this way, a Natural language component can easily interpret data and translates it to natural language in order to be published on SNSs. Finally, the third architectural layer (3rd party/user Layer) is about *users* to get access to stored data in the platform via programmable Web APIs. Paraimpu [97] is a platform that lets people connect, use, share and compose physical and virtual things, services, and devices in order to create their custom applications. For authors, a *Thing* is intended not only as a hardware device, but also as already existing *virtual things* on the Web, like SNSs. Users can interact with Web Things through SNS, sending for instance commands to execute on specific actuators. There exist also several approaches to bring Body Sensor Networks (BSNs) into the Web through SNSs. Among others, we cite [98] and SenseFace [99] studies. In the first one authors propose an architecture for integrating BSNs into Social networks through IP multimedia subsystem; only authorized social users can monitor data coming from other members' BSN in real-time. In the second approach, researchers present a 4-tiers architecture in order to bring sensor data to Social Networks. First, body sensor data is collected by gateways in order to reach the Internet. After that, data is manipulated according to the Social Network destination format required. Clearly, in this way, by acting on the third layer, several Social Networks can be considered for this architecture.

2.5.4 Compose Layer

The compose layer includes features for composing services and mashups that rely on them. In this sense, Mainetti et al. [22] investigate the possibility to easily mash-up constrained application protocol (CoAP) resources by proposing a four-layered WoT architecture whose goal is not only to monitor but also to control them, thanks to a bidirectional communication. Such architecture has been designed to lower the entry barrier for developers and foster rapid prototyping, allowing a wider range of developers to build new services on top of smart things. At the same time, authors claim to guarantee high usability of the architecture for a direct and ubiquitous access for users, meaning that smart things data and services should be accessible from everywhere and by means of different kinds of systems and platforms. In this sense, a lightweight access to data is provided, in order to let resource-constrained devices consume and process data that hence can support a low computational load because of their low computational and memory resources. The architecture is composed of four layers: (i) the *accessibility layer*, (ii) the *execution layer*, (iii) the *proxy layer*, and (iv) the composition layer. The first one deals with physical devices that have to expose their resources for the WoT services and applications through a common way that abstracts their heterogeneous hardware features. Embedded devices can act as small servers and can expose their resources as Web resources. There is no need for devices to be aware of the business logic, so in this sense they can preserve their computing and memory resources. The second level is in charge of virtualizing the physical devices and monitoring their connectivity, and of executing the business logic of the running applications. The third layer enables the communication between the user environment and the running applications: together with the previous layer, this part of the architecture is responsible to discover and indexing the available resources and to send the result of the business logic executions to the client applications at the above layer. Finally, at the last layer user can design and easily implement and compose applications that leverage smart things data. Ideally, this part should provide simple APIs for developers that can hence easily collect data from embedded devices and possibly control them by changing their state.

Reference	Description of the main contribution	Mapping Layer
[64]	Web Things can communicate over the Internet in two possible ways: *Direct Integration* or *Indirect Integration*.	*Access Layer*
[86]	Architecture for smart gateways composed of three major layers: *presentation layer*, *control layer*, *device abstraction layer*.	*Access Layer*
[87]	RESTful architecture for the Electronic Product Code Information Service (EPCIS) for bringing RFID tags into the Web.	*Access Layer*
[91]	Distributed WoT services semantic discovery architecture with clustering, indexing, and ranking filtering mechanisms.	*Find Layer*
[88]	Distributed three-layers architecture to enable semantic discovery for WoT environments and that can handle the dynamism of Wot Services thanks to a clustering algorithm and the mobility of gateways through the use of *Wot Gateways*.	*Find Layer*
[89]	DiscoWoT is an extensible semantic discovery service for Web Things, where multiple discovery strategies can be applied to a Web resource representation.	*Find Layer*
[72]	WOTS2E is a search engine for the Semantic Web of Things that uses a web crawler for discovering Linked Data endpoints and through them web-enabled devices.	*Find Layer*
[90]	Framework for the Semantic Web of Things that bases on an evolution of classics Knowledge Base models. The adopted architecture supplies also solutions for information storage, communication and processing.	*Find Layer*
[26]	The Semantic layer is expanded into three sub-layers: the *modeling level* for a common understanding of Things, the *data processing level* to enable inference over data, and the *IoT Services and Application* for handling service publication and discovery.	*Find Layer*
[71]	System architecture where the ontology of the related knowledge base acts also as a service directory for ubiquitous context-aware applications.	*Find Layer*
[91]	System for sharing data produced by users' sensors by offering REST API which can be queried by any other web application in order to retrieve recorded values.	*Share Layer*
[92]	Each sensor is mapped to its dedicated social page where to publish its data. An *Inter-portal communication* allows seeing published data from other social accounts.	*Share Layer*
[93]	SenseShare allows users to share sensor data with their friends through *Facebook*. It also allows owners to apply different filters to the data before sharing it.	*Share Layer*
[94]	Social Access Controller (SAC) as an application that creates a link between web-enabled devices and SNS through RESTful API.	*Share Layer*
[95]	Web-enabled devices communicate directly or by a smart gateway to a central web server that hosts a Web application that sends data to SNSs through their REST APIs in a smart home context.	*Share Layer*
[96]	Three-layer architecture for enabling social interactions not only between humans, but also between humans and devices or between devices only.	*Share Layer*
[97]	Platform that allows people to connect, use, share and compose physical and virtual things, services, and devices in order to create their custom applications.	*Share Layer*
[98]	Architecture for integrating BSNs into Social networks through IP multimedia subsystem.	*Share Layer*
[99]	SenseFace: 4-tiers architecture in order to bring body sensor data to Social Networks.	*Share Layer*
[22]	Four-layered WoT architecture for monitoring, controlling and mash-up constrained application protocol (CoAP) resources.	*Compose Layer*

Table 2.1 Summary of the studies presented in Section 2.5

2.5.5 W3C WoT

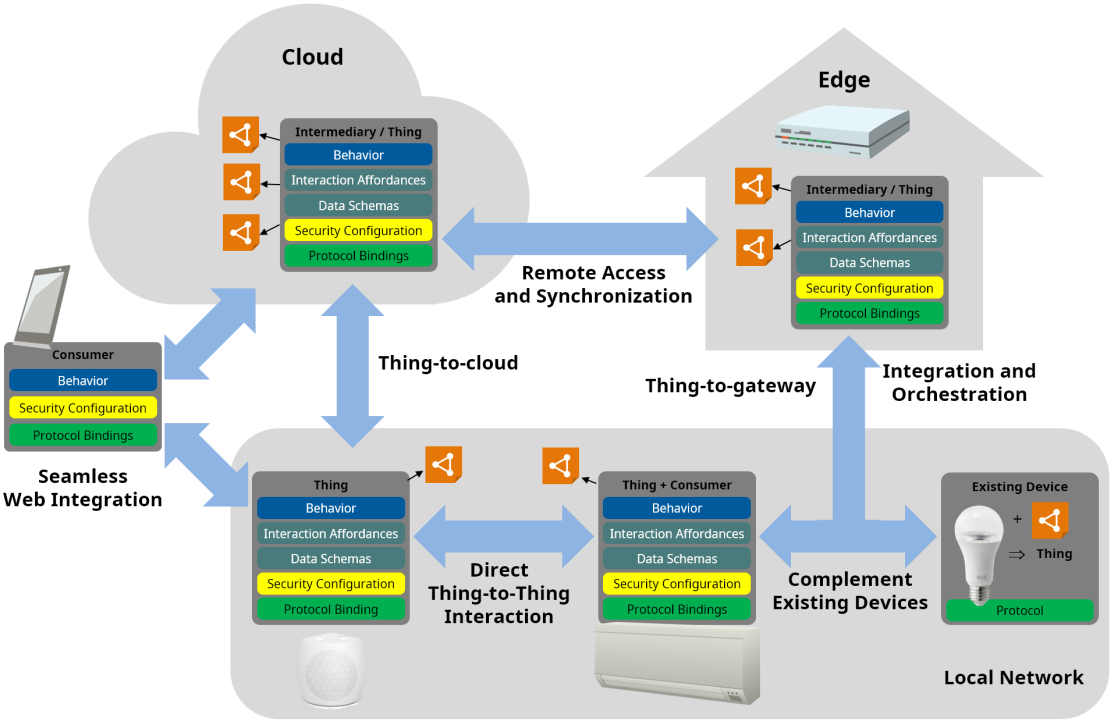


Fig. 2.5 W3C Web of Things Architecture [2]

The goal of W3C WoT standard is to ease the deployment of IoT scenarios by solving the problem of the interoperability, regardless of the scenario. For this reason, the standard is thought to be able to cover almost all the interesting use cases for IoT and WoT, like Industry 4.0, domotics, smart agriculture, just to name a few. Figure 2.5 shows the architecture proposed by the W3C WoT working group and all the possible entities and interactions covered by the standard. The main interaction patterns are the following:

- *Device Controllers*: this pattern involves a local device that is controlled by a user through a remote controller, possibly in a local network. The remote controller can be implemented as a browser or native application and mainly acts as a client, in order to send message requests to the controlled device, like for instance to read a sensor value or to execute a command on the device. This one hence must act as server in order to receive the requests and reply accordingly. Clearly, if events are enabled on device, it has to assume a client role in order to emit the notification to the controller, that in this situation acts as a server. This is the case, for example, of a smart air conditioner

directly controlled by a smartphone application, that can send a confirmation message when the room reaches the desired temperature.

- *Thing-to-Thing*: this scenario involves two devices that directly communicate with each other. In this case, both devices have a server role, but at least one must assume also the client mode to issue requests to the other in order to actuate/reply accordingly. As an example, one device can be a temperature sensor that, once a threshold has been overcome, turns on the other device for increasing the room temperature.

- *Remote Access*: this pattern regards the case in which a mobile remote controller, as for instance a smartphone, can switch between different network connections and protocols, like between a cellular network and a home network. As previously described, when the controller is at home it is considered in a trusted environment and there is no need for additional security mechanisms. On the contrary, when the controller is outside the local network, an access control is required to ensure trusted information exchange between the devices.

- *Smart Home Gateways*: the previous case can be easily managed by a smart home gateway, located between a local network and the Internet. Its main role is to enable a trusted and safe communication between devices in the local network and external controllers that send commands over the Internet. The home gateway has both the role of client and server, since it has to listen for requests coming from Internet and forwards them accordingly to the right devices.

- *Edge Devices*: an edge device can be considered as an augmented home gateway. Like this last one, it mainly has to bridge between public and trusted networks, but it also owns local computational capabilities to manipulate data and can also bridge between different protocols. Edge devices are often used in industrial scenarios where they provide pre-processing, filtering and aggregation of data.

- *Cloud digital twin*: the W3C standard identifies a *digital twin* as a virtual representation, i.e., a model of a device or a group of devices that resides on a cloud server or edge device. In the cloud scenario, a digital twin is meant to *mirror* the gateway with all the connected appliances, managing them and so co-working in conjunction with the gateway. The cloud can receive requests and messages for the Things from remote controllers, which hence can be located anywhere.

We remark that devices often need to act both as client and server at the same time. This mixed use of roles justifies the meaning of a software component called *Servient* that, as

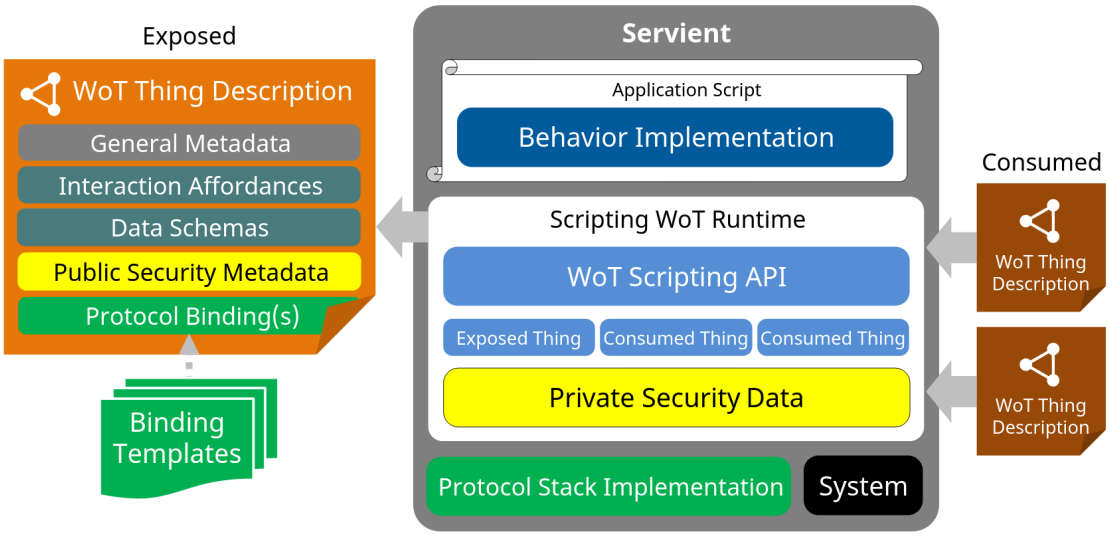


Fig. 2.6 Implementation of a Servient using the WoT Scripting API [2]

the word itself suggests, can behave as server and a client simultaneously. The Servient is the software stack implementation of the WoT Thing building blocks shown in figure 2.3. It provides means for parsing and producing TDs, and typically supports multiple Protocol Bindings to enable interactions with different IoT platforms. On one hand, a thing is implemented by a Servient, that exposes a representation of the Thing called *Exposed Thing* and makes available to consumers the Thing network-facing interfaces. The *exposed thing* can also be used by other software layers of the Servient, like for instance the *application* one for implementing the Thing behaviour. On the other hand, *Consumers* must be always implemented by a Servient, since they need to parse and process the TD and must enable the right protocol stack according to the Thing capabilities. Once the Thing has been handled by the consumer, the Servient provides the so-called *Consumed Thing* software object, making it available to those applications running on the Servient (like for instance a *Mashup application)* that want to interact with the Things. An *Intermediary* is a component implemented by a Servient that performs both the role of a Consumer to the Thing and a Client to the Consumers, being located in the middle between a Thing and its Consumers. Hence, in this case the Servient contains both the *Consumed Thing* and *Exposed Thing* instances of the Thing. This layer separation is well depicted in Figure 2.6, where there are represented all the blocks that compose the software stack for a Servient. The behaviour defines the application logic of a Thing, and includes the *autonomous behaviour* of a Thing (the internal functioning of sensors and actuators), the *handlers* of the Thing affordances (which operation to perform when an affordance is activated), the *consumer behaviour* (controlling Things or running

mashups), and the *intermediary behaviour* (proxying or composing Thing aggregation). Clearly, depending on the behaviour implementation, Servient hosts Things, Consumers, and Intermediaries accordingly. The *WoT Runtime* represents the implementation of the interaction model and it is the execution environment for the behaviour, hence it is able to fetch, parse, serialize, and serve TDs. The optional *Scripting API* component defines the application-facing interface that follows the Thing abstraction, enabling the Thing behaviour to run at run-time through the application scripts. The WoT Runtime is in charge of instantiating the software representation of the Thing: an *exposed Thing* represents a Thing hosted locally and accessible through the protocol stack implementation of the Servient; a *consumed Thing* represents a remotely hosted Thing that needs to be accessed thanks to a communication protocol. It can be considered as a proxy/stub for a Thing. In practice, both *Consumed Things* and *Exposed Things* are software objects that can be manipulated (created, destroyed, queried, and so on) by the application scripts. Nevertheless, some operations may be restricted depending on which security mechanisms are in place. The *Private Security Data* is managed by the WoT Runtime but intentionally kept apart from the application; instead it is used by protocol bindings in order to authorize and protect the integrity and confidentiality of interactions. The *Protocol Stack implementation* implements the network-facing interfaces of an Exposed Thing for letting *Consumers* access the WoT Interfaces of a remote Thing via its Consumed Thing. More in detail, the protocol stack produces the right messages to communicate over the network. Clearly, several protocols can be supported at the same time. The *System APIs* aim at providing local hardware or systems services to behaviour implementations through the Thing abstraction, as if they could be accessible over a communication protocol. In particular, the WoT Runtime enables the behaviour implementation to instantiate a Consumed Thing that internally communicates with the system instead of the protocol stack. This can be the case of Things connected via proprietary protocols or protocols that are not natively WoT-enabled.

Part II

Contributions

Chapter 3

Tools

3.1 WoT Store

3.1.1 Overview

The success of the W3C WoT initiative strongly depends on its wide acceptance from the academic and industrial communities, as well as from the end-users. At present, the existing W3C WoT implementation frameworks (e.g. [25] [100]) provide several low-level functionalities for the Thing modeling and creation; however, their usage requires a solid knowledge of the W3C WoT standards and coding skill, hence they are not easily accessible from the non-technical personnel. The literature on W3C WoT is quite scarce, and mainly limited to proof-of-concept applications [101][102][103][104]. Hence, there is a concrete need of service tools (the so-called Software Ecosystem (SECO) [105]) that can facilitate the adoption of the W3C WoT technology on existing and novel IoT scenarios. In addition, IoT/WoT deployments are often characterized by dynamism, e.g. the need of adding/removing new devices, of re-defining the devices' behaviour (e.g. software updates), of tuning system parameters, just to cite few examples. A straightforward research challenge is how to support the IoT deployment reconfiguration seamlessly, i.e. avoiding the manual intervention on the field. In this Section, we address both the research questions (*RQ*s) mentioned so far, i.e.:

- (*RQ1*) how to ease the discovery and the management of WoT resources (e.g. Things), in both private and public environments?
- (*RQ2*) how to support the dynamic reconfiguration of the WoT scenario, e.g., the deployment of new WoT resources or the interconnection among the existing ones,

while minimizing the need of manual configuration (for system administrators) and coding (for programmers)?

The solution to the *RQ*s above is constituted by WoT Store, a novel platform for managing and deploying resources on the W3C WoT. The WoT Store is not an implementation of the W3C WoT (like [25]), rather a service platform that can be installed on top of it, adding novel functionalities for the end-users. Indeed, the WoT Store allows the dynamic discovery and managing of the active Things available on a public or private deployment; through the Web dashboard, the users can search and list the Things in the scenario, monitor their properties and events, and execute their commands, without any change to the IoT layer. More generally, the WoT management means general-purpose functionalities like for instance: find the Things satisfying specific requirements (e.g. location), perform actions on them or display property values, that we expect to be present in any W3C WoT deployment, regardless of the use case. In addition, thanks again to the fact that the Thing interfaces are well defined and non-ambiguous, the WoT Store allows the management of applications that can make use of the available resources: in this sense, the platform recalls the operations of popular software repositories used for mobile applications. In this Section the WoT Store platform is presented, both in terms of components and evaluation results. More in detail:

- We present the WoT Store main functionalities, architecture, implementation, and use cases. The framework is micro-services oriented, with three main modules available, i.e: the *Things Manager*, the *Application Manager* and the *Data Manager*. We illustrate the operative flow from the Thing discovery and management to the installation and execution of the applications until the aggregation and visualization of the data produced.

- We validate the components of the WoT Store in a real-world testbed composed of three Wireless Sensor Networks (WSNs) mapped on different wireless access technologies (Wi-Fi, BLE and Zigbee). Specifically, the analysis provides evidence of the dynamic discovery of Things/sensors and highlights the possibility to deploy WoT applications orchestrating the sensing operations, regardless of the M2M technology used by each sensor.

- We test the scalability of the WoT Store architecture on a mixed real/simulated large-scale IoT application, i.e. a pedestrian crowdsensing system; each Thing is associated to a simulated mobile smartphone, providing the sensing values and the positions over time on an urban scenario (the downtown area of Bologna). We demonstrate the capability of the Data Manager to aggregate and visualize the data streams originated by the WoT applications in two formats (time-series and geographic data).

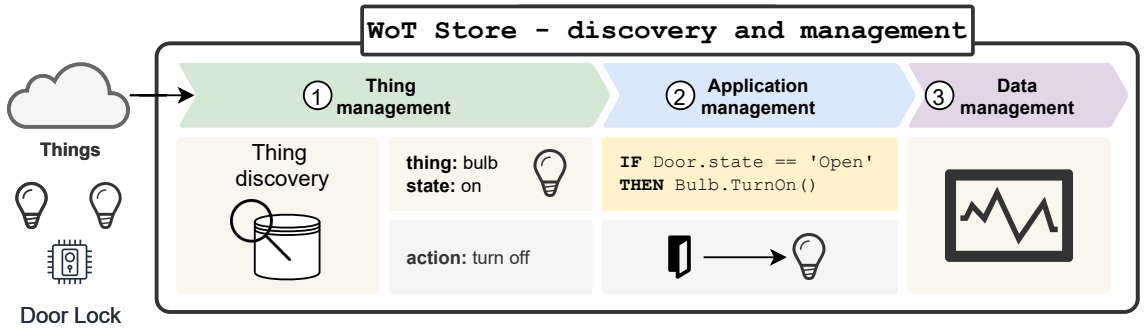


Fig. 3.1 WoT Store functionalities and sequence of operations: Things discovery and deploy of a Mashup application.

The WoT Store framework is designed to be micro-services oriented, with the possibility to easily load/unload new modules based on the specific user requirements. WoT Store services can be grouped into three main areas, i.e.:

1. *Things Management*: the WoT Store allows discovering the available Things in the environment, and/or to select a subset of them according to user-defined, semantic criteria (e.g. the location). Through the GUI, the user can interact with each of them according to its TD, i.e. display properties, execute actions or observe the notifications of events.

2. *Application Management*: beyond monitoring the existing resources, the WoT Store allows to perform changes to the actual WoT scenario, by instantiating new Things or executing applications involving the interaction among the available Things. Here, the WoT Store acts as an application repository: via semantic queries, the users can search for software matching specific criteria (e.g. the compatibility with the actual devices). In addition, the application can be executed on any of the Servient registered to the WoT Store, again minimizing the manual configuration efforts. A distinction between Thing Applications (TAs) and Mashup Applications (MAs) was provided previously on in section 2.4.

3. *Data Management*: the WoT Store allows to process and aggregate the data produced by a WoT application, by providing proper facilities in order to gather the *data streams*, aggregate them and visualize the results on a Web dashboard.

Figure 3.1 and figure 3.2 show how the services can be used in pipeline in two typical use cases, respectively the *discovery and management* and the *reconfiguration of the scenario*. Let us consider a generic IoT environment with heterogeneous devices in terms of communication

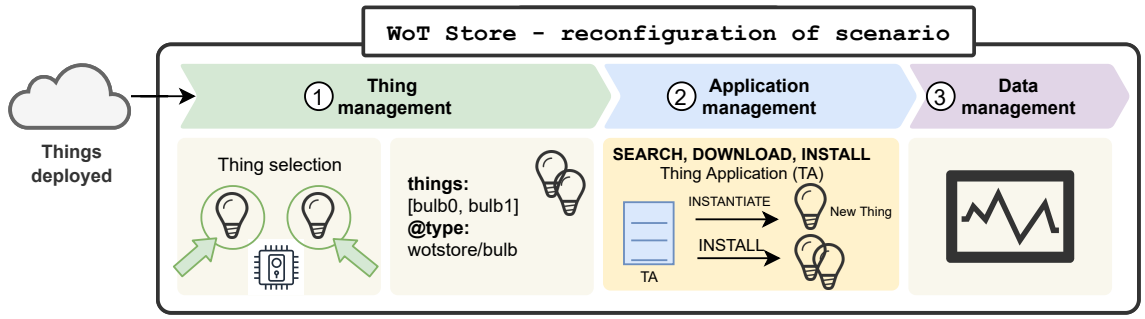


Fig. 3.2 WoT Store functionalities and sequence of operations: reconfiguration of the scenario through the instantiation of new Things and the update of the Thing Applications.

protocols, data formats and implementations. No assumption is made regarding them; they could be native W3C WoT compliant devices or they could have been mapped into the W3C WoT ecosystem by means of any of the architectural patterns shown in Figure 2.5. In any case, we assume that the corresponding Web Things have been deployed on some Servient. We recall that a *W3C Web Thing (WT)*, also referred to simply as *Thing*, is the representation of an *IoT Thing* through the W3C WoT Standard, hence describing it with a Thing Description and possibly providing its Thing Behaviour, as explained in Section 2.4. Furthermore, the instantiation of such Web Thing is made through the deployment of a software stack called *Servient*, that is in charge of turning the description of a Web Thing into a software object that is capable of communicating and interacting with the rest of the components in the scenario. In the first use case, through the Discovery Service, the Things are registered to the WoT Store tool; they are now searchable and displayable from the Web dashboard. For instance, let us assume the presence of a Thing connected to a Smart Bulb device; as soon as the Thing is connected to the platform, the user can perform the action `turnOn` or `turnOff` directly from the GUI. As next step, the user might be interested in downloading applications from the Store through which it might implement coordinated or autonomic behaviours involving multiple Things at the same time. For instance, assuming the Web Things associated to a Smart Bulb and a Smart Lock are both available and active, the user might run an application that issues the action `turnOn` each time the Lock generates an `open` event, in an automatic way. Behind the actuation, some applications might also produce streams of data that can be relevant for the context. In the previous example, the user might monitor the sequence of decisions performed by the application (i.e. the `turnOn` or `turnOff` actions) over a temporal window. This is possible by connecting the application to the proper data facilities of the WoT Store, hence closing the pipeline.

In the second use case, let us assume the Things have been already registered and deployed in the WoT Store tool. From the dashboard it is possible to perform a *search* operation, for

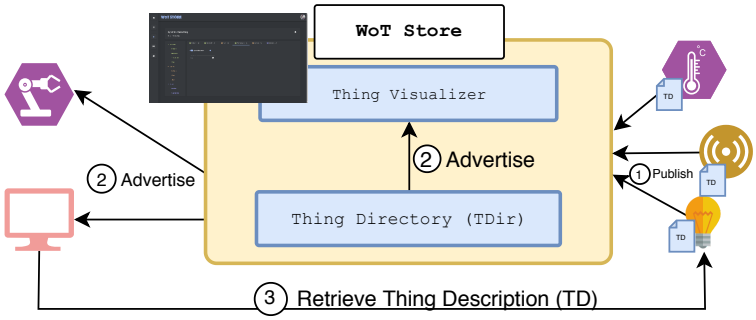


Fig. 3.3 The operations of the Things Discovery Service (TDS).

instance based on the *@type* of the Thing, for selecting a subset of them. The user can now update their behaviour: first he searches and downloads a new *Thing Application* according to his needs, then he issues an *update* operation on the selected Things. The user can also decide to instantiate a new *Thing* with the same TA and using one of the already registered Servients available for this purpose, hence creating a new Web resource. Finally, as in the previous use case, *Things* are now ready to display data from the *Data Management* component or to be managed through the Thing Management component. In section 3.1.2 we provide an in-depth description of the three main modules of the WoT Store. The implementation details and the technologies used are sketched in Section 3.1.3.

3.1.2 Service Components

We detail here the main modules of the WoT Store i.e. the *Thing Manager*, the *Application Manager* and the *Data Manager*.

Things Manager

The Things Manager module allows the users to interact with the Things already available in the WoT environment. We further distinguish between two sub-modules. i.e.: the Thing Discovery Service (TDS), that is in charge of registering the active Things on our tool, and the Thing Visualizer Service (TVS), that is in charge of displaying the registered Things on the GUI of the WoT Store.

Things Discovery Service (TDS) The overall discovery procedure is depicted in Figure 3.3. The procedure is initiated by the Web Things when they register themselves to the WoT Store and more specifically to the Thing DIrectory (TDI) module, assuming that the URI of this latter is fixed and known. In the registration phase, each Thing provides its Thing Description (TD). The TDI works as a broker and as a repository of TDs; in the first case, it

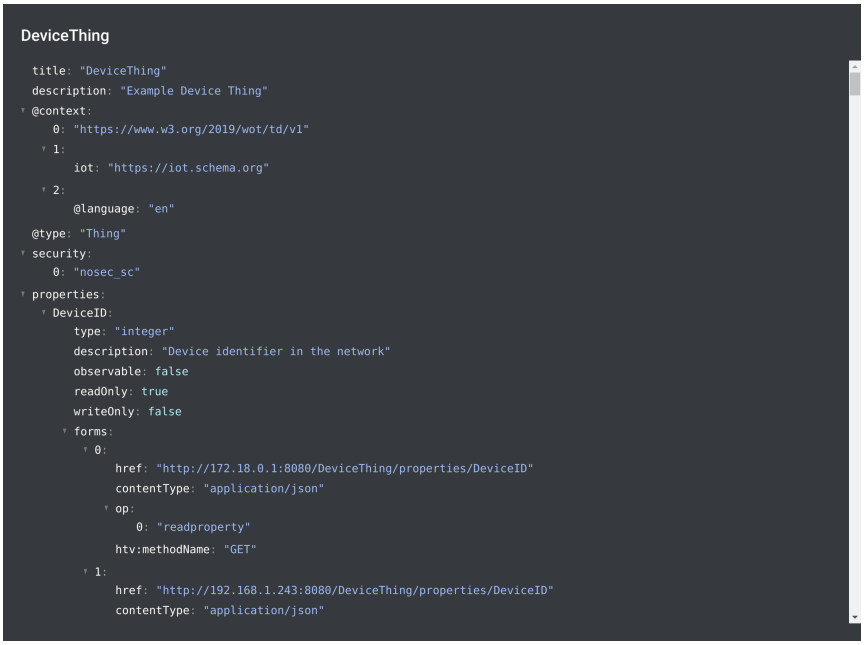


Fig. 3.4 A portion of the TD of the Device Thing of Table 3.3. The Thing is associated to a wireless sensor producing temperature values.

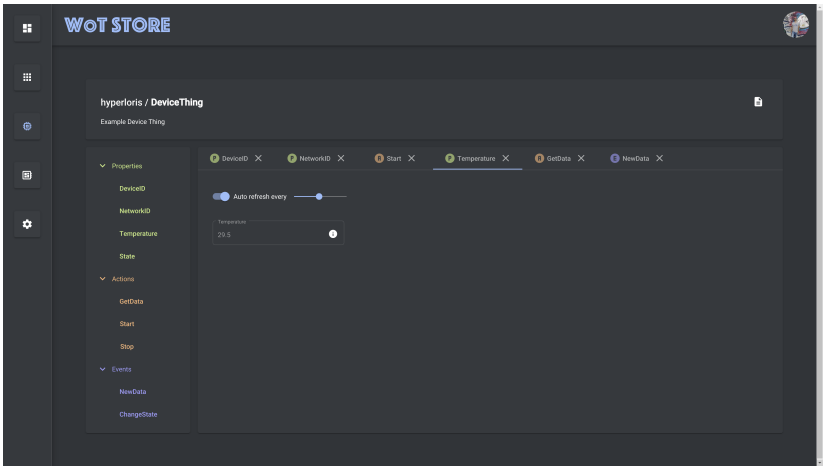


Fig. 3.5 The rendering of the actions of the TD of Figure 3.4 within the WoT Store.

notifies the presence of a new Thing to all the clients, including the TVS described below. Each client can then retrieve the TDs directly from the Things, in order to consume the most updated version. In addition, the TDI stores a copy of the TD of all the registered Things; this is required since the TDI can support search and filter operations, issued by the user through the Web dashboard. Two usage modes of the TDI are considered, according to its visibility level: i.e. public TDI or private TDI. In the first case, all the Things registered to the TDI are searchable from the clients: this might be the case for instance of a smart city willing to share its IoT resources with all its citizens. Vice versa, in the private case, the access to the Things is restricted, and proper authorization mechanisms are employed by the WoT Store: this is the case of smart home or industrial IoT deployments with severe security concerns. The visibility flag must be set during the TDI configuration process, together with other meta-data such as the authentication mechanisms (e.g. header-based authorization, token-based authorization like OAuth 2.0[1]) required by clients to access the TDI.

Things Visualizer Service The TVS is a Web dashboard and the main GUI of the WoT Store. It allows to visualize the list of available Things registered to the TDI (by subscribing to it). Moreover, it supports search operations, where a subset of Things is selected according to user-defined conditions; search operations are enabled by a Web form with a list of predefined fields that can be filled through the GUI, and involve a subset of the meta-data contained in the TDs. Finally, the TVS allows the user to interact with each Thing available in the TDI or contained in the result of a search operation; this is performed by parsing the corresponding TD and creating an ad-hoc Web GUI, through which it is possible to monitor the state properties, click and execute actions (passing the needed parameters if requested), or receive notifications of the events occurred. Figure 3.4 shows a small portion of the TD of a Device Thing measuring temperature values and used in the Pervasive Sensing testbed of Section 3.1.4; the full interaction model is reported in Table 3.3. The corresponding GUI rendered within the WoT Store with the list of available actions and properties is depicted in Figure 3.5.

Applications Manager

The Application Manager supports the dynamic search, download and execution of third-party WoT applications, involving the interactions with the available resources, or the creation/update of new resources. The current applications can be assumed to be coded in Javascript (JS), since this is the language of the WoT implementation made available by the

[1] https://oauth.net/2/

WoT W3C community [25], although this choice does not impact the general functionalities of the WoT Store.

Conceptually we distinguish between two classes of WoT applications supported by the WoT Store, i.e.: *Things Applications (TAs)*, and *Mash-up Applications (MAs)*. The TAs correspond to the source code of a Thing, hence to a static object that can be activated when executing it. Through the TAs, it is possible to instantiate a new Thing in the WoT Store, or to update the behaviour of current Things, as better described in the following. Vice versa, the MAs implement automatic policies that involve the interactions of multiple Things (active and registered on the TDI); the result of a MA can be an actuation or a data stream that can be processed through the Data Manager described in Section 3.1.2. More in detail, the Application Manager provides three main functionalities:

1. *App Storing*. The source code of the WoT applications is stored in a database. Moreover, each application (MA or TA) is associated to a semantic description, including a list of meta-data, like its category, description, and the resources required (e.g. the type of Things used). For instance, Table 3.1 contains the RDF description of a MA that queries all Things of type "Temperature" registered to the TDI and computes the average of sensed data. In the current implementation, we describe each WoT application through a list of RDF fields; clearly, more formal descriptions of the MA and TA can be considered, by means of dedicated WoT ontologies.

2. *App Searching*. Through the Web dashboard and the compilation of specific fields, the user can build SPARQL queries[2] in order to filter the WoT applications matching specific criteria, defined again through the meta-data. The results are then displayed on the WoT Store GUI.

3. *App Executing*. After having selected the application meeting his/her requirement, user can download and execute it. In this case, the proper run-time environment (i.e. the Servient where to deploy the application) must be selected among the ones registered to the WoT Store. Additional features for the execution of the TAs can occur in *Normal* or *Update* mode. The first case is equivalent of creating a new Thing, and registering it to the TDI. The second case (*Update*), instead, gives the possibility to replace a list of active Things with new ones implementing the behaviour described by the TA downloaded by the WoT Store. Hence, a SPARQL search query is issued on the TDI in order to select the Things to unregister; then, a new set of Things is deployed with the updated source code provided by the TA.

[2]The SPARQL code in all the search operations must not be inserted manually, rather, it is generated automatically based on the search option fields filled by the user on the Web GUI.

subject	predicate	object
<WoTStore://smartAtmosphere>	*schema:applicationCategory*	Domotics
<WoTStore://temperatureMonitor>	*schema:downloadUrl*	coap://wotstore.cs.unibo.it:8081/market/actions/getApplication?application=smartAtmosphere
<WoTStore://temperatureMonitor>	*schema:downloadUrl*	http://wotstore.cs.unibo.it:8080/market/actions/getApplication?application=smartAtmosphere
<WoTStore://temperatureMonitor>	*wotstore:involve*	sosa:Actuator
<WoTStore://temperatureMonitor>	*rdf:type*	schema:SoftwareApplication
<WoTStore://temperatureMonitor>	*dcterm:description*	smartAtmosphere is an application that sets a RGB color to each smart bulb, in order to reproduce different kinds of atmosphere
<WoTStore://temperatureMonitor>	*rdfs:label*	smartAtmosphere

Table 3.1 Example of RDF Description of a MA application available in the WOT STORE.

Data Manager

This module contains functionalities for the processing and visualization of the data produced by the running WoT applications. The block components of the Data Manager are the *data streams*; each data stream can be configured in order to be attached to a MA, and to gather data from it through a set of APIs made available by the WoT Store. Each data stream consists of two sub-components: a *data aggregator*, that filters/aggregates the output of the MA, and a *data plotter*, that creates the proper Web dashboard of the processed data. Clearly, the stages above are strictly dependent on the data format, on the MA in use and on the user needs; it is nearly impossible to cover all possible requirements. For this reason, at the moment we provide basic data stream templates that must be extended/customized by users/developers. Moreover, as proof of concepts, we implemented two specific data-flows: one for temporal data-series (composed by a time-stamp and a numeric field), the other one supporting geo-data (in GEOJSON) and generating the corresponding heatmap. Further details regarding the two data streams are provided in Section 3.1.4.

3.1.3 Implementation

The WoT Store is composed of four internal components, reported in Figure 3.6: the Market Service (MS) and the Thing DIrectory (TDI) on the server side, the Market Interface (MI) on the client side, and the Runner (RNN) on the physical device hosting the W3C WoT Servient. The WoT Store implementation involved the usage of several software libraries: we briefly discuss here the main solutions adopted, while Table 3.2 provides a mapping of the service components of Section 3.1.2 with the enabling technologies.
The Market Service (MS) has been implemented as a `Node.js`[3] v10.x application using the `LoopBack`[4] v3 framework and the Socket.IO[5] library. The MS exposes the REST APIs for all the Things-related and application-related operations and a WebSocket endpoint for

[3] https://nodejs.org
[4] https://loopback.io
[5] https://socket.io

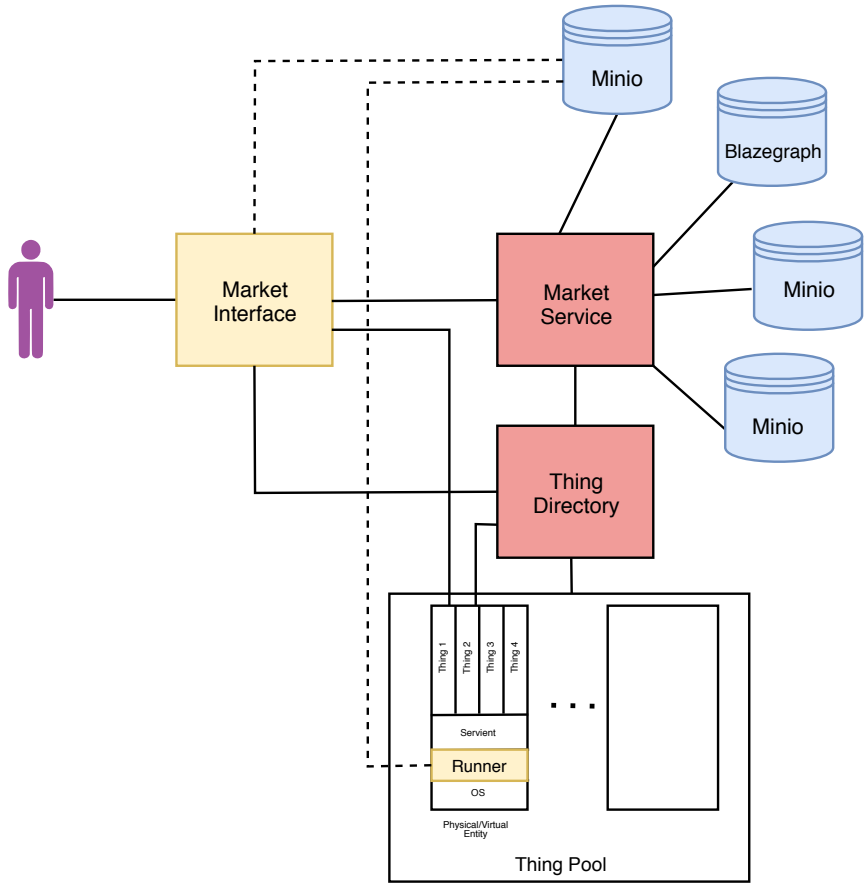


Fig. 3.6 The WoT Store internals.

real-time notifications. In addition, it stores the platform information through four database technologies: (*i*) `Minio`, an object storage server containing the WoT applications (MAs and TAs) source code, (*ii*) `MongoDB`, the popular NOSQL database used to store the user data, (*iii*) `Blazegraph`, the triplestore used to save the application metadata and the TDs and (*iv*) `Redis`, a high-performance in-memory database, used for the real-time processing of the Things notifications. A complete LoopBack connector for `Blazegraph` implementing all the necessary methods to initialize the connection has been developed, the data migrations and CRUD operations. In addition, a second component is in charge of converting the JSON-LD to N-Quads when pushing the data to the triplestore, and from JSON to JSON-LD when they are pulled out.
The Runner (RNN) is a piece of software developed to facilitate the installation of the WoT Store, and to automatize the execution of the WoT applications on the devices. The RNN is written in JavaScript and exploits the `ShellJS` library. When installed on a machine (which could be a physical device, like a Raspberry Pi, or a Virtual Machine), the RNN registers the machine to the MS. Then, through the RNN, the user can install the WoT Servient on its device, by choosing the version compliant with the current hardware and software (operating system) configuration.[6] The RNN allows issuing commands from the WoT Store directly on the device, like for instance the execution of a MA or a TA selected from the repository. To this aim, it supports multiple run-time environments through the executors, i.e. the `Shell` and `Docker`[7] in the current implementation, while the support for `Kubernetes`[8] can be considered as future work. Finally, the Market Interface (MI) is an `Angular`[9] v6 web application composed of several modules. Among these, there is the Thing Visualizer Module, which implements the TVS introduced in the previous Section, i.e. it renders a Thing starting from its TD. Properties and events are updated in real-time thanks to libraries such as `ngx-mqtt` and `rxjs-websockets`; for each action, a specific form is created with the necessary constraints for the data input.

3.1.4 Components Validation

The components of the WoT Store have been validated through two evaluation studies: (*i*) a small case testbed of a pervasive sensing scenario (Section 3.1.4), and (*ii*) a large-scale simulation combining real Things and virtual devices in an urban crowdsensing scenario (Section 3.1.4). The studies addressed different goals and evaluated different components of

[6] At present, we rely on the JS Servient made available in [**?**]; however, we imagine the case where multiple Servient implementation will be available for a specific device.
[7] https://docker.com
[8] https://kubernetes.io
[9] https://angular.io

Area	Service	Technologies / Libraries
Thing Manager	TDS	`Node.js`, `Blazegraph`, `SPARQL`.
Thing Manager	TVS	`Angular`, `ngx-mqtt`, `rxjs-websockets`, `socket.io-client`.
Application Manager	App Storing	`Minio`, `MongoDB`, `Blazegraph`.
Application Manager	App Searching	`SPARQL`.
Application Manager	App Executing	`Docker`, `shelljs`.
Data Manager	Data Aggregator	`bull`.
Data Manager	Data Plotter	`ngx-echarts`.

Table 3.2 List of technologies used for the implementation of the WoT Store service components of Section 3.1.2.

the WoT Store framework. In the testbed, we focus on the Thing Manager and Application Manager modules; more specifically, we demonstrate the possibility to orchestrate the sensing operations of multiple, heterogeneous wireless sensors through the MAs, and we provide evidence of the Thing Discovery Service. The crowdsensing study aims to verify the scalability of the WoT architecture and of the WoT Store under an increasing number of Things to manage; moreover, it demonstrates the capabilities of the Data Manager to aggregate and visualize both time-series and geographic data streams produced by MAs orchestrating the sensing operations of simulated mobile devices.

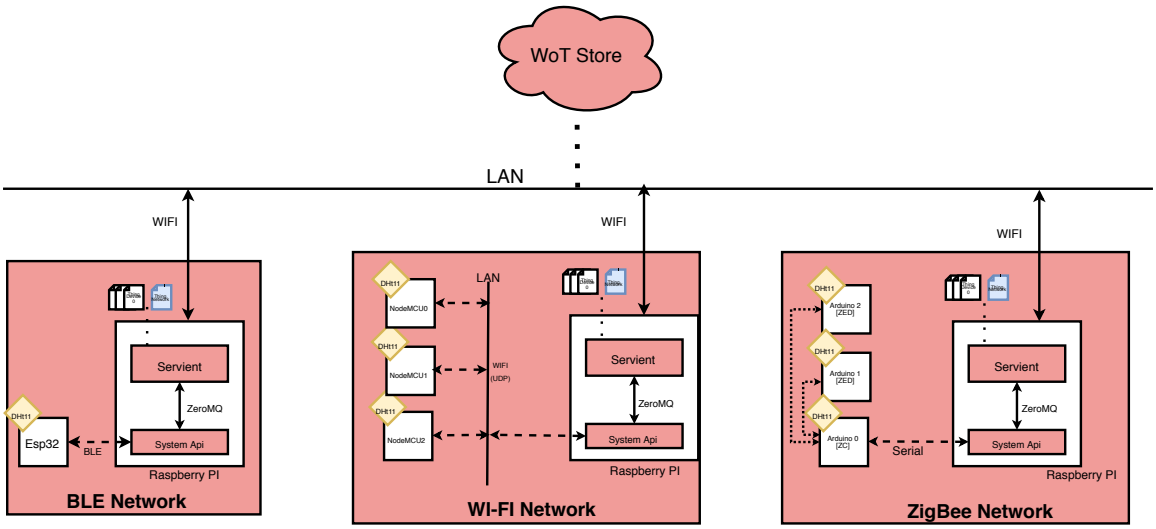


Fig. 3.7 The IoT/WoT monitoring system deployed in this study.

Pervasive Sensing Testbed

The testbed represented in Figure 3.7 consists of an indoor monitoring system composed of three layers: sensing, fog, and processing. The sensing layer is constituted by three Wireless

Sensor Networks (WSNs), operating at different rooms of the same building: an IEEE 802.15.4 WSN network, an IEEE 802.11 Wi-Fi WSN network and a Bluetooth Low Energy (BLE) WSN. The 802.15.4 network includes four devices (*Arduino Xbee* boards), with one Coordinator and three Leaf nodes equipped with sensing units (*ThinkerKit* temperature sensor). The Wi-Fi network includes three devices (two *NodeMCU* and one *Arduino WiFly* board), all provided with a direct link toward the Access Point (AP) and with a *DHT11* temperature/humidity sensor. The BLE WSN consists of one *ESP32* board, provided with a *DHT11* sensor. The 802.15.4 coordinator, the BLE and the Wi-Fi devices are connected to a *Fog* node, via USB cable links (for the 802.15.4 Coordinator) or wireless links (for the BLE and the IEEE 802.11 devices). Finally, the processing layer is constituted by a Linux server, connected to the Fog nodes via Wi-Fi links. The W3C WoT architecture and the WoT Store have been deployed as follows:

- Edge devices, i.e. the wireless sensors, implement low-level communication and sensing operations in the embedded firmware (written in C language). The implementation as well as the list of operations and the data format used by each device are technology-dependent. This layer is part of the IoT, while it is not covered by the WoT architecture.

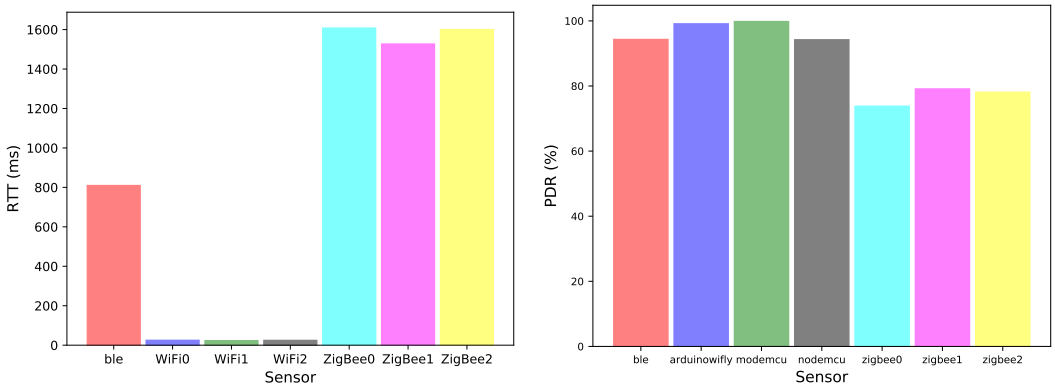
- Fog nodes run a W3C WoT Servient, by using the JavaScript (JS) framework available at [25]. Each Fog node exposes two types of Web Things, i.e.: multiple (*i*) *Thing Devices*, describing the properties, events and actions of physically managed edge devices, and one (*ii*) *Thing Network*, describing the overall performance of the virtual WSN composed by the list of connected Thing Devices. Table 3.3 displays some of the properties, actions, and events described in the Thing Description (TD) for a Device Thing.

- Finally, the Processing node hosts the WoT Store. This latter allows to manage the Web Thing Devices and Web Thing Networks through the Thing Manager presented in Section 3.1.2. Also, we implemented multiple MAs that are in charge of orchestrating the sensing operations, i.e. of selecting a subset of devices to query at each sensing slot, according to MA-specific policies. At each interval, the MA works by querying the TDI and gathering the list of Things Devices available on the WoT Store; hence the MAs are also able to adapt to dynamic conditions where the number of available Things is varying over time, as demonstrated by the analysis below.

We highlight that the WSNs are heterogeneous in terms of M2M technology and network performance. To this purpose, Figure 3.8(a) depicts the average per-sensor Round Trip Time

Name	Type	Description
DeviceID	Property	Device identifier in the network.
NetworkID	Property	Network identifier the device belongs to.
Temperature	Property	Last temperature value.
State	Property	Current state of the device.
GetData	Action	Get the temperature data.
Start	Action	Start sending data at each time slot.
Stop	Action	Stop sending data.
NewData	Event	This event is fired when new sensor data is produced.
ChangeState	Event	This event is fired when the connection state changes.

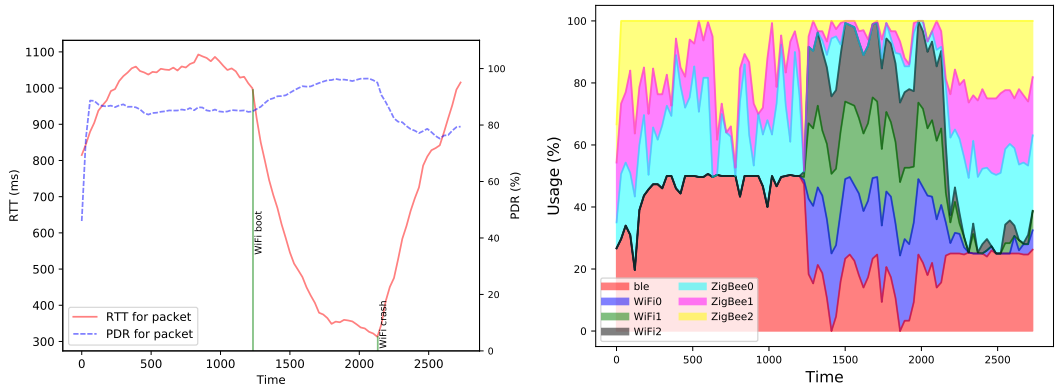
Table 3.3 List of Properties, Actions, and Events of a Device Thing.



(a) (b)

Fig. 3.8 The per-sensor RTT and PDR metrics are shown respectively in Figures 3.8(a) and 3.8(b).

(RTT), computed as the delay to issue the `getData` command from the MA and to receive the sensor data. As expected the Wi-Fi devices experience the lowest RTT values due to the higher channel bandwidth provided by the M2M technology. Figure 3.8(b) shows the per-sensor Packet Delivery Ratio (PDR), defined as the ratio of successful `getData` command issued by the MA. As expected, the Wi-Fi sensors are also the most reliable nodes. Based on these results, we implemented three MAs on the WoT Store, simply denoted as P_1, P_2, P_3. Each MA selects M different Things Device to query at each sensing slot, but according to different policies, i.e.: (*i*) the MA P_1 (RTT-aware) selects the M Things with the lowest mean RTT values; (*ii*) P_2 (PDR-aware) selects the M Things with the highest mean PDR values; (*iv*) P_3 (PDR-RTT aware) selects the M Things providing the best RTT-PDR trade-off. We assume that -at system startup- the MAs have no knowledge about the WSN performance (i.e. the results shown in Figures 3.8(a) and 3.8(b)), and hence they have to discover the optimal



(a) (b)

Fig. 3.9 The impact of the Thing Discovery Service on the PDR and RTT performance indexes in a scenario with a varying number of available Things/devices is shown in Figure 3.9(a). The Thing/device utilization over time is depicted in Figure 3.9(b).

set of M Things maximizing the specific policy in use. This is implemented through basic online reinforcement learning mechanisms, provided by the Q-learning algorithm [106]: the MA computes a numeric reward each time the `getData` command is issued toward a sensor, related to the specific policy (e.g. the packet RTT in case of MA P_1). More details of how the Q-learning algorithm has been applied in the testbed, since they can be found in 4.1, while here we focus on the operations of the Things and Applications Managers.

To this purpose, Figures 3.9(a) and 3.9(b) provide a validation of the Thing Discovery Service (TDS). We considered the following experiment: at system startup, only the BLE and Zigbee Things are registered to the WoT Store. Hence, the MA relies exclusively on them for sensing operations. At t=1200 seconds, the Wi-Fi Things are activated; they autonomously publish their TDs and hence become discoverable by the MA via the TDI. We highlight that the process above occurs in an autonomic way without any manual configuration. At t =2100 seconds, the Wi-Fi devices are physically detached from the environment, without notifying the WoT Store. Figure 3.9(a) shows the average PDR and RTT metrics over time as computed by the running MA (in this case, we used P_3). It is easy to notice the impact of the TDS since both the metrics improve from t >1200 seconds, as a direct consequence of the fact that the Wi-Fi devices are used by the MA; we recall that -in accordance with the results of Figures 3.8(a) and 3.8(b)- the Wi-Fi technology maximizes both the RTT and PDR performance. At the same time, it easy to notice that the RTT decrease and the PDR increase occur gradually and not instantaneously; this is due to the Q-learning convergence delay, since the usage of Wi-Fi is reinforced at each packet transmission, hence increasing the selection probability over time. Finally, both the performance indexes become worse

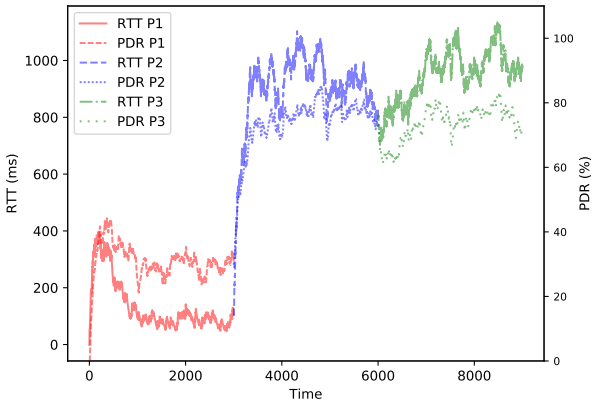


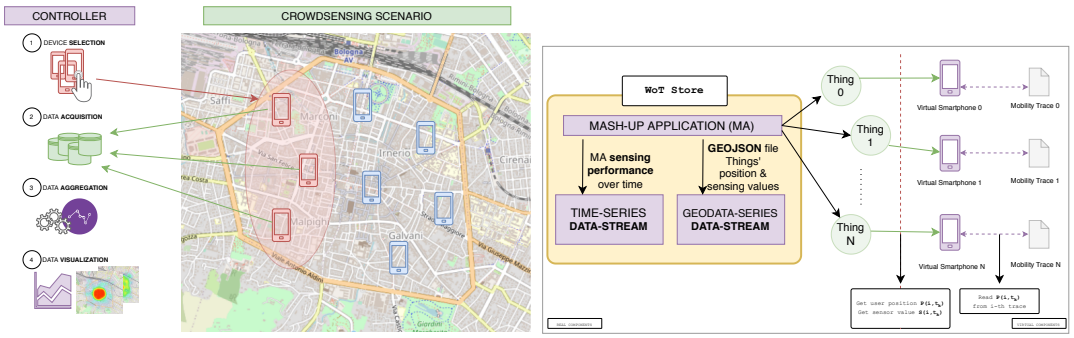
Fig. 3.10 The RTT and PDR values when switching the MA in use over time.

when the Wi-Fi devices stop sending the data because of the hardware crash. Figure 3.9(b) supports the discussion by showing the sensor utilization over time. For t <1200 seconds, the MA queries the BLE and two Zigbee Things, while from $1200 \leq t < 2100$, it mostly relies on the Wi-Fi Things; however, the Wi-Fi Things do not achieve the 100% of utilization due to random actions performed by the Q-learning for the periodic exploration phase [106]. Thanks to it, the Q-learning mechanism is able to discover alternative sensor selections once the Wi-Fi devices become not available (t >2100 seconds).

Figure 3.10 shows the RTT and PDR metrics over time when dynamically switching from one MA to another. Moreover specifically, from t=0 to t=3000, policy P_1 is used (delay minimization), then P_2 from t=3001 to t=6000 (PDR maximization), finally P_3 (delay-PDR trade-off) from $t > 6000$. We remark that the application replacement is performed through the WoT Store GUI, and consists of selecting a new software, and the Servient where to execute it. No hardware or software re-configuration of the WSNs is required. We can notice the values of the metrics (RTT and PDR) vary over time in accordance with the MA that is in execution in that temporal instant. We remark that a deeper analysis of this kind of scenario is presented later in section 4.1.

Urban Crowdsensing Scenario

In the second study, we consider an urban crowdsensing application composed of multiple, heterogeneous mobile devices (e.g. smartphones). Like in most existing crowdsensing systems [107], the mobile devices perform environmental sensing through their embedded sensors and transfer the data to a central processing unit; here, data are aggregated and analyzed. We assume that the central unit is also in charge of orchestrating the sensing operations, similarly to the testbed described in the previous Section. The overall architecture



(a) (b)

Fig. 3.11 The crowdsensing system considered in this study is depicted in Figure 3.11(a). The abstraction of the WoT deployment with the WoT Store and the real/simulated entities is represented in Figure 3.11(b).

of the crowdsensing system is reported in Figure 3.11(a). A W3C Thing is associated to each mobile device: the list of actions and properties is provided in Table 3.4. The system administrators can download MAs implementing different sensing policies from the WoT Store; in addition, they can aggregate and visualize the data gathered from the mobile devices through the Data Manager. Differently from the testbed, the system APIs of the Web Things do not query a physical device rather a simulated entity -denoted as Virtual Smartphone (VS)- that provides the current position and the result of each sensing action. In the following, we detail how the mobility and the sensing phases have been modeled.

Mobility simulation

We consider a pedestrian mobility model on a realistic city map (in this case, the downtown area of Bologna), extracting the street information from the OpenStreetMap web service[10]. A random direction model is considered: each user moves toward a random point of the scenario on the shortest path (computed over the graph of streets), and stops there for a random interval before selecting a new destination. We assume that the sensing and mobility phases are not mutually dependent; hence, the mobility traces of the N users have been generated offline and saved on N different files. The position information of each Thing (i.e. the *Latitude* and *Longitude* properties at each time slot t_k) is provided by the VS by reading the corresponding entry on the trace file owned by the Thing.

Event simulation

We model the sensing operations through a function that returns the sensing value at each location of the environment and at each time slot. To this purpose, we consider a generic situation where an event occurs in the urban scenario -and more specifically in its central

[10]https://www.openstreetmap.org

Name	Type	Description
PhoneID	Property	Smartphone unique identifier.
Latitude	Property	Current latitude coordinate.
Longitude	Property	Current longitude coordinate.
State	Property	Current state of the device (connected/disconnected).
GetSensingData	Action	Perform a sensor reading.
NewData	Event	This event is fired when new sensor data is produced.

Table 3.4 List of Properties, Actions, and Events of a Virtual Smartphone Thing in the crowdsensing scenario.

position- and the crowdsensing system is used to monitor the event and its evolution over time. Let $C = \langle c_{lat}, c_{long} \rangle$ denote the center of the scenario that coincides with the event origin. We abstract from the physical meaning of the event, of the sensing values and of the type of sensor in use, since they are not relevant for this study. Let $S(i,t_k)$ be the event sensing function that provides the intensity of the event as sensed by Thing i at time slot t_k (i.e., the values returned after invoking the *GetSensingData* action). The $S(i,t_k)$ function is modeled as follows:

$$S(i,t_k) = e^{\frac{d_i(t_k,C)}{\sigma}} \cdot I(t_k) + \chi \tag{3.1}$$

where $d_i(t_k,C)$ is the distance between the position of Thing i at time slot t_k (denoted as $P_i(t_k) = \langle lat_i(t_k), long_i(t_k) \rangle$) and the event origin C, σ is a normalization value, χ is a Gaussian noise with zero mean and variance equal to β (it models the sensing error of each device) and $I(t_k)$ is the function modeling the event intensity over time. Hence, on the spatial domain, the event intensity assumes the maximum value in C while it decreases proportionally with the distance from it. Let I_{max} and I_{min} be the maximum and minimum event intensity values. The $I(t_k)$ function defines an event with a time-varying intensity, i.e.: (*i*) it is equal to the minimal value (i.e. $I(t_k) = I_{min}$) until instant t_k=900 seconds; (*ii*) it increases linearly until instant t_k=1350 seconds, when the maximum value (I_{max}) is achieved; (*iii*) it remains equal to the maximum value (I_{max}) until instant t_k=2150 seconds; (*iv*) it decreases linearly until becoming equal again to the minimum value (I_{max}) at time instant t_k=2600 seconds. Let $I_{min} < \eta < I_{max}$ be a system threshold; a Thing/device[11] is said to detect the event at time t_k if $S(i,t_k) > \eta$.

[11] Things and devices are used indifferently in the following, since each Thing corresponds to one device; we used the word *device* when referring to the operations of the crowdsensing system, and *Thing* when referring to its implementation using the WoT architecture.

The WoT deployment of the crowdsensing system with real/simulated entities is shown in Figure 3.11(b). In order to save the battery of the mobile devices, the controller is querying only a subset of the available N devices at each sensing interval t_k. Let $\Psi(t_k)$ be such subset, and M be the number of devices queried, assumed constant over time. More formally: $M = |\Psi(t_k)| = \lfloor N \cdot \gamma \rfloor$, with $0 < \gamma \leq 1$. Also, we denote with $\Omega(t_k) \subseteq \Psi(t_k)$ the list of devices detecting the event at time slot t_k, i.e $\Omega(t_k) = \{i \in \Psi(t_k) | S(i,t_k) > \eta\}$. We implemented two MAs in WoT Store with differentiated policies to compute the $\Psi(t_k)$ set:

- *Random* MA. At each sensing interval t_k, the MA chooses randomly the subset of M sensors to query among the available N.
- *Adaptive* MA. Like the previous policy, the MA chooses M sensor to query at each sensing interval; it chooses them randomly if $|\Omega(t_k)| \leq \alpha$, the number of devices detecting the event is below a system threshold α. Vice versa, when $|\Omega(t_k)| > \alpha$, the MA attempts to estimate the area where the event is occurring and to concentrate the sensing operations over it. To this purpose, the position of the event $C^{est}(t_k) = \langle c^{est}_{lat}(t_k), c^{est}_{long}(t_k) \rangle$ at time t_k is estimated as the centroid of the position of the users detecting it, i.e.:

$$c^{est}_{lat}(t_k) = \frac{\sum_{i \in \Omega(t_k)} lat_i(t_k)}{|\Omega(t_k)|} \tag{3.2}$$

$$c^{est}_{long}(t_k) = \frac{\sum_{i \in \Omega(t_k)} long_i(t_k)}{|\Omega(t_k)|} \tag{3.3}$$

  Similarly, the radius of the event $R(t_k)$ is estimated as the maximum distance between $C^{est}(t_k)$ and all the devices that detected the event in $\Omega(t_k)$, i.e. $R(t_k) = max_{i \in \Omega(t_k)} (d_i(t_k, C^{est}(t_k)))$. In order to build the list of devices to query at the next time slot (i.e. $\Psi(t_{k+1})$), we consider only the devices at a distance lower than $R(t_k)$ from c^{est}_{long}. Let $\Gamma(t_k)$ denote such set. Then, we order $\Gamma(t_k)$ according to the distance values $d_i(t_k, C^{est}(t_k))$, and we select the top M elements. In case $|\Psi(t_{k+1})| < M$, the remaining $M - |\Psi(t_{k+1})|$ devices are randomly chosen as for the Random Policy.

Clearly, much more complex MAs can be defined for the scenario in use. However, we remark that the goal of the study is not on the crowdsensing algorithms rather on the deployment and execution of MAs through the WoT Store. Unless stated otherwise, we used the following parameters in the tests: N=400, M=80, γ=0.2, η=5, α=2, β=2, σ=300.

We implemented two data streams in the Data Manager, one handling time-series data and the other handling geographic data, coded in GEOJSON. Each MA generates two time-series:

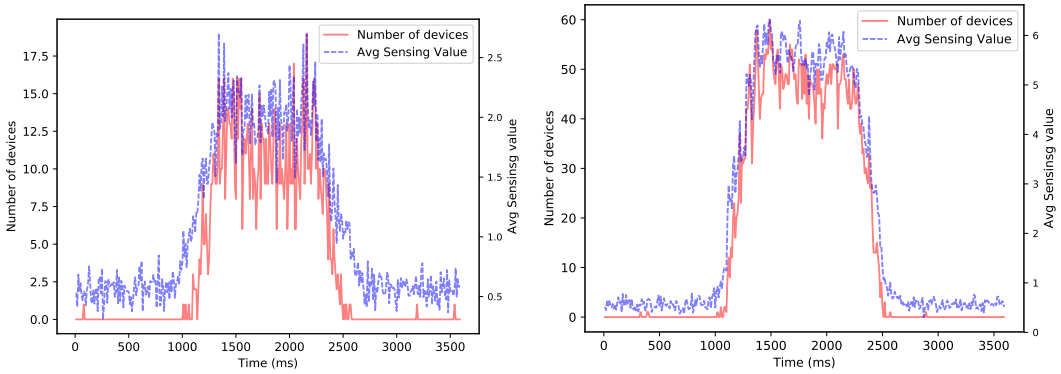
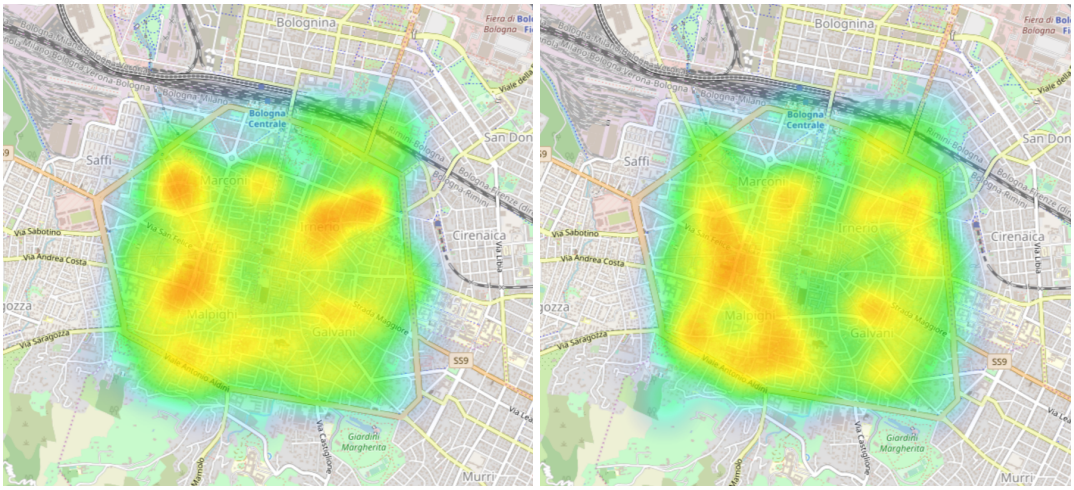


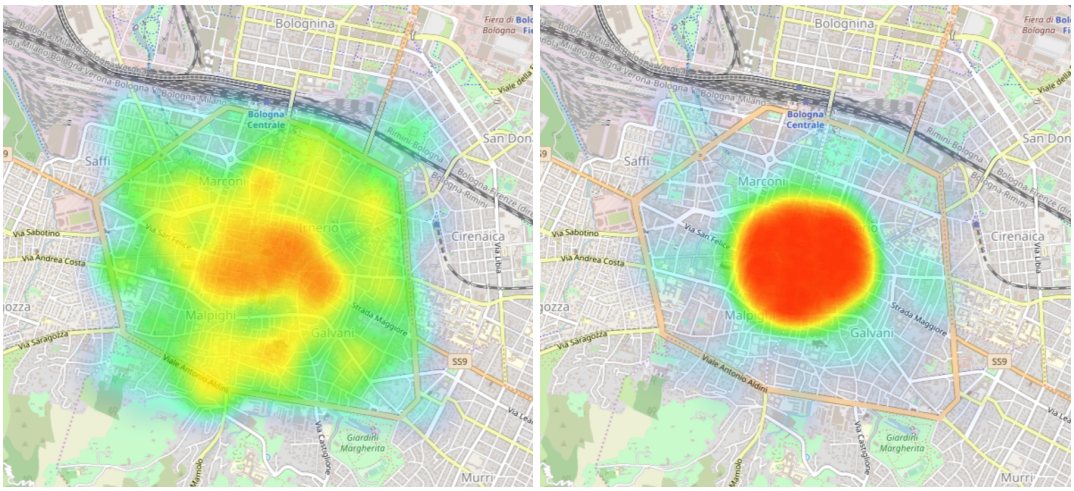
Fig. 3.12 The average sensing value and the number of Things detecting the event for the *Random* MA are shown in Figure 3.12(a). The same metrics for the *Adaptive* MA are shown in Figure 3.12(b).

(*i*) the average sensing intensity at each t_k, computed as the average value over the M queried devices, i.e. $\frac{\sum_{i \in \Psi(t_k)} S(i,t_k)}{M}$, and (*ii*) the number of devices detecting the event at each t_k, i.e. $|\Omega(t_k)|$. For readability reasons, we visualized both the time-sequences in the same plot. Figure 3.12(a) refers to the *Random* MA, while Figure 3.12(b) to the *Adaptive* MA. Both the plots show a similar trend, since the average sensing intensity follows the variations over time of the event intensity, provided by the $I(t_k)$ values; however, it is easy to notice that the absolute values are much greater for the *Adaptive* MA compared to the *Random* MA, since the sensing activities are focused on the area where the event is occurring. As a result, the number of devices detecting the event is also significantly higher for the *Adaptive* MA. In addition to the time-series, at pre-defined time slots each MA generates a GEOJSON file, containing the position (i.e. $P_i(t_k)$) and the instantaneous sensing value (i.e. $S(i,t_k)$) of the Things queried (i.e. belonging to the set $\Psi(t_K)$). The Data Manager allows visualizing the GEOJSON file as heatmaps. Figure 3.13(a) and 3.13(b) show the heatmaps of the *Random* MA at t_k=100 and t_k=1000, i.e. before and during the occurrence of the event. We can notice that the sensing actions of the *Random* MA are equally distributed over the scenario in both cases. Figure 3.14(a) and 3.14(b) show the heatmaps of the *Adaptive* MA at the same time slots. Before the occurrence of the event (Figure 3.14(a)), the *Adaptive* MA behaves similarly to the *Random* MA since no device has detected the event, and hence the device selection is performed randomly. Vice versa, after the occurrence (Figure 3.14(b)), most of the M sensing actions are performed on the central area of the scenario, i.e. on the estimated event origin C^{est} that also coincides with the real event origin C. This result further justifies the higher performance of the *Adaptive* MA in terms of number of devices detecting the event compared



(a) (b)

Fig. 3.13 The geodata stream visualization for the *Random* MA before the occurrence of the event (Figure 3.13(a)) and during the occurrence (Figure 3.13(b)).



(a) (b)

Fig. 3.14 The geodata stream visualization for the *Adaptive* MA before the occurrence of the event (Figure 3.14(a)) and during the occurrence (Figure 3.14(b)).

to the *Random* MA (Figures 3.12(a) and 3.12(b)). In conclusion, the Data Manager can be useful to monitor the crowdsensing operations over both the time and space dimensions; moreover, since different MA can be installed from the WoT Store, the Data Manager allows to compare the performance of different sensing control policies in a straightforward way.

We conclude the analysis by investigating the scalability of our platform when increasing the number of deployed Things. Figure 3.15 shows the usage of resources (RAM utilization,

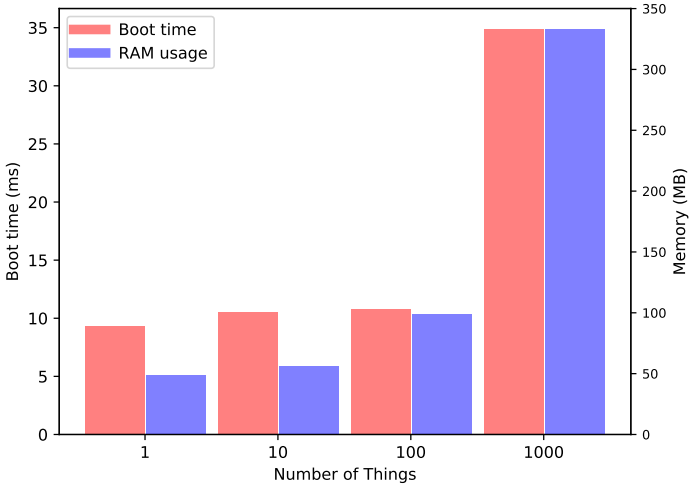


Fig. 3.15 The resource (CPU, RAM) utilization of the WoT Store for increasing number of deployed Things in the crowdsensing scenario.

CPU time) of the machine[12] hosting the Servient. It is easy to see that the resource utilization increases linearly with the number of Things to manage, and in any case no performance bottlenecks are introduced even for large-scale WoT deployments.

[12]Core i5 7600 Kaby 3,5GHz with 16GB RAM DDR4 and ArchLinux OS.

3.2 Web of Things as REST APIs and communication with legacy IoT Systems

The WoT Store has been explicitly designed to represent a concrete solution for all those IoT scenarios in which the Web of Things could mitigate the interoperability problems. WoT is, nowadays, the most promising standardization effort within the application layer of the IoT, however there can be many cases in which legacy, obsolete, or even proprietary systems need an ad-hoc integration with such world. For this reason, in this Section we propose a *bridge* solution for interconnecting these systems to the WoT Store, hence enabling the transition to the Web of Things. Additionally, this proposal aims at encouraging developers and system maintainers to make use of the WoT Store - with all its WoT-native services and applications - even in cases where this transition might be complex and not straightforward. From this point of view, the following study shows how the integration can be twofold: on one hand, it brings WoT services and devices to IoT systems in a transparent way, making them available to be easily consumed by the IoT services. On the other hand, it lets the WoT Store use services registered and integrated into old legacy IoT systems. For this purpose, the Arrowhead Framework [16] has been chosen as a candidate for such IoT systems, since its architecture and nature represent a good example for such study. In particular, besides a consistent set of other useful functionalities, it perfectly suits this use case because every IoT service is exposed through a REST API, and for this reason, the integration with the WoT Store ecosystem is quite straightforward. Clearly, despite this particular case, the goal of this study is to be as generic as possible, since the same approach can be replicated for hundreds of other IoT systems with similar characteristics: the only requirement in this sense is to expose functionalities through some kind of APIs. We believe that this kind of integration can significantly boost the usage of the WoT Store, easing the design and developing tasks especially in those situations where Web technologies are already in place.

3.2.1 Overview

Over the last decades, ecosystems revolving around IoT in its various facets have shown the common trend of shifting from monolithic or ad-hoc deployments to architectures in which each entity is responsible for producing or consuming services, as in any Service-Oriented Architecture. The Arrowhead Framework (AHF) [16]is the result of an effort of more than

80 European partners [108] and has been used extensively in several other EU initiatives such as Productive 4.0[13], Far-Edge[14] and Arrowhead Tools[15].

The Framework, designed for supporting IoT automation scenarios at any application level, is based on the key guidelines that characterize a SOA: late binding, loose coupling, and lookup [109]. More in detail, each System of Systems (SoS) based on the AHF is deployed in connected local clouds, each of them managing their internal services and communicating with each other in a non-hierarchical structure, therefore separating responsibilities while still guaranteeing interoperability. Each local cloud hosts several Systems, defined as the software components that interact with each other and shape the application workflow. Systems can expose a number of Services as well as consume other services in the network, they are indeed defined as Service Providers or Service Consumers (clearly any system can be both). The interaction between systems and services within each local cloud is given by the "Core Systems" (CS) - one instance is deployed per local cloud - that support and orchestrate the exchange of information. They are divided into "Mandatory" CS, which have to be deployed within a local cloud to make it Arrowhead-compatible, and "Support CS" [110]. Mandatory CS are described in detail below:

- The **Service Registry** system is responsible for the registration of each service within the local cloud. It acts as a repository, against which other systems can perform a service lookup - i.e. a discovery operation - in order to obtain the information and the endpoint of the service they are looking for. In the last version of the CS (4.1.3 at the time of writing), the service lookup is performed through HTTP REST calls.

- The **Authorization** system is responsible for the correct interaction between producers and consumers according to their rights. It manages the correct authentication of providers and consumers as well as their authorization for consuming or producing resources based on a set of rules that can be added or modified by the cloud manager.

- The **Orchestration** system is responsible for coordinating the interactions between systems freeing the consumers from the burden of establishing their preferences at design time. With the Orchestration system, the Service Provider that is best suitable for the consumer's request can be chosen dynamically based on a list of orchestration rules about the type of service requested. This can potentially handle cases of faults and perform load balancing.

[13]https://productive40.eu/
[14]http://faredge.eu/#/
[15]https://www.arrowhead.eu/arrowheadtools

Support CS are not mandatory and can be included in any local cloud where needed. Examples of available Support CS are: QoS Manager, Translator System, Event Handler, and Configuration Manager. Furthermore, the Gatekeeper System and the Gateway System, which are still Support CS, are devoted to the inter-cloud communication, mediating the exchange of, respectively, lookup requests and chunks of data [111]. The latest version of all the AHF components (4.1.3 at the time of writing) can be found in [46].

In this proposal, we analyze a possible integration between the Arrowhead Framework and the W3C Web of Things (WoT) paradigm. In particular, the goal is to enable the twofold communication between the WoT Store and other IoT legacy systems, i.e., making possible for Web Things to request services registered on Arrowhead and the same time letting Arrowhead services use Web Things' capabilities. For this last purpose, Web Things could be considered as a simple REST APIs. The integration is shown through a proposed three-layered architecture in which a standalone WoT ecosystem is integrated within an Arrowhead local cloud and vice versa.

More in detail, the contributions of this study can be summarized by the following three points:

- we propose a three-layered architecture thanks to which clients compatible with both the Arrowhead Framework and the W3C WoT will be able to interact with the IoT devices.
- we design an essential middleware component, defined as WAE, that acts as a discovery bridge for the WoT layer.
- we validate the proposed architecture providing performance evaluation in a real-world scenario in which a multitude of Web Things is instantiated and published to the main service broker.

3.2.2 Architecture

In this Section, we define in detail the architectural structure of our proposal for enabling the communication within Web Things and services available at the Arrowhead Framework. Despite the great potential of WoT paradigm, since the process of making an already existing service W3C WoT-compliant requires some effort, there can still be cases - especially those involving old legacy systems - in which a service could be interested to benefit from capabilities offered by Web Things, without joining the WoT ecosystem. At the same time, the same services could offer useful information to native Web Things without being necessarily turned into Web Things too. This proposal hence is also designed for such

components that may either be unable to understand the same language and use the same protocols as the WoT ecosystem does, or be unaware of the location of the Web Things that need to be queried. More in detail, we envision an ecosystem in which a set of Web Things, which are devoted to collect data through sensors, expose their data to potential consumers that are external to their ecosystem. Simultaneously, a set of legacy services registered on Arrowhead offer data that can be consumed by some Web Things, in order to augment the set of their capabilities. In particular, the architecture proposed mainly focuses on enabling the interactions for:

- a consumer that is able to communicate using the WoT standard and that does not know how to reach and query the legacy IoT services registered on Arrowhead.
- a consumer that has no information about the WoT ecosystem and communicates only using another legacy protocol (say, HTTP).

In order for the whole ecosystem or the System of Systems (SoS), as it is defined in the Arrowhead official documentation model[16], to be able to cope with such cases, we propose an architecture in which each Web Thing is also a Service Provider of an Arrowhead local cloud, therefore exposing sensor data as a service. The Arrowhead Framework allows indeed each service to be discoverable through its main component: the Service Registry. As anticipated in Section 3.2.1, the Service Registry acts as the main service broker in a SOA: for each service in the local cloud that advertises its endpoint through a publish call, it keeps in memory a service record, encoded in JSON, that includes a set of service metadata. The record includes the type of service, its endpoint, and the protocol used, although other metadata can be added upon need. In a typical and simple scenario, a Service Provider first publishes its service record, then a Service Consumer willing to consume such service performs a service lookup call against the Service Registry and obtains information about the endpoint and the protocol of the desired service. Once this information is held by the Service Consumer, the communication with the Service Registry is no longer needed and the Service Provider and the Service Consumer can communicate directly.

Note that in an Arrowhead service consumption the Orchestration module also has its part: it gets queried by the Service Consumer for a service of a defined type and it searches the Service Registry for the most suitable service record, based on a set of rules. The use of the Orchestration service is out of the scope of this study, therefore we intentionally simplify the interaction bypassing the Orchestration and only demonstrating the architectural integration.

[16]https://www.arrowhead.eu/arrowheadframework/this-is-it/documentation-model/

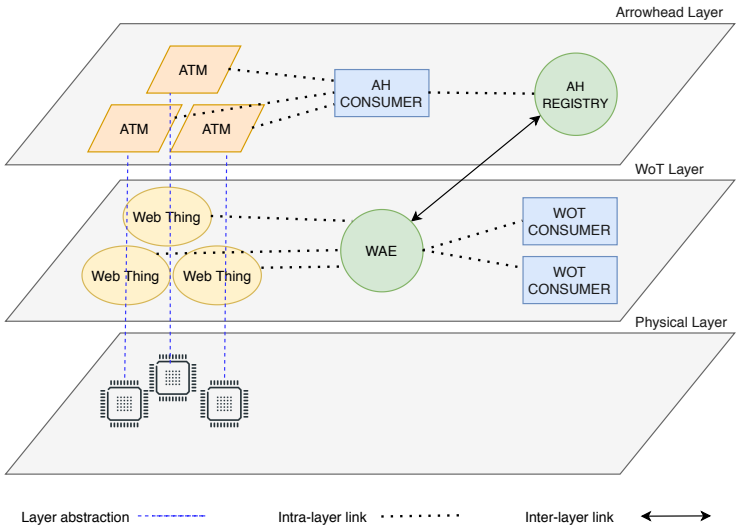


Fig. 3.16 The System Architecture

Service Interactions

In order to support different types of external consumers interacting with the WoT ecosystem, the layered architecture in Figure 3.16 is proposed. The architecture consists of three conceptual layers: the Physical layer, the WoT Layer, and the Arrowhead Layer. Entities on each layer can communicate directly with other entities belonging to the same layer as they are assumed to use the same application protocol. For different layers, instead, entities either have an abstraction or an inter-layer communication channel, as will be explained later. On the bottom-left corner the physical sensors are depicted, the only entity of the physical layer. Sensors can be of any type as long as they produce a numerical observation from the real world. Each sensor gets abstracted onto a Web Thing, according to the WoT paradigm. Each Web Thing is then registered onto the Thing Directory of the WoT Store. The central component of the WoT Layer, the WoT Arrowhead Enabler (WAE), can be classified as a WoT Mashup Application, a concept previously outlined in 3.1.2. In detail, it periodically queries the Thing Directory to detect new Web Things right after they spawn (i.e. the binding with the actual sensor is generated). As new Web Things are detected, the WAE performs a publish operation for each of them against the Service Registry in the Arrowhead Layer to publish Web Things as new Arrowhead services. The communication between the WAE and the SR is the sole case of the inter-layer communication channel, in which a component (in this case the WAE) acts as a proxy able to use two different communication protocols. Furthermore, each Web Thing is extended onto the Arrowhead Layer by a new module, called Arrowhead Thing Mirror (ATM). The ATM exposes the Web Thing service endpoint as an HTTP Web Service in the Arrowhead local cloud. Note that a Web Thing and its relative

ATM can run on the same piece of software as well as in separate components connected by a custom communication link. The ATM plays, to some extent, the role of an Arrowhead service adapter, however, it does not perform publish operation, as they are handled by the WAE.

The record published by the WAE exposes by default the endpoint and the metadata of the ATM related to the Thing, while the JSON-LD description of the Thing at the WoT Layer must be retrieved through its endpoint. This way, a consumer can interact with the Web Thing in two ways, depending on its communication capabilities:

- An HTTP-enabled Consumer queries the Service Registry, selects the service that provides the type of data needed and gets the endpoint of the service, which corresponds to the endpoint of the related ATM. The Consumer then performs the consume calls against the service offered by the ATM which, in turn, queries the Web Thing and retrieves the data point. Response data travels then backwards to the Consumer.

- A WoT-enabled Consumer queries the WAE, which retrieves the list of services from the Service Registry. As the consumer is only able to interact with WoT-enabled systems, the WAE acts as a Web Thing Proxy, whose affordances reflect the capabilities of the legacy service required by the Web Thing. The proxy hence is in charge of turning the request coming from a Web Thing into a REST interaction, collecting the result of the query and returning the result back to the Web Thing.

The whole interaction for the two types of consumers is shown in detail through the sequence diagram in Figure 3.17. In particular, it shows mainly two patterns:

Web Things' publication on SR

When a Web Thing is generated, it automatically instantiates an ATM to be able to fulfill a request coming from an AH Consumer. At the same time, the Web Thing publishes itself on the Thing Directory, according to the draft of the W3C standard proposal. The WAE is in charge of keep checking if new Web Things appear on the Thing Directory, and in case to publish themselves on the SR. This can be achieved in two ways, depending on the WAE implementation: either the WAE polls the Thing Directory or the WAE is implemented as a Web Thing, so it can subscribe to Thing Directory's events. The SR is listening from queries of services' consumers and replies with all the Services is aware of, including the Arrowhead ones that however are not shown in this diagram.

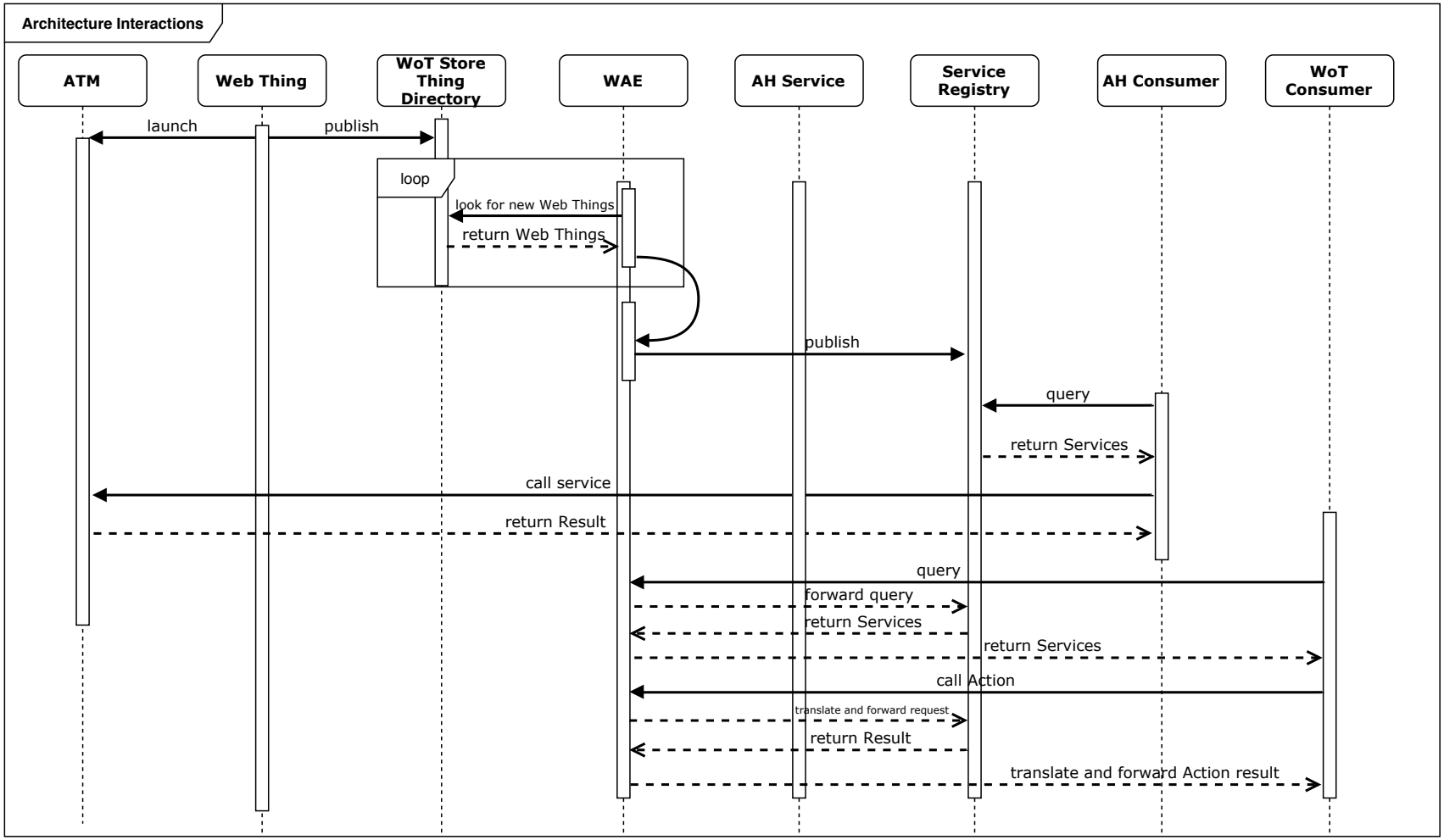


Fig. 3.17 Sequence diagram presenting the interactions of all the components of the three-layer architecture

Name	Type	Description
listOfWebThings	Property	List of all the Web Things the WAE is aware of.
startCrawling	Action	Start to look for new Web Things that are published on the TD.
query	Action	Forward the query of a Wot Consumer to the WAE.
proxyQuery	Action	Forward and translate a query coming from a Web Thing to an AH service and returns the response.
newWebThing	Event	This event is fired when a new Web Thing has been discovered on the TD by the WAE.

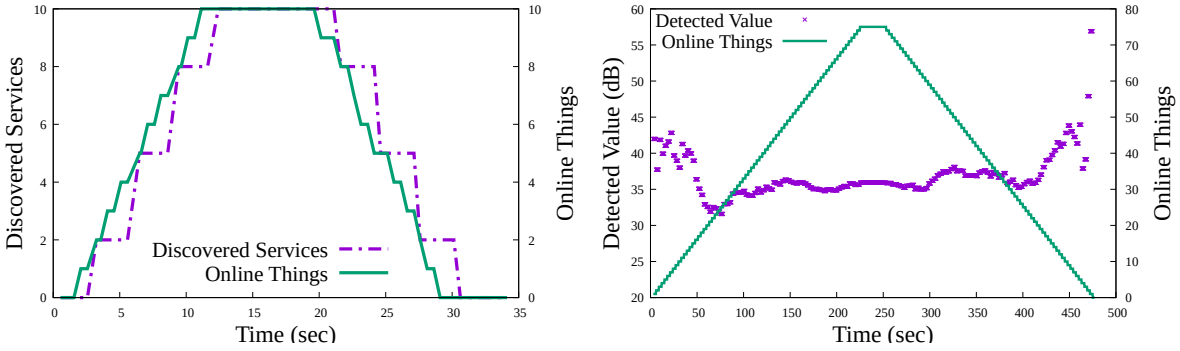
Table 3.5 List of Properties, Actions, and Events of the WAE Thing.

Services/Web Things' consuming

Once the services related to Web Things become available on the SR, they can be queried by consumers. An AH Consumer has only to query the SR, to get the list of services and to directly interact with them through their ATM. A Wot Consumer instead requires more steps in order to be able to interact with a legacy service. First, it has to send the query to the WAE in order to retrieve the list of available services from the Arrowhead registry. Hence, the WAE forwards the request to the SR and waits for the list of services that match the query originally coming from the WoT Consumer. After that, the WAE returns the list to the WoT Consumer, which is now able to select the proper service from the list according to its needs. The Web Thing communicates to the WAE the service it is interested in, which then starts the proxy service to be able to handle and satisfy all the requests for the legacy service coming from the Web Thing.

3.2.3 Implementation and Validation

The implementation of the main components of the architecture is here briefly described. The Service Registry is a JAVA server that exposes some REST APIs. In particular, we used the API already available as an open-source project [46]. All Web Things involved in the scenario have been implemented and instantiated by using *node-wot* [25], the official W3C framework for the WoT. The WAE component has been designed as a Web Thing - for being able to natively speak to other W3C WoT entities - and as an HTTP client - in order to use the SR's APIs. As shown in table 3.5, following the paradigm *Properties, Action, Events* explained in section 2.3, the WAE Web Thing exposes the *listOfWebThings* Property for listing all the already known Web Things it has published. Additionally, it exposes also the *startCrawling* and the *query* actions. The first is automatically invoked once the



(a) (b)

Fig. 3.18 Figure 3.18(a) shows the Online Things vs Discovered Services, while Figure 3.18(b) shows the mean detected value over all the sensors.

WAE has been deployed to look for new Web Things that have been published on the Thing Directory. The second one is invoked by a WoT Consumer in order to query the SR and to get the list of services that match its request. Finally, a generic event *newWebThing* is fired each time new Web Things have been discovered by the WAE. Both the HTTP client of WAE and the AH Consumer have been customized for the need by taking advantage of the already existing open-source NodeJS Arrowhead Client [46]. Each WoT Consumer is a *Mashup Application*, i.e., a javascript application that uses the *node-wot* framework as a library and simply consumes multiple Things to interact with them in order to collect data and manipulate it for its needs. Lastly, the ATM is an *ExpressJS* web server that maps each Web Thing Affordance to a REST API and that uses the *node-wot* as a library behind the scenes in order to interact with the Web Thing it represents.

In order to validate this proposal, we created a proof-of-concept scenario where the components of the architecture previously described were deployed. The goal of the validation is dual: first, we want to prove the functionalities of the Arrowhead discovery in conjunction with the WoT ecosystem. Secondly, we want to show the benefits of such discovery method for a generic Consumer that is agnostic of the real nature of the services used for its application. In particular, we instantiated a WoT Arrowhead Enabler (WAE) which is in charge of discovering new Web Things and publishing them on the Arrowhead SR to make them available to all the possible legacy consumers. Each new Web Thing is launched after λ seconds from the previous one, and then published as soon as it has been discovered by the WAE. Additionally, after a pre-defined $TIME_{MAX}$ interval of time, Web Things start disconnecting every λ seconds and so they are unregistered from the SR. This means that, depending on the WAE's Update Frequency (WUP), there could be some delay before Web Things become available/unavailable on the SR. Figure 3.18(a) shows the total number of

services made available on the SR and the number of Web Things online. In this case, since $\lambda = 1$ second and $WUP = 3$ seconds, the plot shows the misalignment between the online Web Things and the ones registered as Arrowhead services in the Arrowhead SR. Both Web Things and their correspondent Arrowhead services increase until $TIME_{MAX}$ is reached and then they start decreasing accordingly respectively to λ and WUP frequencies. While in the first case the focus is on validating the Thing Discovery, in the second case we set up a testbed for validating a Consumer entity. In particular, we instantiated an Arrowhead consumer that retrieves values from some services and elaborates them. More in detail, the services are WoT services that return the value of acoustic sensors, with an error estimated of $\pm 5dB$, while the application's goal is to detect the walk of a human inside a room. The application first queries the Arrowhead SR for looking for all the available sensors, then it retrieves the detected values and computes the average of the value obtained. Depending on the use case, a threshold can be set for identifying a particular feature. Figure 3.18(b) shows the behaviour of the application over the time, with the number of Web Thing Services that changes over the time. For this testbed, the parameters are set to: $\lambda = 3$ seconds, $WUP = 1$ second, and $TIME_{MAX} = 250$ seconds. It is clear that the more WoT Things services contribute to the detection phase, and the more the average gets closer to 35 dB, that is a reasonable sound level for human walking.

Chapter 4

Use cases validation

Considering that the Web of Things is eligible to be adopted in almost every IoT scenario, the WoT Store can be potentially deployed in all the cases where the Web of Things is required to mitigate the IoT interoperability problems. This means, among the others, that one fundamental requirement for the WoT Store is the ability to dynamically adapt to the conditions of the environment, enabling features and functionalities customized for that specific case. Although the WoT Store includes a market functionality to share and use third-party WoT applications designed for a specific use case, a scalable solution to this problem cannot be represented only by the possibility to find an existing application ready for all the possible IoT scenarios. Instead, the WoT Store proposes to make available to developers and maintainers a set of tools that can be customized and configured depending on specific needs. For this reason, in this Section we introduce and show two use cases - taken as examples - where the WoT Store has been effectively used and analyzed for validating its effectiveness as well as its potential.

Similarly to the Web of Things architecture proposed in [38], and following the same level of abstraction, the WoT Store considered as a software stack can be mapped into a layered architecture, as shown in Figure 4.1: on the bottom layers we can find the *Network Layer* and the *Access Layer*, that are responsible for the communication and the manipulation of the Web Things. More in detail, the first one is in charge of enabling the communication between devices that natively do not speak any Web protocol, and hence cannot be mapped into W3C native Web Things. For instance, these include all kinds of micro-controllers, or devices that speak network protocols of lower level, like *Zigbee* or *Bluetooth*. The second instead allows to add/remove devices that have been turned into W3C Web Things or that can natively communicate through a Web protocol - or a protocol that can be wrapped into one of them -, like *HTTP* or *CoAP*. Furthermore, it allows users to interact with the Web Things. The upper layers are the *Find and Share Layer* and the *Application Layer* that include all the

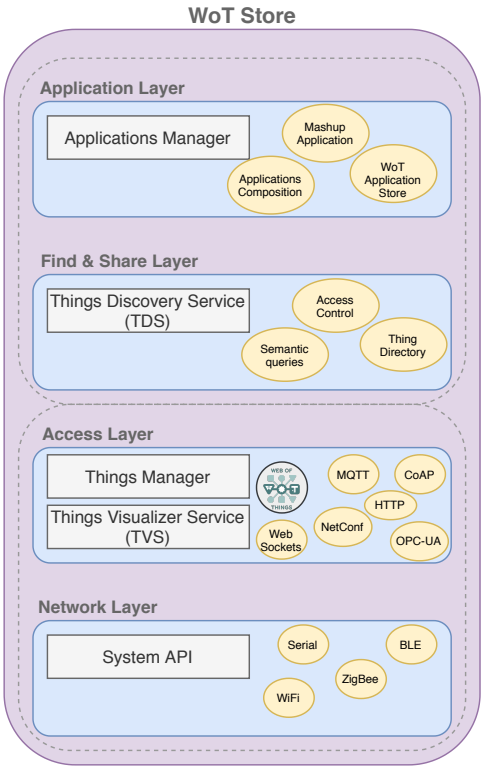


Fig. 4.1 The WoT Store Architecture mapped into a layered architecture

functionalities for discovering and managing services built upon Web Things. In particular, the first one is in charge of providing services for registering/unregistering the Web Things on the WoT Store, as well as services for retrieving them via semantic queries. The last layer allows users to look for applications already present in the WoT Store and to load them as new behaviors of the devices, or to use them as services that make use of the Web Things handled by the WoT Store. That said, in this Chapter we present two contributions whose goal is to validate the WoT Store from the layers perspective: the first study deeply investigates the ability of the WoT Store to be used in environments characterized by the presence of heterogeneous devices that do not speak Web protocols. On top of them an orchestrator application is in charge of collecting data based on different policies that take into account some parameters strictly influenced by the network nature used by the devices for communicating, like the `RTT` or `Packet Delivery Ratio`. In this sense, the study principally focuses on the first and last layer of the architecture presented so far. Instead, the second contribution is about a SHM scenario and almost covers all the layers, since the goal is to collect and analyze data in order to make predictive maintenance of buildings. Sensors built on purpose have been modeled and accurately turned into Web Things, so validating the first and second layers. On top of them, a precise searching mechanism has been designed for

selecting and discovering Web Things, possibly handling different access policies. Finally, the data collected is visualized and manipulated through the last layer in order to enable services for predictive analysis. We remark that, differently from the component validation proposed in Section 3.1.2, this Chapter aims at validating the WoT Store in its whole, and precisely by focusing on the architectural layers as previously explained.

4.1 Heterogeneous environmental monitoring

4.1.1 Heterogeneous Sensing Scenario

Recently, the Industry 4.0 has emerged as a new paradigm able to radically transform the organizations' production and business in a myriad of sectors beside the smart manufacturing one [8] [9]. The core of the paradigm that justifies also its generality and viability on different markets is the concept of Cyber-Physical Systems (CPSs), i.e. the strict integration between physical elements and computational data enabled by the recent advances on the Internet of Things (IoT) [8]. Hence, the ability to collect, aggregate and analyze sensor data is crucial for the growth of the Industry 4.0 model. At the same time, today's IoT is a chaotic environment characterized by heterogeneous hardware devices, network protocol stacks and data formats. The generality of the WoT architecture makes it suitable for all those scenarios characterized by the need of aggregating data from multiple, heterogeneous sources, like the Industry 4.0. However, due also to its recent appearance, few implementations and test-bed of the W3C WoT have been described so far in the literature [112][102].
In this Section, we attempt to extend the WoT Store proposal as enabler for the Industry 4.0, by describing the design and implementation of a WoT testbed, consisting of a monitoring system of a generic production site that must retrieve and process sensor data from heterogeneous devices using different wireless access technologies (i.e. Wi-Fi, 802.15.4/Zigbee, BLE). The overall goal is to devise mash-up applications able to orchestrate the sensing operations over the target scenario regardless of the network protocols and hardware, hence decoupling the rationale of the monitoring process (e.g. minimal scenario coverage) from its implementation (i.e. the technology used to query the sensor). More specifically, we introduce three main contributions in this study:

- First, we describe how the scenario can be modeled within the WoT W3C framework. One Thing is associated to each sensing device, and one Thing to the sensor network, by defining the metadata of each. Moreover, we discuss how the components of the WoT W3C architecture have been concretely modeled in this application.

- Second, we describe the design and implementation of mash-up applications aimed to orchestrate the sensing operations on the target scenario. Four different sensing policies are taking into account, aimed to balance the coverage of the scenario with the network performance (e.g. delay, packet delivery ratio and energy). All the policies are in charge of dynamically selecting the sensors to query at each instant in order to maximize the policy-specific metric: to this purpose, given the dynamism of the environment, we

employ the Reinforcement Learning (RL) framework [106] to optimally balance the exploration-exploitation tasks.

- Third, we report a subset of the experimental results from the WoT testbed. We investigate the performance of the sensing mash-up applications with respect to the policy goal (e.g. delay), and the convergence over time. Moreover, we show the benefit introduced by the WoT architecture in terms of adaptive design, i.e. the possibility to dynamically switch the sensing policies over time without re-configuring the communication infrastructure, and the overhead introduced by the WoT components.

4.1.2 Testbed and Architecture

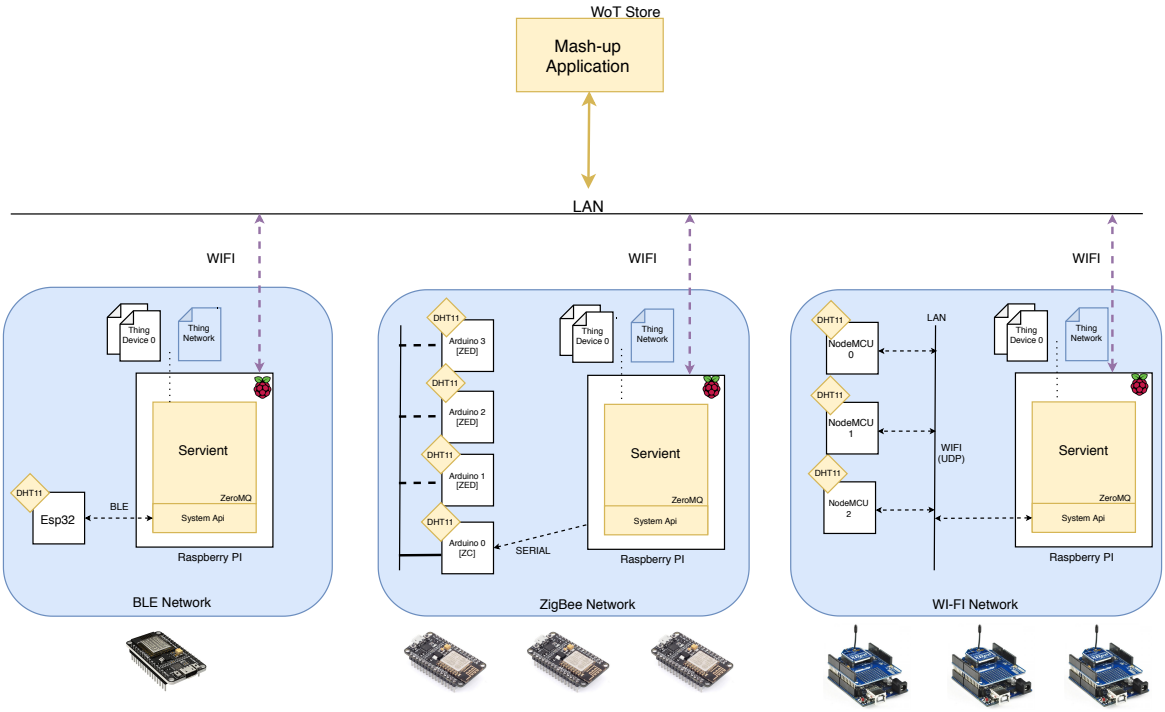


Fig. 4.2 The IoT/WoT monitoring system deployed in this study.

The goal of this study is to investigate the suitability - both in terms of ease of deployment and of performance - of the W3C WoT architecture for Industry 4.0 applications. To this purpose, a generic IoT monitoring system of a production site is considered, characterized by the presence of heterogeneous sensors using different communication technologies. The overall architecture of the testbed, depicted in Figure 4.2, is structured on three tiers:

- **Edge layer**. This layer is composed of three Wireless Sensor Networks (WSNs), operating over the same environment: an IEEE 802.15.4 WSN network, an IEEE

802.11 Wi-Fi WSN network and a BLE device. The 802.15.4 network includes four devices (*Arduino Xbee* boards), with one Coordinator and three Leaf nodes equipped with sensing units (*ThinkerKit* temperature sensor). The Wi-Fi network includes three devices (two *NodeMCU* and one *Arduino WiFly* board), all provided with a direct link toward the Access Point (AP) and with a *DHT11* temperature/humidity sensor. Finally, the BLE WSN consists of one *ESP32* board, provided with a *DHT11* sensor.

- **Fog layer**. The 802.15.4 coordinator, the BLE and the Wi-Fi devices are connected to the corresponding Fog node, via USB cable links (for the 802.15.4 Coordinator) or Wireless links (for the BLE and the IEEE 802.11 devices). Each fog node is constituted by a *Raspberry PI3B+* board and it is in charge of exposing the corresponding Web avatar (i.e. the Web Thing) for each managed device and WSN.

- **Processing layer**. This layer implements the logic of the monitoring system. It is constituted by a local instance of the WoT Store running the mash-up applications further defined in Section 4.1.3, and connected to the Fog nodes via Wi-Fi links. More specifically, the layer is in charge of: (*i*) orchestrating the sensing operations, by properly selecting the devices to query at each time slot according to the policies of Section 4.1.3; (*ii*) storing the collected data within a time-series database; (*iii*) processing and analyzing the data in order to implement the Digital Twin model of the monitored site.

We omit the data analytics process and the creation of the Digital Twin model, and instead, we detail here the data retrieval operations, and specifically the way we implemented the WoT W3C components of the architecture reported in Figure 4.2. More in detail:

- **Edge** devices implement low-level communication and sensing operations in the embedded firmware. The implementation as well as the list of operations and the data format used by each device is technology-dependent. This layer is part of the IoT, while it is not covered by the WoT architecture.

- **Fog nodes** run a W3C WoT Servient, by using the official JavaScript (JS) framework provided by the W3C [25]. Each Fog node exposes two types of Web Things, i.e.: multiple (*i*) *Thing Devices*, describing the properties, events and actions of physically managed edge devices, and one (*ii*) *Thing Network*, describing the overall performance of the virtual WSN composed by the list of connected Thing Devices. Moreover, we consider three possible protocol bindings for each Thing, i.e. interaction modes with the Things, based on the HTTP (default choice), the CoAP or the MQTT protocols. The System APIs are implemented in Javascript, and further structured into two

Name	Type	Description
DeviceID	Property	Device identifier in the network.
NetworkID	Property	Network identifier the device belongs to.
Temperature	Property	Last temperature value.
State	Property	Current state of the device.
GetData	Action	Get the temperature data.
Start	Action	Start sending data at each time slot.
Stop	Action	Stop sending data.
NewData	Event	This event is fired when new sensor data is produced.
ChangeState	Event	This event is fired when the connection state changes.

Table 4.1 Example of Properties, Actions, and Events described in a Thing Description of a Device Thing.

layers, i.e.: (*i*) a *Device Query* level, that is in charge of issuing request-response communication with the Edge device, based on the wireless technology and the protocol stack supported by this latter (e.g. UDP socket for the WiFi devices, Serial socket for the Zigbee Coordinator, BLE connected mode for the BLE device), (*ii*) an *Inter-Process Communication* (IPC) level, that makes the sensor data available to the upper Scripting APIs via IPC facilities (in our case, implemented through the ZeroMQ library[1]).

- Finally, the **Processing node** interacts with each Fog node/Servient in order to consume Things, e.g. by periodically *invoking* the `getData` *action* from the Things selected according to the actual mash-up policy.

Table 4.1 shows some of the properties, actions, and events described in the Thing Description (TD) for a Device Thing. The TD of a Network Thing includes only properties that are referred to the average network performance (i.e. the delay, the packet delivery ratio and the throughput) and actions that can be invoked from the entire network, like for instance *getAllData()*. Similarly, the snippet below shows a code fragment of the mash-up application, specifically the way how to query a sensor device in order to read its temperature value. We can notice that - through the WoT architecture - the mash-up application is agnostic on the wireless access technology in use, and retrieves data from heterogeneous sensors by means of a common API regardless of the WSN implementation. The rationale of the sensing applications is presented in the Listing 4.1.

[1]ZeroMQ Project Website, http://zeromq.org

```
let type = "http://wots.unibo.it/labWireless/testbed"
let THINGS = []
//get Thing Descriptions from the discovery service
for(var t in discovery.discoverByType(type)) {
    //Consume things
   let thing = await consumer.consumeThing(t);
   //Set http as protocol required
   thing.getClients().set('http', http_client);
   THINGS.push(thing)
}
for(var i = 0; i < lambda; i++) {
    //invoke the getData action for collecting data
    let thing = THINGS[i%THINGS.length];
    var res = await thing.actions['getData'].invoke();
}
```

Listing 4.1 Example code for discovering and invoking actions on Things.

4.1.3 Mashup sensing Policies

Using W3C WoT in a heterogeneous environment - like the one presented in the previous Section - has the concrete advantage of hiding all the complexity for handling different technologies and protocols. This translates into the fact that, for instance, it is easier to implement and manage applications that benefit from heterogeneous sensing sources, without explicitly addressing different kinds of operations. For this reason, we demonstrate the possibility to decouple the mash-up policies from the network functionalities, and we evaluate the overhead introduced by the WoT approach. We implemented multiple mash-up sensing policies, and we tested the functionality of switching among them in a seamless way in Section 4.1.4. To this purpose, let D be the set of available devices, and $W(d_i)$, $\forall d_i \in D$, be the function describing the WSN type. In the testbed, $W : D \rightarrow \{WiFi, BLE, Zigbee\}$. We can assume the time to be divided into discrete time slot, i.e. $T = \{t_0, t_1, ...\}$, corresponding to sensing events when the mash-up application is issuing `getData` command toward a selected subset of the available devices. Let $t_{interval}$ be the temporal interval between two measurements, i.e. the time difference between t_{i+1} and t_i, assumed constant. Moreover, let $\kappa : D \times T \rightarrow \{0,1\}$ the function indicating whether device d_i is active, i.e. it is used at time slot t_j (in this case, $\kappa(d_i, t_j) = 1$, otherwise $\kappa(d_i, t_j) = 0$). All sensing policies share a common rationale, i.e.: they keep the area covered higher than a predefined threshold, while maximizing a performance index I. In our case, the area coverage is expressed in terms of number of active devices (M) at each time slot. More formally, all policies address the optimization problem

formally defined below:

$$\begin{aligned} \text{Goal} &: \text{Maximize } I \\ \text{Constraint} &: \sum_{d_i \in D} \kappa(d_i, t_j) = M, \forall t_j \in T \end{aligned} \tag{4.1}$$

The performance I can vary according to the sensing policy in use. We implemented and tested four different metrics:

- Static *Energy-aware* policy (P_0). The mash-up application selects the M active devices at each time slot according to a pure round-robin scheme, in order to discharge them at the same rate.
- Dynamic *Delay-aware* policy (P_1). The mash-up application takes into account the average delay required to issue a `getData` command and to receive the corresponding reply message. The M devices with the lowest Round Trip Time (RTT) are selected at each time slot.
- Dynamic *PDR-aware* policy (P_2). The mash-up application takes into account the communication reliability of each sensor expressed in terms of average Packet Delivery Ratio (PDR), i.e. the ratio of received replies over the total number of `getData` requests sent toward each d_i. Specifically, the M devices with the highest PDR values are selected at each time slot.
- Dynamic *Delay-PDR-aware* policy (P_3). The mash-up application takes into account both the delay and the PDR, as better explained in the following.

Excluding P_0, all the other policies compute the M sensors to query at each time slot based on the current traffic loads and network conditions. For this reason, we employ a dynamic, learning-based scheme based on the Reinforcement Learning (*RL*) framework[106]. In brief, this latter refers to a class of machine learning algorithms where an agent learns over time the optimal sequence of actions needed to perform a task, by dynamically interacting with the environment and by receiving a numeric reward at each interaction. More formally, the *RL* framework can be represented as a Markov Discrete Process (MDP) $< S, A, R, TR >$ where: S is the set of States, A is the set of Actions, $R : \{S, A\} \rightarrow \mathbb{R}$ is the Reward function, expressing a numeric reward received by the agent when executing action $a_j \in A$ in state $s_i \in S$, and $TR : \{S, A\} \rightarrow S$ is the transition function, expressing the next state s_j after performing action a_j from state s_i (a deterministic environment is assumed). The goal of the *RL* agent is hence to determine the optimal policy function $\tau : S \rightarrow A$ that indicates the optimal action to execute

at each state, so that the long-term reward is maximized. In this modeling, we omit the state function S, while the list of action A coincides with the list of devices D. The immediate reward $R(d_i)$ is computed when issuing a `getData` command on sensor d_i, according to the policy in use:

- P_1: this is the RTT for each `getData` command. Only successful requests (i.e. reply messages are received) are considered.
- P_2: this is a positive value (+1) if the `getData` is successful, 0 otherwise.
- P_3: similarly to P_1, however a penalty equal to $t_{timeout}$ is applied in case no reply is sent back after a timeout.

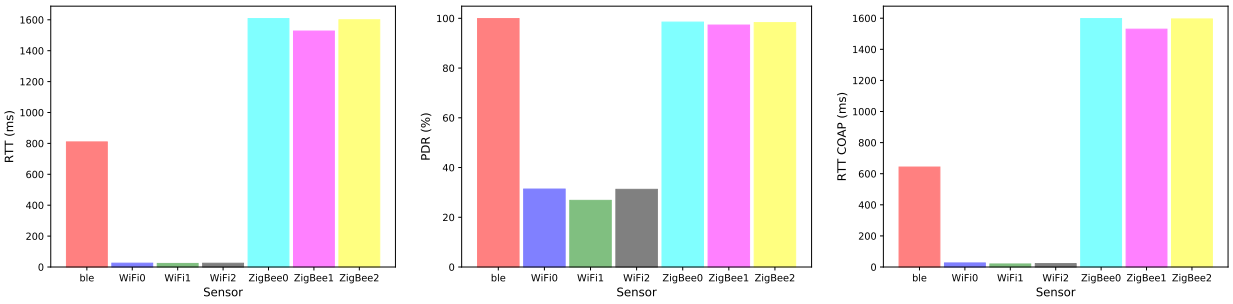
Each time a `getData` is issued on d_i, and the immediate reward $R(d_i)$ is computed, the Q-value entry is also updated at time slot t for d_i as follows:

$$Q_t(d_i) = Q_{t-1}(d_i) + \alpha \cdot (R(d_i) - Q_{t-1}(d_i)) \tag{4.2}$$

where α is a learning rate, set equal to 0.7 in the experiments. Balancing the exploration and exploitation issue is a crucial issue in dynamic environments [106]. For this reason, we consider an ε-greedy exploration scheme, i.e.: each time a `getData` is executed, the policy selects with probability $1-\varepsilon$ the sensor with the k-th highest Q-value, and it performs a random selection over D otherwise (avoiding duplicates). We repeat the ε-greedy selection M times at each time slot, since all policies need to guarantee an M-coverage of the scenario (in other words, the k above varies between 0 and $M-1$). The ε parameter is progressively discounted at each time slot, i.e. $\varepsilon_t = \varepsilon_{t-1} \cdot \psi$, with $0 < \psi < 1$, in order to reduce the exploration over time. At the same time, the ε parameter cannot decrease below a minimal threshold (ε_{min}), i.e. a default exploration rate is kept anyway in order to detect any possible change in the scenario, and to adapt the system policy accordingly. In the testbed ε=0.8, ψ=0.97, ε_{min}=0.1.

4.1.4 Evaluation

In this Section, we report a subset of experimental results collected through the WoT testbed described above. The experimental analysis is divided into three stages: (*i*) first, we characterize the overall performance of different WSNs and sensors; (*ii*) second, we evaluate the four different mash-up policies of Section 4.1.3; (*ii*) finally, we demonstrate the possibility of dynamic mash-up policy replacement and quantify the overhead introduced by the W3C WoT architecture. Figures 4.3(a), 4.3(b) and 4.3(c) refer to the first analysis. Specifically,

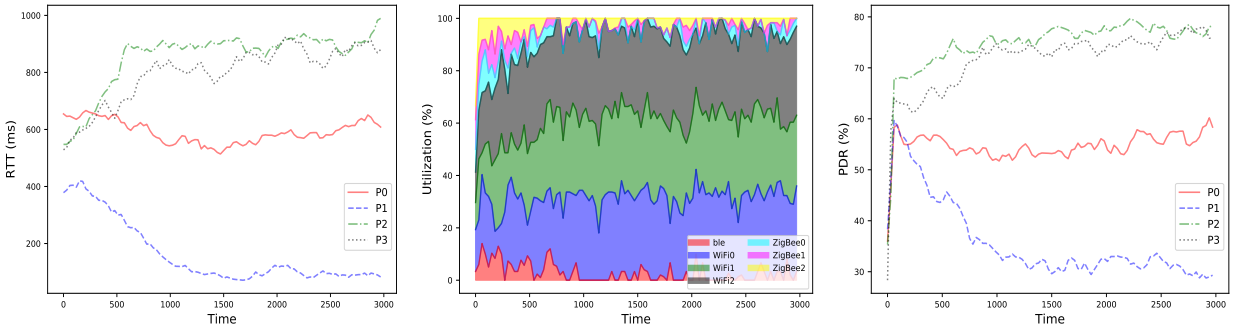


(a) (b) (c)

Fig. 4.3 The average per-device RTT and PDR is shown in Figures 4.3(a) and Figure 4.3(b), respectively. The per-device RTT for the CoAP protocol is shown in Figure 4.3(c).

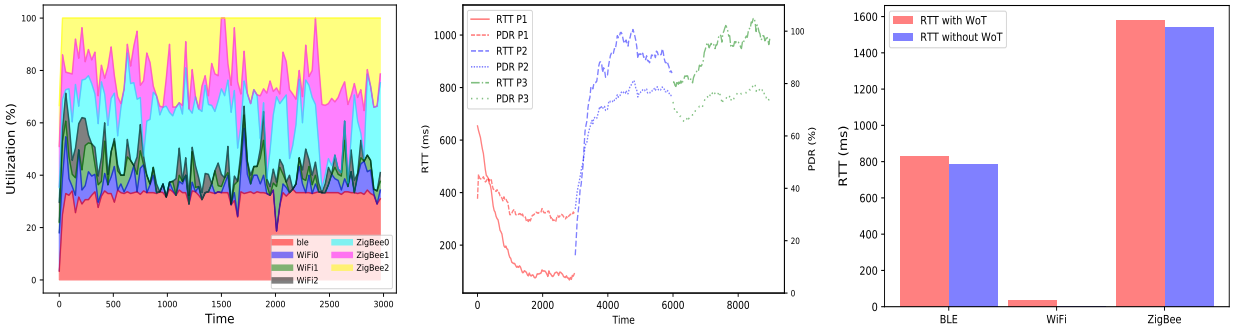
Figure 4.3(a) and 4.3(b) show respectively the average RTT and PDR for each device and WSN type, when the HTTP protocol is used to interact with each Web Thing. It is easy to notice that Wi-Fi devices are producing the lowest RTT values. The PDR original results demonstrated that the Wi-Fi WSN is also the most reliable technology. However, in order to differentiate the mash-up policies, we introduced a probabilistic packet filter on the Wi-Fi Servient, discarding the sensor data messages with a loss rate equal to 70% to emulate a congested access point. As a result, comparing Figures 4.3(a) and 4.3(b), we can notice that the sets of M=3 nodes maximizing the RTT or the PDR depends on the selected performance index. Finally, Figure 4.3(c) shows the per-device RTT when the CoAP protocol is used for data gathering. Only minimal differences can be noticed compared to the HTTP case (Figure 4.3(a)).

In Figures 4.4(a)-4.5(a), the performance of different mash-up policies is evaluated. Figure 4.4(a) shows the RTT values of P_0, P_1, P_2, P_3 algorithms over time slots; as expected, P_1 produces the lowest delay since it takes into account the per-packet RTT as an immediate reward. Also, we can appreciate the learning phases of P_1: the RTT is high during the exploration phase and it is progressively reduced when increasing the amount of exploitation. After time slot 1000, the RL algorithm has discovered the optimal set of sensors, however it keeps performing random actions for continuous, minimal exploration. This justifies the jagged shape of the plot. In Figure 4.4(b) the per-device ratio of utilization over time for the policy P_1 is depicted. While during exploration all the devices are equally used, after time slot 1000 the mash-up policy is mostly exploiting the three Wi-Fi devices since -in accordance with Figure 4.3(a)- they are associated to the lowest RTT values. Figure 4.4(c) compares the policies in terms of PDR. Here, the optimal policy is P_2; from Figure 4.5(a) we can notice that, after the exploration phase, the three Zigbee devices are maximally used, hence conversely to Figure 4.4(b) but again in accordance with Figure 4.3(b). We tested



(a) (b) (c)

Fig. 4.4 The RTT and PDR values for the four mash-up policies are shown in Figures 4.4(a) and 4.4(c). The device utilization ratio for the P_1 policy is shown in Figure 4.4(b).



(a) (b) (c)

Fig. 4.5 The device utilization ratio for the P_2 policy is shown in Figure 4.5(a). The RTT and PDR values when replacing the active policy at run-time are shown in Figure 4.5(b). The RTT when enabling/disabling the WoT approach is shown in Figure 4.5(c).

the dynamic policy replacement in Figure 4.5(b); i.e. from time slot 1 to 3000, policy P_1 is used (delay minimization), then P_2 from 3001 to 6000 (PDR maximization), finally P_3 (delay-PDR trade-off) starts from instant 6001. We remark that the policy replacement is simply implemented as the shut-down of a Javascript process and the execution of a new one, thanks to the abstraction provided by the W3C WoT architecture; no hardware or software re-configuration of the WSNs is required. Finally, we evaluate in Figure 4.5(c) the overhead introduced by the W3C WoT deployment, and specifically by the WoT servient: to this aim, the RTT required to perform a sensor request directly at the System API level is computed. We can notice that most of the overhead is due to the channel access and the processing at the firmware level, while the overhead introduced by the Servient and by the additional communication with the Web Thing is negligible.

4.2 Structural Health Monitoring (SHM)

4.2.1 SHM Scenario

Structural Health Monitoring (SHM) identifies the general process of assessing the condition of aging structures and infrastructures by checking their current integrity status. In addition, SHM data can be used to feed prognostic schemes to predict the remaining useful life of structures/infrastructures [113]. Thanks to its potential to reduce the vulnerability of strategic structures and increase the preservation of architectural heritage, and to some extent to save human lives, SHM can play a crucial role in modern smart cities [114].

Several research studies recently demonstrated the possibility to improve both the efficiency and the reliability of SHM systems through the adoption of Internet of Things (IoT) technologies for sensors data management and analytics [115]. Indeed, it has been proved that big-data techniques can support distributed storage and real-time processing of SHM deployments, even in presence of high sampling rates and long-lasting measurements [116]. Hence, given the huge amount of collected datasets, Machine Learning (ML) and Artificial Intelligence (AI) strategies provide useful tools for the condition assessment and/or for structural damage identification. For example, such ML/AI-driven approaches demonstrated to be particularly effective in presence of image-based inspection [117]. Thus, to ensure real-time and over-time functionalities, a complete SHM system must jointly optimize all the different architectural layers. While several ad-hoc software deployments have been reported in the literature about SHM systems (e.g. [114][118][119]), only a few propose complete and versatile IoT platforms for sensor-to-cloud data collection and analytics [120].

In this Section, we address two main requirements of IoT-based SHM scenarios, namely the system *scalability* and the *interoperability*. The first issue not only relates to the need for managing large data volumes, but also implies the optimal balancing of the computation among the available resources of the network in a cloud-to-edge continuum. On the other hand, the necessity for interoperability is often determined by the heterogeneity of sensor devices that may be installed on the structure [121], a solution that is usually preferred to effectively increase the robustness of the SHM systems [122]. Despite this, sensor devices can actually be produced by different manufacturers, be mapped on different data protocols (e.g. MQTT, MODBUS, etc), or even exploit different sensing technologies (e.g., accelerometers, hygrometers, strain-gauges). Therefore, such a fragmentation of devices, communication protocols and data formats is a primary reason of high computational and installation costs, that may unavoidably hamper the full-scale applicability of present IoT implementations.

This section presents the MODRON platform, an extension/customization of the WoT Store framework (previously presented in section 3.1) for SHM scenarios. Specifically, the

framework addresses sensor data acquisition from the monitoring layer, sensor management, distributed data storage, data visualization, and analytics. The MODRON software platform - installed on a remote cloud infrastructure - is a novel and versatile software framework for sensor-to-cloud data acquisition and management in SHM scenarios. MODRON relies on the WoT W3C standard in order to support heterogeneous sensor environments: this is achieved by means of a layered architecture, with an edge layer composed of WTs exposing the sensor data, and a remote cloud layer consuming the WTs and accessing their affordances. Beside interoperability, and like WoT Store, the platform is designed in order to be (*i*) *extensible*, thanks to the modular design which allows, for instance, to support new classes of sensor devices by designing their corresponding WTs and (*ii*) *highly adaptive*, since the cloud platform is able to update its functionalities (e.g. the GUI) according to the TDs of the active WTs, and hence it abstracts as much as possible from the sensing technologies in use. We present the MODRON platform in three stages:

- First, we expand the illustration of the WoT Store architecture 3.1, by focusing on edge software components and cloud nodes. We discuss how sensing data acquisition and storage have been handled with the WoT approach, and the overall data modeling of the SHM scenario.
- Second, we discuss the current software implementation, which supports two classes of sensor devices, namely MEMS accelerometers and piezoelectric sensors. Beside the enabling technologies, the TD of the MEMS accelerometer is illustrated.
- Third, we present a preliminary SHM testbed (related to the BRIC MAC4PRO [41] project founded by INAIL) involving the utilization of the MODRON platform for sensor data acquisition and storage. The testbed involves the real-time monitoring of a metallic structure located at the University of Bologna, instrumented with multiple sensor networks of low-power and low-cost inertial devices.

4.2.2 Architecture

The layered architecture proposed for the MODRON platform is depicted in Figure 4.6. The monitoring layer includes the sensor devices and the network infrastructure installed on the physical structure and it is in charge of generating real-time data streams related to vibration and acoustic events. The edge layer defines the hardware/software components (located on the structure or in its close proximity) which implement the first steps of the SHM data pipeline, i.e. data acquisition, cleaning, filtering, and pre-processing. The data management layer is constituted by the software platform addressing the storage, aggregation, and visualization

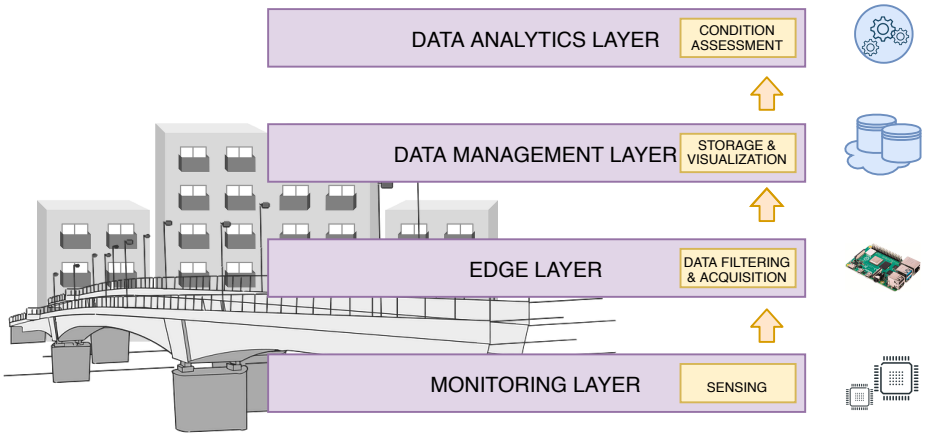


Fig. 4.6 Proposed layered SHM architecture.

of the acquired data, and the remote control of the sensor devices. Finally, the analytics give meaning to the collected data by techniques of condition assessment and damage detection/localization/prediction. The focus of the contribution deals with the software platform deployed on the edge/cloud levels. The overall software architecture (depicted in Figure 4.7) is designed to abstract from the sensing technology in use, and to support interoperability across heterogeneous data sources of the monitoring layer. At the same time, the current implementation and system validation rely on the MEMS accelerometer devices, whose characteristics are detailed in Section 4.2.4.

Edge Layer

The software layer running on the edge device is in charge of: (*i*) acquiring data from the monitoring layer, by supporting the most common IoT and messaging protocols (e.g. MODBUS, OPC UA, MQTT, etc); (*ii*) making sensor values and devices' status information available to the data management layer, by hiding the heterogeneity of acquisition protocols/hardware while supporting the remote control and configuration. The requisites above have been addressed by resorting to a W3C WoT approach, hence developing a collection of W3C WTs on the edge node. More specifically, we distinguished among two types of WTs: *Sensor* WTs and *Digital Twin* WTs. In the first case, each WT corresponds exactly to one sensor device (e.g. an accelerometer), and allows to tune its property or to acquire new data, based on a publish-subscribe or request-response paradigm; the communication with the monitoring layer is handled by the System API of the WT [24]. More in detail, we associated three *Sensor* WTs to each device: the *Observable Sensor* WT, the *Controllable Sensor* WT, and the *Debug* WT, respectively. An *Observable Sensor* WT exposes only properties and actions allowing to read sensors' measurements but not to modify the sensor configuration. Con-

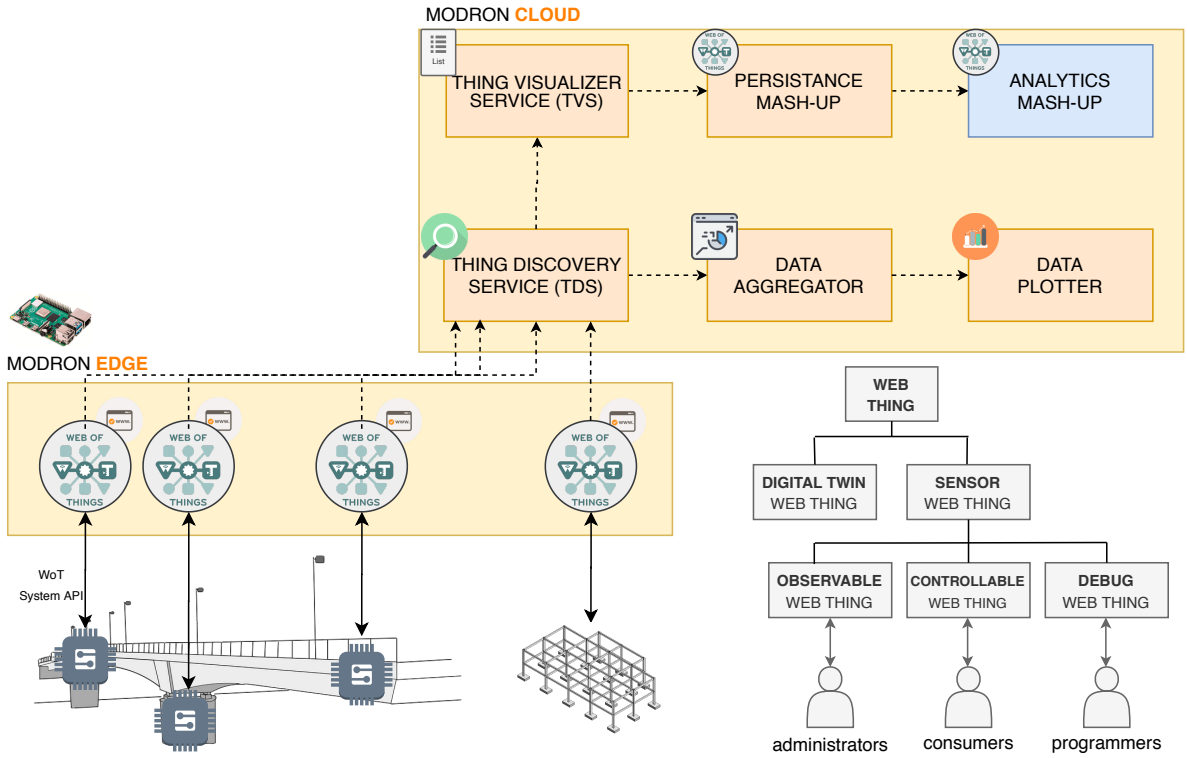


Fig. 4.7 The MODRON software platform with the edge/cloud components.

versely, a *Controllable Sensor* WT also supports the remote device configuration, including the possibility to upload at run-time a new Behaviour of the WT. A *Debug* WT offers specific Affordances for the diagnostic and self-testing functionalities, which can be useful during the installation and calibration of the SHM system. The hierarchy of *Sensor* WTs is motivated by security reasons: in a typical SHM scenario, multiple categories of end-users might access the software platform, with different roles and duties (e.g. system administration, data consuming, maintenance and testing). Effective access control policies can be devised through the mapping of the user profiles with the instances of the *Sensor* WTs, as depicted in Figure 4.7. Finally, a *Digital Twin* WT models a virtual entity derived from the aggregation of multiple *Sensor* WTs. For instance, we consider a Digital Twin of the monitored structure as a whole entity (e.g. the bridge); the WT provides new descriptive properties (e.g. the GPS coordinates and the 3D model) as well as the list of the installed *Sensor* WTs. The Sensor Area Network (SAN) [123], namely a single daisy-chain connection of sensors, is also modeled as a *Digital Twin* WT.

Data Management and Analytics Layers

In MODRON, we exploit the Thing Manager module for the WT discovery, while at the same time we deploy ad-hoc Mash-up Applications (MAs) for data gathering and storage. Similarly, the Data Manager presented in section 3.1.2 has been customized to process and visualize SHM sensor data. The internal architecture of the MODRON cloud platform includes the six components of Figure 4.7, i.e. the Thing Discovery Service (TDS), the Thing Visualizer Service (TVS), the Persistence and Analytics MAs and the SHM Data Aggregator and Plotter. The TDS serves as WoT directory: each time a new WT is spawned on the edge server, its related TD is registered onto the TDS. The TVS allows for the visualization of the list of WTs currently registered to the TDS. Moreover, it allows end-users to interact with each registered WT or with a filtered subset of them. This operation can be performed by parsing the corresponding TDs and dynamically generating an ad-hoc Web GUI, through which it is possible to monitor the state properties, click and execute actions (passing the needed parameters if requested), or receive notifications about the occurred events. As a result, an end-user with a Consumer profile can visualize the status and the last recorded data of an *Observable Sensor* WT; similarly, an end-user with Administrator profile can start/stop or update an *Observable Sensor* WT with a button click. We remark the extensibility of this approach, since the cloud platform is agnostic with respect to the available WTs: in case a new sensor device is added to the SHM system, no changes are required to the TVS since the latter parses the TD and dynamically creates the corresponding Web interface. The Persistence MAs are in charge of gathering data from the Observable WTs; to this purpose, they consume the WTs registered in TDS and query them at fixed intervals (for request-response interactions) or register the events produced by the WTs (for publish-subscribe interactions). The sensing data are then stored in a distributed database, entailing data consistency and replication capabilities (see Section 4.2.3 for further details). Again, we remark the generality of the Persistence MA, which is able to extract the data structure from the Affordances property of the WT, and based on them to build the corresponding tables (one for each sensor) and to expose the corresponding CRUD operations. The Analytics MAs (not implemented yet) process the stored data and provide the signal processing techniques for structural integrity evaluation. Finally, a custom Data Manager for SHM systems has been deployed and organized in two sub-modules: (*i*) a Data Aggregator, allowing to select, merge or extract features from the stored time-series, and (*ii*) a Data Plotter supporting the visualization of the output of the Data Aggregator, and/or their exportation on files.

4.2.3 Implementation and Validation

The MODRON architecture has been validated by implementing its components and deploying them in a real use case scenario, as also explained in the next Section. The goal of this operation is to provide evidence of the fact that MODRON hides all the differences - in terms of protocols and technologies - of the sensors involved, providing a transparent and uniform management of heterogeneous resources. We validated the MODRON platform by implementing its architecture and deploying whose goal is to show

The MODRON platform implementation is based on state-of-the-art front-end and back-end Web technologies. More in detail:

Edge Layer. The WoT run-time environment is constituted by the `node-wot` Servient [25] for JS language. We implemented a basic W3C WT of a `SANSensor`, which later was extended in order to support two classes of sensor devices: tri-axial MEMS accelerometers and piezoelectric sensors. In both cases, the communication with the monitoring layer (System API) is mapped over a USB Serial port. The TD is semantically annotated by using the Semantic Sensor Network Ontology (SOSA) [20]. Table 4.2 reports the main Affordances of a MEMS accelerometer device.

Data Management Layer. The cloud platform uses a wide number of JS libraries/tools and database systems. The back-end functionalities are implemented through `Node.js`[2] and related libraries, including, among the others: `LoopBack`, `SocketIO`, and `Nest.js`. The TVS exposes APIs for WT registering, searching and filtering based on `GraphQL`[3], an open-source data query and manipulation language. Also, the framework includes a combination of database technologies, such as: (*i*) texttttBlazegraph[4], a triplestore used to save the application metadata and the TD; (*ii*) `Apache Cassandra`[5], a NOSQL database used to store the sensing data gathered by the Persistence MA; (*ii*) `Redis`[6], an in-memory data structure store used as a temporary cache of sensor data. Specifically, the `Redis` tool is used to optimize the performance of high-frequency sensing applications: the sensor data gathered by the Persistence MA when consuming the `Observable Sensor` WTs are immediately saved in `Redis` so as to be immediately available to the upper-layer services (e.g. the Analytics MA), and then periodically transferred to the `Cassandra` database management system. The latter has been configured for distributed operations: the cluster is composed of

[2]https://nodejs.org/
[3]https://graphql.org
[4]https://blazegraph.com
[5]https://cassandra.apache.org
[6]https://redis.io

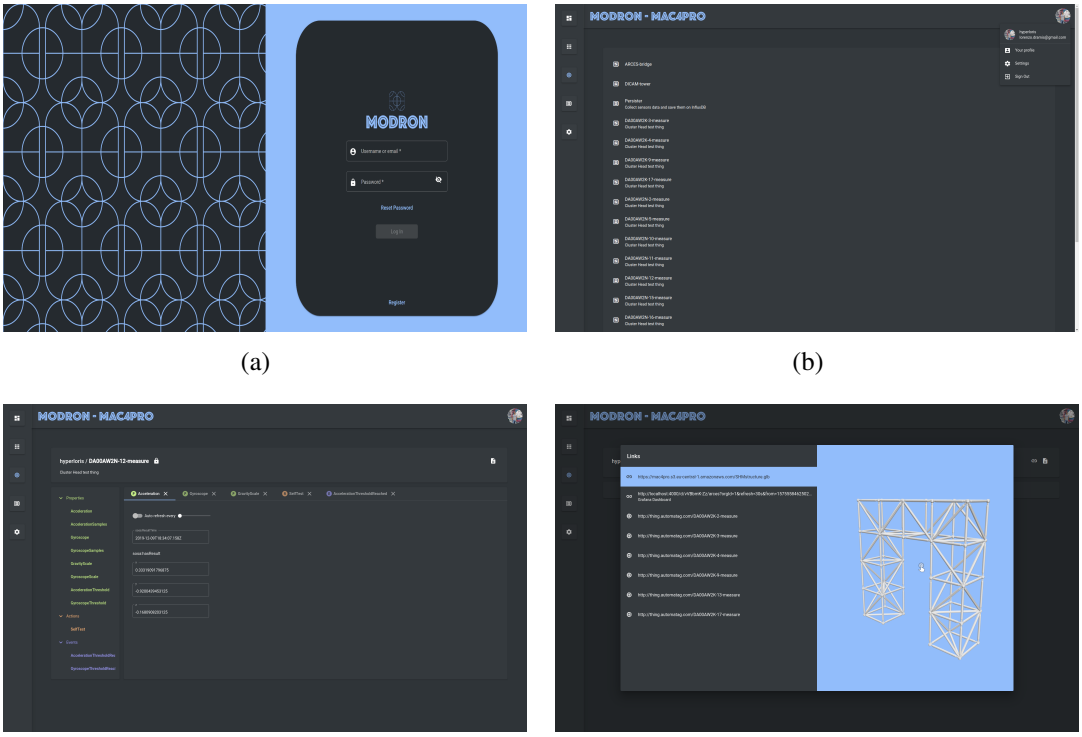
Name	Type	Description
AccelerometerSample	Property	Current accelerometer 3-axial values.
GyroscopeSample	Property	Current gyroscope 3-axial values.
AccThresold	Property	Accelerometer threshold value.
Start	Action	Start the data acquisition.
Stop	Action	Stop the data acquisition.
OnOverThreshold	Event	Even triggered when the accelerometer module is greater than the AccThreshold.

Table 4.2 List of Affordances of a Controllable Sensor WT.

three instances, and employs a distributed data balancer (the `Murmur3Partitioner` policy) and a basic replication strategy with a factor equal to the number of available instances.

4.2.4 Deploy

The proposed MODRON platform has been employed within the INAIL 2018 MAC4PRO [41] project for the monitoring and predictive maintenance of manifold industrial plants and civil structures. In particular, the metallic truss structure in Figure 4.9, which is located at the research laboratory of the Department of Civil Engineering of the University of Bologna, was considered as a preliminary testbed. The data measurement layer consists of light-weight, low-cost and small-footprint MEMS accelerometers, each of them featuring a 6 Degree-of-freedom system in package LSM6DSL inertial measurement unit (IMU) providing both tri-axial accelerations and as many angular velocities. The sensor device integrates an ST Microelectronics STM32F303 32bit, 3.3 V low-power microcontroller unit (MCU) embedding Digital Signal Processing (DSP) functionalities and a floating-point unit (FPU) [123] compliant with basic data pre-processing. As far as the network topology is concerned, the sampling positions in Figure 4.9 were chosen, which are arranged in two distinct chains of six accelerometers (identified with red and green colors, respectively) bolted in correspondence of the junction elements. Sensors are connected in a daisy-chain fashion by means of a Sensor Area Network (SAN) bus, which leverages data-over-power communication capabilities on the basis of the EIA RS-485 standard. Data are transmitted sequentially, in packets, by exploiting a proprietary lossless encoding technique; for this reason, a Gateway (GW) edge device has been purposely developed to connect the SAN bus with external communication protocols [124]. To maximize redundancy and minimize the data losses, two logical SANs are configured and installed on the structure: the GWs are in turn connected (via USB cables) to a Raspberry PI 3b+ device, which serves as edge node for the architecture of Figure 4.6.



(a) (b)

(c) (d)

Fig. 4.8 Screenshots of the MODRON platform: the landing page (Figure 4.8(a)), the list of available WTs (Figure 4.8(b)), the rendering of a *Controllable Sensor* WT (Figure 4.8(c)) and of a *Digital Twin* WT (Figure 4.8(d)

The Raspberry PI hosts the WoT Servient framework and hence the *Sensor* and *Digital Twin* WTs described in Section 4.2.2. We highlight that -through the WoT layer- the MODRON framework can access and control each single accelerometer device, albeit they are not directly connected to the Internet. Indeed, the WT provides a Web interface for each sensor both by communicating with the GW according to the target SAN protocol and also by virtualizing the sensor device. Figures 4.8(b), 4.8(c), 4.8(d) and 4.11 show four screenshots of the MODRON GUI. More specifically, Figures 4.8(b) and 4.8(c) depict some operations of the TVS supporting the interaction with the registered WTs. In 4.8(b), the user (with Administration privileges) is listing the set of WTs in the system: by clicking on one *Controllable Sensor*, the corresponding TD is rendered as in 4.8(c), allowing the users to check the last property values (e.g. the current acceleration value), or to set some writable properties (e.g. the alarm threshold). Similarly, by clicking on the *Digital Twin* WT of the metallic structure, its 3D model is displayed (Figure 4.8(d)). Finally, Figure 4.11 shows the GUI of the Data Plotter; the user can create the data query to filter the sensors and the time interval of interest by filling the proper Web forms, which are dynamically built according to the WT TDs. The query is executed by the Data Aggregator, returning the selected time-series which are then displayed by the Data Plotter. The user can also export the acquired data to different file formats.

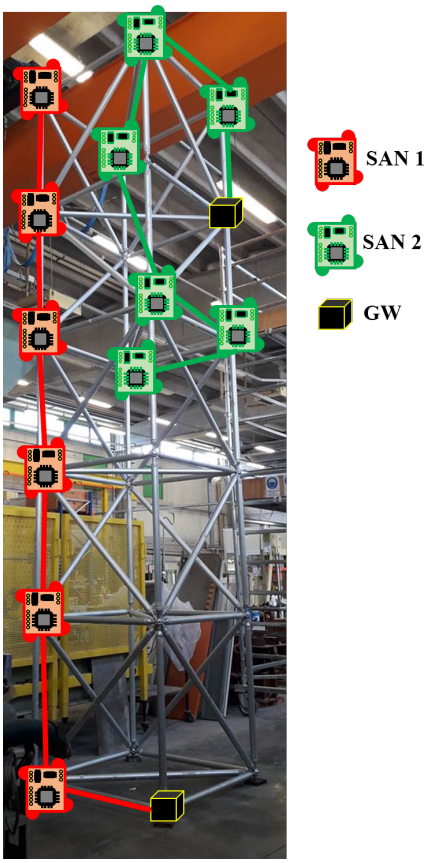


Fig. 4.9 The monitored metallic truss structure and relative sensor equipment: two logical SANs (red and green colors, respectively) are connected to as many GW interface devices (black boxes).

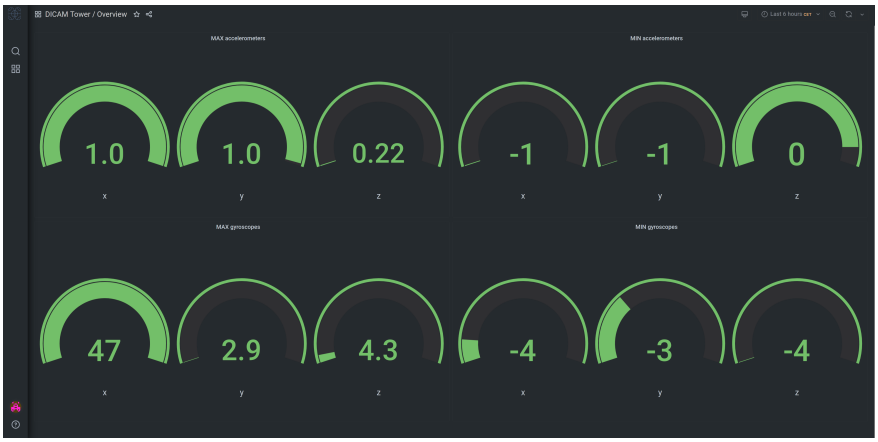


Fig. 4.10 The MODRON Data Manager allowing the selection, visualization, and exportation of sensor time-series.

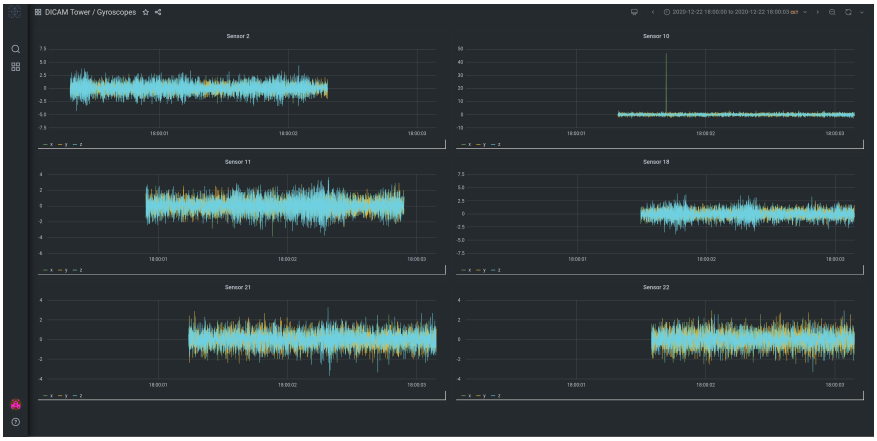


Fig. 4.11 The MODRON Data Manager detail of sensor data.

Chapter 5

Improvements for WoT

The W3C standard is quite new and still under definition. Because of its recent appearance, there are some scenarios of possible WoT applications that need to be investigated more deeply. This kind of process not only highlights the great potential of the WoT, but sometimes it reveals possible lacks in this kind of solution. In our effort to augment and improve the WoT Store, we often had to face some obstacles - both in the standard and in the official software stack implementation - that encouraged us to advance some proposals that can be useful for solving those criticisms that could come to the surface once new kinds of studies are conducted. In particular, in this Chapter we present two different studies: the first work is intended to explore the suitability of the Web of Things in the context of Industry 4.0, specifically where strict QoS requirements must be guaranteed. More in detail, the objective is to define and design an automatic process for configuring the Time Sensitive Networks, according to the QoS resulting from merging both the QoS requirements of the Mashup Application and the QoS capabilities of the Web Things. For this purpose, it was necessary to bring the TSN technologies as well as the OPC UA standard into the W3C WoT ecosystem. During this process, we encountered some lacks in the standard that drove us to propose accurate solutions to have the standard include also this kind of technologies. The other work presented in this Chapter is about migrating WoT services: the main goal is the ability to dynamically migrate services from the edge to the cloud and vice-versa in order to guarantee the edge-to-cloud continuum. Clearly, this research has been made from the WoT perspective, revealing that the task appears quite complicated without any modifications to the official W3C WoT implementation. Hence, also in this case, new proposals are advanced in order to augment the *node-wot* software stack for W3C WoT.

5.1 Industrial WoT on the Edge

5.1.1 Scenario

Most quality of service (QoS) solutions proposed for the Internet of Things (IoT) rely on service and/or path selection strategies or prioritized message handling, but cannot provide guarantees [125]. Industrial IoT, however, where IT converges with OT (Operational Technology), requires strict QoS guarantees to fulfill the requirements of real-time applications: messages have to arrive at deterministic times with low jitter and virtually zero loss. Here, Time-Sensitive Networking (TSN) is emerging as a common standard for industrial automation (but also other domains such as automotive or the fronthaul of the mobile network). However, given its complex Layer-2 ecosystem and classic OT approach, deterministic networking forms its own silo with specialized protocols such as LRP/RAP, NETCONF, and its YANG models.

The W3C Web of Things (WoT) (see section 2.5.5) aims at breaking up such silos to enable cross-vendor, and cross-domain IoT applications. It also explicitly lists industrial use cases as its target. However, so far W3C WoT has not considered bindings toward the required field-level communication mechanisms with hard real-time QoS.

Industrial IoT applications are also in the scope of W3C WoT [126]. Here, TD is a good candidate for the realization of the Asset Administration Shell of the German Industry 4.0 initiative[127]. TDs can be enriched with arbitrary information, also over time, and different views can be serialized dynamically depending on the use case from a holistic knowledge graph (e.g., user interaction, maintenance, digital forensics).

QoS, however, has so far not been investigated by the WoT community. In this Section, we consider QoS parameters used for industrial automation to give guarantees for certain traffic types:

Zero congestion loss: Control loops managing heavy and fast-moving machine parts must rely on guaranteed message delivery. As physically broken channels are rare in wired environments (e.g., a severed cable), the main reason for message loss is congestion in the network: when the bandwidth of an outgoing (egress) link is exceeded, packets have to be buffered and will be dropped if the congestion lasts too long, as the buffers are limited. Tight control over the resource allocation in the network must ensure that critical traffic will never encounter congestion. This QoS requirement is usually combined with the following two.

Deadline: Control loops also require that input data arrives by a certain time, so that the output can be computed, sent, and delivered within one cycle of the loop. Hence, networks must be able to guarantee that such messages always arrive by the given deadline. For this,

applications must also send their messages in time, and hence are synchronized to a common time. This is also called the isochronous traffic type.

Latency: Slightly less demanding applications only require a guarantee for the latency between sending a message and receiving it at the destination. Usually, these messages are periodic and the required latency corresponds to this application cycle time, but it can also be a standing reservation for critical event-based messages. This is also called the cyclic traffic type.

Bandwidth: For other applications it is enough, when they are allowed to send and receive messages within a given period of time, i.e., their transmission is not starving completely. Examples are discovery (e.g., LLDP), configuration (e.g., NETCONF), parameterization (i.e., application configuration), and any kind of diagnostics.

Best effort: The IT part in the IT/OT convergence usually has no QoS requirements (e.g., for PCs and printers connected to the shop floor network). However, just like in offices, messages should come through eventually. Hence, a portion of the network resources usually remains unallocated and available for this traffic type.

Fieldbuses are industrial computer network protocols that provide such QoS guarantees. Modern examples are PROFINET, EtherCAT, PowerLink, or EtherNet/IP. They are maintained in quasi-open standardization groups, but are mainly driven by their creator companies, and hence form silos up to the engineering tools. Controllers usually support multiple fieldbuses, so that they can integrate into existing factory environments. In 2018, for the first time the majority of fieldbuses in use were Ethernet-based with a share of 59% and growing in 2019.[1] Wireless such as Wi-Fi 6 or 6TiSCH currently plays a minor role in IIoT with only 6%, although a lot is promised with 5G, not fully considering the cost and complexity of running a mobile network core for a local installation.

OPC UA [128] is a relevant IIoT technology, as it provides a whole suite of specifications covering communication protocols, security, functional safety, and most importantly information modeling of industrial systems. Thus, it has become popular at the management level of industrial systems. With its Field-Level Communications (FLC) initiative [129], the OPC Foundation is now pushing downward to the device level and could establish a converged alternative to the plethora of fieldbuses. FLC uses OPC UA PubSub as underlying communication framework, as it is light enough for constrained devices and can leverage real-time QoS mechanisms through its UDP and Layer 2 transports. For the latter, FLC aims at using the IEC/IEEE 60802 Industrial Automation Profile for TSN[2]. Yet also other

[1] https://www.hms-networks.com/news-and-insights/news-from-hms/2019/05/07/industrial-network-market-shares-2019-according-to-hms

[2] https://1.ieee802.org/tsn/iec-ieee-60802/

fieldbuses are converging to 60802, as it will enable cheaper silicon and common off-the-shelf hardware.

5.1.2 Time-Sensitive Networking

The origin of Ethernet QoS lies in the IEEE 802.1p Task Group, which introduced traffic classes as QoS technique based on a 3-bit Priority Code point (PCP) and a Strict Priority selector for which Ethernet frame a *bridge* (network switch) should send next on a given bridge egress port. Both concepts were ultimately incorporated in the IEEE 802.1Q standard [130] using the VLAN tag/header.

The *Audio/Video Bridging (AVB)* Task Group added 802.1Qav[3] defining the Credit-Based Shaper (CBS), which can prevent traffic bursts from congesting the network, as well as the 802.1AS standard for isochronous time synchronization[4]. However, AVB only provided two traffic classes with CBS, as it was intended for multimedia streaming applications and not control data traffic. [131]

The *Time-Sensitive Networking (TSN)* Task Group[5] evolved Ethernet QoS further to achieve delivery of data with bounded low latency, low delay variation (jitter), and low loss. TSN encompasses over 30 standards and amendments, many of them already incorporated into the ever growing 802.1Q standard. This study only highlights a few specifications to explain the QoS concept for W3C WoT:

802.1Qbv – Time-Aware Shaper (TAS) uses a cyclic schedule to control a transmission gate for each queue of a bridge egress port as seen in Figure 5.1. This schedule is executed in hardware and defined by the so-called Gate Control List (GCL), which contains entries with a specific time within the cycle and an 8-bit vector for the gate states (open/closed). With this, one can define TAS windows in which a certain traffic class can be given exclusive usage of the network for some time (cf. virtualization or slicing). In extreme cases, such TAS windows are defined for a single (periodic) frame and are adjusted for each hop, so that the frame can pass the network without any queuing delay along its path.

802.1Qbu/.3br – Frame Preemption allows the ongoing transmission of a frame to be preempted by a frame marked as express traffic. This can allow especially short control message frames to overtake potentially large frames currently blocking the egress port, and thereby lower the latency. Preemption can also help to reduce guard band sizes, which are required before exclusive TAS windows to ensure the port is idle and ready to send.

[3]The lowercase letters indicate amendment for the uppercase standard
[4]Updated by the TSN Task Group to cover multiple TSN domains as 802.1AS-2020
[5]https://1.ieee802.org/tsn/

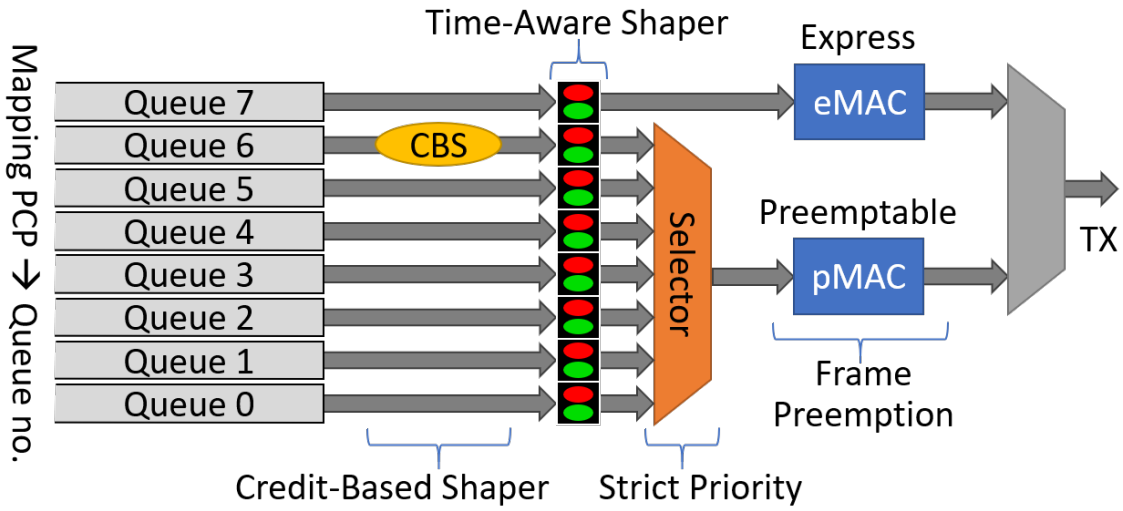


Fig. 5.1 Different TSN concepts at the egress port

802.1Qcc – Stream Reservation comprises improvements and new definitions for the network management and outlines three fundamental configuration models. However, Qcc is just a framework, as several other standards and amendments are required to provide the *managed objects* to perform proper network management for TSN-enabled Ethernet. Critical traffic is managed as so-called *streams*, which are identified by an assigned multicast MAC address and 12-bit VLAN ID and represent frames flowing from a *Talker* to one or more *Listeners* (both are *end stations* and considered *users*) following certain QoS parameters. Usually, it is assumed that the forwarding path of a stream is configured statically (802.1Qca – Path Control), opposed to the usually learned forwarding entries of bridges with the default flooding behavior. There are three models how streams can be configured in the network:

Fully Distributed is the original model used for AVB. The user QoS requirements are propagated along the active topology using a link-local protocol (originally MSRP, but to be replaced by LRP/RAP through 802.1Qdd). The lack of a centralized configuration entity means that the resource allocation decisions are performed locally without the knowledge of the entire network.

Centralized Network / Distributed User (Hybrid) overcomes this limitation by centralizing the resource allocation at a Centralized Network Configuration (CNC) entity, which is a full view of all streams and the entire network. The configuration is pushed to the bridges using a remote network management protocol and standardized managed objects, e.g., NETCONF [132] and YANG [133], resp. The limitation of this model is that the CNC can only configure bridges, but not the end stations - 802.1Qdd will fix this.

Fully Centralized introduces a new entity called Centralized User Configuration (CUC). The CUC is responsible for the discovery of end-stations, retrieval of end-station QoS requirements and capabilities, and configuration of TSN features in the end stations. For this, the CUC communicates with the CNC through an interface currently being standardized in 802.1Qdj. The configuration of the bridges is pushed from the CNC similar to hybrid above.

Note that CUC and CNC are abstract entities in 802.1Qcc and their implementation is left open. PubSub TSN Centralized Configuration (PTCC) [134] is a proposal for a CUC using OPC UA as interface to the users. [135] proposes TSN configuration using OpenFlow, yet does not align with the interfaces of typical industrial equipment. This study proposes an approach using the W3C WoT abstractions, where user QoS capabilities are embedded in TDs, user QoS requirements declared in the mashup applications, and CUC/CNC are running on a WoT Servient with NETCONF and OPC UA protocol bindings.

5.1.3 Design

Term (`qos:`)	**Description**	**Assignment**	**Type**
`flowName`	Unique name for data flow (unique within QoS domain, e.g., TSN domain)	mandatory	string
`talker`	Source of the data flow (e.g., interface MAC address)	mandatory	string
`listener`	Destination of the data flow (e.g., interface MAC address)	mandatory	string
`trafficClass`	Traffic class for the data flow (Literal, one of `deadline`, `latency`, `bandwidth`)	mandatory	string
`cycleTime`	Interval between messages or for bandwidth definition in nanoseconds	mandatory	integer
`maxBytes`	Maximum message size in bytes above Layer 2 (same as QoS capability, originates in Form)	mandatory	integer
`deadline`	Relative time within cycle by when the message must be received in nanoseconds	m. for deadline	integer
`offset`	Relative time within cycle after which the message is sent	o. for deadline	integer
`lossLimit`	Number of acceptable message losses in a row	optional	integer

Table 5.1 QoS requirement terms are assigned values by the mashup application and define the class `FloW`. They need to be parameterized through QoS capabilities (`cycleTime` between `minCycle` and `maxCycle` of the Thing, `maxBytes` given in Form). The combined parameters are passed as inputs to an `ScheduleFlow` Action of a scheduler Thing to request network configuration.

WoT applications, or *physical mashups*[82], are quite similar to industrial automation applications, where one or more controllers contain all the application logic and exchange control messages with devices such as drives or remote I/Os, which can be represented as Things.

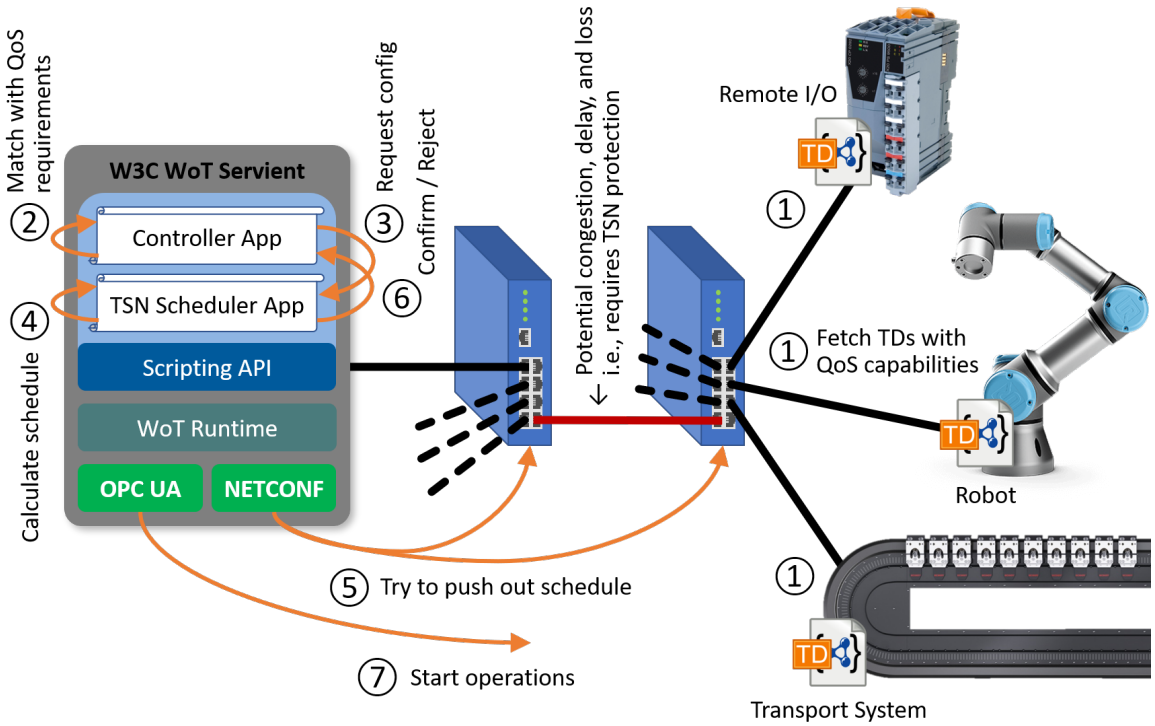


Fig. 5.2 Industrial IoT environment with TSN-enabled Ethernet connectivity and end-devices with OPC UA interfaces, orchestrated through a W3C Web of Things Servient

Concrete QoS requirements are defined by the application logic, as only it knows at what rate and precision it requires inputs and at what rate and precision it wants to control outputs. These requirements, however, can only be met if the Things used have the capability to meet the choices by the application. Hence, Things must declare their QoS capabilities, so that the right application behavior including QoS can be verified. Thing QoS capabilities result from various aspects such as network interface bandwidth or processing power marking a lower bound on the possible cycle time, or mechanical reasons calling for an upper bound between the control messages.

WoT Thing Description in general is designed to retrofit metadata to existing devices. Hence, TDs can also retrofit QoS information to industrial devices that are not aware of TSN or similar QoS mechanisms, but have documented capabilities. This helps to use such devices in a TSN-enabled converged network and, with the right network configuration, give them the illusion of exclusive network usage to meet QoS, while in reality IT traffic and even other real-time applications may co-exist in the same network.

Figure 5.2 shows the operational flow for enabling QoS as part of W3C WoT in an Industrial IoT environment. More in detail:

1. WoT applications such as the 'Controller App' fetch the TDs for the consumed Things as usual, however, the TDs are annotated with QoS capabilities.
2. WoT applications then first validate their internal QoS requirements against the QoS capabilities. If this fails, they have to discover alternative Things or notify the QoS mismatch. (Note that this can also be done offline.)
3. WoT applications then use the QoS requirements terms to request the necessary stream reservations from a scheduler such as the 'TSN Scheduler App' (cf. CUC/CNC). (Note that the Scheduler App may also run in its own Servient or even as an existing service, as long as it provides an exposed Thing interface described by a TD).
4. Schedulers try to calculate a possible schedule for the network.
5. If successful, schedulers try to push out the schedule over a remote network management protocol such as NETCONF.
6. If successful, schedulers confirm the WoT application request. If this or the previous step failed, the request is rejected and the WoT application has to perform error handling (e.g., notify user).
7. If confirmed, WoT applications can start operation

Note that ideally the network components, i.e., bridges, are also treated as Things. The 'TSN Scheduler App' consumes bridge Things through their TDs and uses the NETCONF protocol binding (see Section 5.1.5) to configure them. This may not be the case, when an existing service is chosen as the scheduler.

TD Vocabulary

As the vocabulary is not reviewed by the W3C WG yet, we use the example namespace `http://example.org/2020/wot/qos#`.
The recommended prefix to use in TDs is `qos:`.

The QoS requirements terms listed in Table 5.1 are used to manage data within the WoT application requiring QoS and serve as DataSchema definitions for the network configuration request.

The QoS vocabulary terms in Table 5.2 are used to annotate device TDs with QoS capabilities and parameters. Most got into an instance of the class `Capabilities`, which is similar to the `VersionInfo` container [78]. All of them are optional.

Term (`qos:`)	**Description**	**Class**	**Type**
`capabilities`	Container for the QoS capabilities	Thing	Capabilities
`workingClock`	ID of the working clock (e.g., 802.1AS) if synchronized; required by `trafficClass` deadline	Capabilities	string
`minCycle`	Fastest supported cycle time in nanoseconds	Capabilities	integer
`maxCycle`	Slowest supported cycle time in nanoseconds	Capabilities	integer
`maxBytes`	Maximum message size in bytes above Layer 2 (i.e., including IP headers if any)	Form	integer

Table 5.2 QoS capability terms, which are defined by the Thing. Their assignment is optional for TDs, but they are required if the Thing shall be used with QoS. The working clock must be identical to the one of the application (e.g., same PTP domain).

5.1.4 Protocol Bindings: OPC-UA

OPC UA has two modes of communication: client-server and PubSub. Client-server uses a binary protocol over TCP (UA-Binary). The often mentioned HTTP support has little relevance today, as it actually is WS-* (SOAP). PubSub reuses UA-binary over four possible transports: AMQP, MQTT, UDP, and Ethernet (Layer 2). The UA JSON format intended for browser applications is usually not spoken by devices. The information modeling on top is inspired by the Web and uses *nodes* (identified by namespace and ID) that have *attribute* values (defined per node type) and are connected via bidirectional, browsable *references*. FLC is expected to use client-server for configuration and connection establishment, while PubSub is used for the operational (real-time) data.

Design Challenges

Mapping to Properties, Actions, and Events: The read/write/monitorable variable nodes (Properties), method nodes (Actions), and nodes with alerts (Events) fit directly onto the WoT affordance classes. Other OPC UA nodes serve for structuring the information, which is done with the Linked Data annotations inside the TD. Opposed to variable values, node attributes are usually static and can become TD annotations (e.g., AccessLevel bitmasks can be converted to the standard `readOnly`/`writeOnly` TD fields).

DataSchema: Since OPC UA uses binary data types, JSON schema is not sufficient to describe how data will travel on the wire. Hence, the DataSchema has to be extended with the explicit OPC UA types (e.g., Float vs Double), so that the data can be encoded correctly. This information is binding-specific, and hence should go into the form field. However, the form field is not available to the entity processing the content based on the content type (`ContentSerdes` and its codecs in node-wot). Furthermore, forcing it into the form field would require repeating the whole structure description done by the DataSchema (e.g.,

complex types with object/array). Hence, the `opc:dataType` annotation within DataSchema has been used for this purpose.

Form: href URI: Many protocols lack a URI scheme to string-encode their addressing information. While the scheme `opc.tcp:` is used by OPC UA, it is not registered with IANA. It is only used to pass host address and port; the path part is left open for custom interpretation by each server. To integrate OPC UA better into the Web world, we propose a stricter and more useful definition for the URI scheme: it shall be able to point to address an individual node through the URI. A blocker for this is that numeric IDs are generated non-deterministically on server startup, yet string IDs do work in practice. There are multiple options to encode namespace and node ID such as normal query notation with `?` and `&` or assigning specific path segments like in a RESTful API. Note that the URI scheme defines how the rest after the first colon is interpreted. We decided to reuse the notation that is already used in most of the OPC UA tooling, which uses semicolons to separate the ID tokens:

```
opc.tcp://localhost:5050/server-path?ns=1;s=mynode
```

Although methods are defined for nodes, they are called through a special method service, which requires node-info for the method and for the node on which it is called. We added this as additional semicolon-separated tokens prefix with m for method:

```
opc.tcp://localhost:5050?ns=1;s=mynode;mns=1;mb=9997FFAA
```

Form: contentType There is no available media type for UA-Binary payloads. Thus, we use `application/x.opcua` out of the experimental range.

Resulting Binding Template

The OPC UA binding requires the definition of a URI scheme. The scheme-specific syntax is mostly similar to http URIs. The query-like node-info uses a semicolon delimiter and fixed variables:

```
opc-URI = "opc.tcp:" "//" authority path-abempty [ "?" node-info ]
```

The possible node-info variables are `ns` (namespace), `i` (numeric ID), `s` (string ID), `g` (GUID), `b` (opaque/hex ID), and once more with an `m`-prefix to address a method node.

The binding vocabulary is given in Table 5.3. The possible method names map directly to the OPC UA services except for `Monitor`. This is a shorthand to make the creation of subscriptions for monitored items transparent and leave the management of monitored items to the binding implementation (similar to the hidden session establishment).

Term (opc:)	**Description**	**Class**	**Assignment**	**Type**
methodName	OPC UA method name (Literal, one of Read, Write, HistoryRead, HistoryWrite, Call, Monitor)	Form	default	string
dataType	OPC UA specific type (one of Null, Boolean, SByte, Byte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Float, Double, String, DateTime, Guid, ByteString, XmlElement, Guid, ExpandedNodeId, StatusCode, QualifiedName, LocalizedText, ExtensionObject, DataValue, Variant, DiagnosticInfo)	DataSchema	mandatory for opc.tcp form	string

Table 5.3 OPC UA Binding Template vocabulary

A TD sample with OPC UA binding is given in Listing 5.1. Note that the binding vocabulary does not cover the application-specific OPC UA information model annotations. Converting these OPC UA core and companion standards to Linked Data is out of the scope of this work.

```
"properties": {
  "Velocity": {
    "type": "number",
    "observable": true,
    "opc:dataType": "Double",
    "forms": [{
      "href": "opc.tcp://xts.local:5050/ns=1;
      s=GVL.OPC_Interface.MOVER[1].Input.Velocity",
      "contentType": "application/x.opcua-binary" }] },
  ... },
"actions": {
  "Execute": {
    "input": {
      "type": "boolean", "opc:dataType": "Boolean" },
    "output": {
      "type": "boolean", "opc:dataType": "Boolean" },
    "forms": [{
      "href": "opc.tcp://xts.local:5050/ns=1;s=GVL.OPC_Interface.
      XTS.Input.Execute",
      "contentType": "application/x.opcua-binary",
      "opc:method": "Call" }] } }
```

Listing 5.1 Thing Description sample with OPC UA binding for one of the ten transport system movers used in Section 5.1.6

5.1.5 Protocol Bindings: NETCONF

The Network Configuration (NETCONF) protocol is a network management protocol developed and standardized by the IETF [132]. It is XML-based and uses RPCs over SSH to install, manipulate, and delete configurations of network devices. The information modeling used for the XML messages is YANG [133], which is modular and specialized for the recurring structures in network equipment. There are also two relatives following YANG: RESTCONF [136] mainly used for SDN controllers and the CoAP-based CoRECONF mainly used for LP-WANs. Network devices such as managed bridges, however, usually have a NETCONF server, as there are slightly more features for close control.

Design Challenges

Mapping to Properties, Actions, and Events: YANG structures fit quite well with Web resources, which is intuitive considering RESTCONF. *Leaf-nodes* contain read-/writable data like Properties and defined RPCs map to Actions. Some NETCONF servers have a notification capability that can be used through Events.

DataSchema: Since NETCONF uses text-based XML, JSON schema mostly works. The only obstacle is that some values need to be qualified through a namespace. NETCONF does this through XML node attributes, which are not supported in JSON. Many JavaScript XML libraries as well as RESTCONF solve this by reserving special characters or member names to carry attributes for the parent member. However, due to the JSON-LD nature of TD, this pattern is not used so far and should be avoided. Hence, we flag an object as `nc:container` and mark each property representing an attribute with `nc:attribute` (both terms are of type boolean):

```
"type": "object",
"nc:container": true,
"properties":{
  "xmlns":{
    "type": "string", "format": "urn", "nc:attribute": true
  },
  "value":{ "type":"string" }
}
```

Form: href URI: NETCONF uses SSH and XPath to address nodes in its YANG structure. This information has to be encoded in a potential `netconf:` URI. SSH host and port are trivial, while XPath has special notations that are not allowed by the generic URI syntax (e.g., brackets to select list items). Hence, it made sense to adopt the URI path logic (with selectors and inlined namespaces) from RESTCONF, which is already URI compatible.

This still requires an adaptor to recreate the XPath for NETCONF. Hence, we included an improvement that enables support for the different datastores that NETCONF servers provide: *running* for the current configuration, *candidate* for staging updates, and *startup* for the configuration after boot. RESTCONF defines implicit rules on how data is copied between these stores. The proposed NETCONF URI scheme uses the first path segment to explicitly address the datastore, and thereby keeps this feature. Examples:

```
netconf://localhost:830/candidate/ietf-interfaces:interfaces/interface?type=iana-if-type:modem
```

```
netconf://localhost:830/running/ietf-interfaces:interfaces/interface=eth0/type
```

Form: contentType: Because of RESTCONF, there is a suitable media type registered with IANA: *application/yang-data+xml*.

Special: Different datastores There is another speciality attached to the NETCONF datastores. After writing the Properties, a commit RPC is required to apply the new configuration (copy it to the running store). One option is to make the commit immediately after each successful edit like RESTCONF. Another is to implement the *writemultiple* operation for the binding and performed the commit on a complete batch. NETCONF is still used on network devices because of its detailed explicit features. Hence, we use an explicit commit Action that is semantically annotated with a term from the NETCONF vocabulary.

Special: Namespaces: NETCONF uses namespace CURIEs, a short prefix that must be associated with the fill namespace URI. It is also required to make the href URI work. As this is just additional binding metadata, it simply goes into the form field:

```
"nc:curies": {
  "ietf-interfaces": "urn:ietf:params:xml:ns:yang:ietf-interfaces",
  "iana-if-type": "urn:ietf:params:xml:ns:yang:iana-if-type"
}
```

Special: YANG complexity:

YANG models are specialized for deep and repetitive structures. There might be hundreds of leaf nodes for a single interface, while large bridges might have 48 interfaces or more. This gets very verbose with the flat TD structure. Hence, we recommend making good use of URI Templates (cf. `uriVariables`) when creating TDs with NETCONF binding.

Resulting Binding Template

The NETCONF binding requires the definition of a URI scheme. The scheme-specific syntax is identical to http URIs, except for the `netconf:` scheme and a default of 830 when the port subcomponent is missing. The semantics of the URI scheme is identical to

Term (nc:)	**Description**	**Class**	**Assign.**	**Type**
methodName	NETCONF RPC to be used in the request (Literal, e.g., get-config, edit-config, commit, cancel-commit, copy-config, validate, lock, unlock)	Form	default	string
curies	Dictionary mapping namespace URIs/URNs to CURIEs (used in href and data)	Form	optional	Map of string
container	If true, indicates that the object member has attributes in addition to properties	DataSchema	optional	boolean
attribute	If true, indicates that the property is an attribute of the parent container	DataSchema	optional	boolean
Target	Class to be used as @type annotation indicating the URI variable defines the datastore			
CommitRPC	Class to be used as @type annotation indicating the Action triggers a commit			

Table 5.4 NETCONF Binding Template vocabulary

RESTCONF [136], except that the root resource must be a single path segment that identifies the datastore (i.e., NETCONF "target") and is directly followed by the YANG path (and not the RESTCONF mandatory or optional API resource such as data or operation):

```
1 netconf-URI = "netconf:" "//" authority "/" datastore
2 path-abempty [ "?" query ]
```

The binding vocabulary is given in Table 5.4. The possible method names map to the NETCONF-defined RPCs, but also custom YANG-defined RPCs are allowed.

A TD sample with NETCONF binding is given in Listing 5.2. Note that the binding vocabulary does not cover the application-specific YANG semantics. Converting YANG modules to Linked Data is out of the scope of this work.

```
"properties": {
  "admin-control-list": {
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "index": {
          "type": "number",
          "minimum": 0, "maximum": 127 },
        "time-interval": {
          "type": "number",
          "minimum": 0, "maximum": 4294967295 },
        "gate-state": {
          "type": "number",
          "minimum": 0, "maximum": 255 } } },
    "uriVariables": {
      "datastore": {
        "@type": "nc:Target",
        "type": "string",
        "enum": ["candidate", "running", "startup"] },
      "interface": {
        "type": "integer",
        "minimum": 0, "maximum": 7 } },
    "forms": [{
      "href": "netconf://172.17.0.2:830/{datastore}/
      huawei:tsn-configuration/
      interface={interface}/
      gate-parameters/admin-control-list",
      "contentType": "application/yang-data+xml",
      "nc:curies": { "huawei":
          "urn:ietf:params:xml:ns:yang:huawei-tsn" } }] },
  ...
```

Listing 5.2 Thing Description sample with NETCONF binding for a switch used in section 5.1.6

5.1.6 Proof of Concept

The implementations of the proposed protocol bindings have been merged into Eclipse Thingweb's[6] *node-wot*[7], the official reference implementation of the W3C WoT Working

[6]https://www.thingweb.io/
[7]https://github.com/eclipse/thingweb.node-wot

Group. It is available under *Eclipse Public License v. 2.0* and/or *W3C Software Notice and Document License (2015-05-13)* (similar to MIT License).

The OPC UA binding implemented in the `binding-opcua` package uses the node-opcua[8] library for communication. Currently, only the client part is implemented, as this is the main use case for WoT applications, but exposing a Thing over OPC UA would also be possible. OPC UA clients and their sessions are handled via the ClientFactory pattern. The main effort is to adapt between the DataSchema-based JSON inputs/outputs and the node-opcua API, which requires a library-specific format with UA-specific types.

The NETCONF binding implemented in the `binding-netconf` package uses the node-netconf[9] library for the XML messaging layer and SSH transport, for which in turn the ssh2[10] package is used. The sessions are already abstracted by the library. The main effort is to convert from the href URI to the internally used XPath and namespaces, and to match the JSON inputs/outputs against the library API, which uses its own JSON convention.

The *ContentSerdes* (Content Serializer/Deserializer) of node-wot is designed to handle the JSON input/output mapping to protocol payloads based on media types. The protocols, and hence libraries of the binding implementations, however, do not follow this uniform interface constraint and require a library-specific representation of the body. Hence, we had to implement ContentSerdes codecs that register for a media type, but actually produce and parse library-specific target formats. Forcing the codecs to a standard representation format would increase the implementation as well as processing effort notably. More experience has to show if the split into bindings and ContentSerdes makes sense in the long run; it did make sense for HTTP and CoAP.

The credentials for connecting to the OPC UA server and the ones required for the SSH connection to the NETCONF server can be added into the `wot-servient.configuration.json` as usual.

The TSN Scheduler App is implemented using a proprietary scheduler developed in Java. It provides an interface for the stream requirements (input) and schedule with network configuration (output), which can be used to build a WoT wrapper that exposes the scheduler as Thing.

The design and implementation have been validated through a proof of concept on a real TSN testbed. The testbed hardware is shown in Figure 5.3. It consists of a linear drive transport system as its center, where two-colored levers are mounted on the movers. A retractable cylinder connected to a remote I/O can flip over selected levers by extending at the right time. A robot arm can re-erect selected levers when it is triggered at the right time.

[8] https://github.com/node-opcua/node-opcua
[9] https://github.com/darylturner/node-netconf
[10] https://github.com/mscdex/ssh2

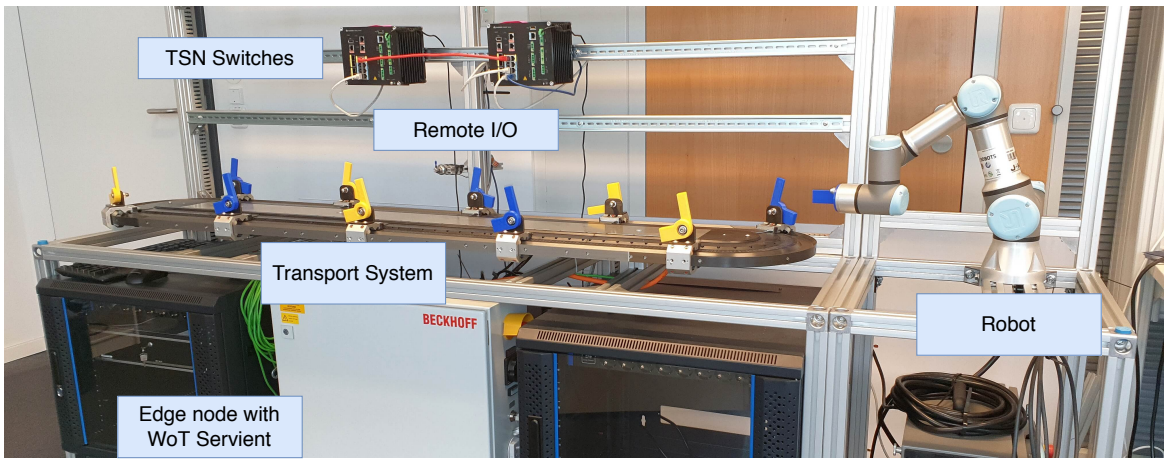


Fig. 5.3 TSN Testbed with WoT-based supervisory logic on edge node and field devices with OPC UA interface

The WoT Servient with the supervisory logic is running on an edge computing node. All components are connected through a TSN fabric provided by some prototype TSN switches made by Huawei.

The demo application is to flip all levers of a selected color and erecting them again through the robot. For this, the supervisory logic requires cyclic updates of the mover positions and must be able to trigger the flipping cylinder and the robot at exact points in time. This can be disturbed by cross-traffic by an exemplary video stream from an IP camera to an HMI (Human-Machine Interface, an industrial touchscreen) and more drastically by a traffic generator.

This implementation proofs that the W3C-WoT based QoS concept is equivalent to classic network management with CUC/CNC. The QoS metadata in the TDs and application itself allows the scheduler to calculate a valid schedule. The network is successfully configured through the NETCONF binding and the operations are carried out correctly through the OPC UA binding.

Neither the industrial devices nor the network equipment were modified for this evaluation. It shows that the TD model is powerful enough to access all features through the affordances and that the concept and bindings work for existing real-world equipment.

Note that at the time of writing, OPC UA PubSub is not available for the industrial equipment used. We thus used client-server, which still works given the generated schedules for the gigabit TSN fabric.

5.2 Web Things migration from Cloud To Edge

5.2.1 Context and Motivations

Problem statement The impressive growth of the Internet of Things (IoT) in terms of devices connected and of data produced can be explained by the versatility of its paradigm, which applies to a wide range of different use cases: from digital manufacturing to smart cities and environmental monitoring [137]. Most of those IoT environments are characterized by the heterogeneity of hardware and software components involved in the system deployment, as well as by the dynamism of the interactions among them. In fact, the actual IoT landscape comprises an uncountable number of protocols, stacks, and cloud ecosystems. Although cloud ecosystems are able to mitigate some of the interoperability issues by means of Web technologies (i.e. REST APIs, JSON, Web Sockets), they are often based on silos architectures with implicit or explicit vendor lock-in. Furthermore, such solutions employ a sensor-to-cloud approach in which devices are managed thanks to cloud connectivity, again with limited extensibility. The WoT architecture and its reference implementation [25] envisage that the run-time environment of a WT (called *Servient*) is statically deployed on a network node. In this Section, we aim at extending the WoT potential for dynamic IoT environments, characterized by the presence of computational nodes on the full IoT spectrum (edge/fog/cloud). More specifically, We address the following key questions:

1. How to enable seamless migration of a WT between two different nodes?
2. How to optimize the performance of a WoT deployment by orchestrating the Web Thing (WT) allocations on a cloud-edge continuum?

Research context Besides the wide literature on live migration of Virtual Machines (VMs) in data-centers, service mobility has gained considerable interest also in the IoT domain for different purposes. On the one side, several large-scale IoT applications operate in dynamic environments, characterized by the rapid changes of bandwidth/computational requests, of devices connected, and of service requirements. To this purpose, IoT platforms like [138][139] provide seamless workload mobility on the edge-cloud continuum, to distribute software tasks to the available computational nodes. On the other side, mobile IoT devices generating space/time-variant data streams are further pushing the research towards flexible platforms able to self-configure to meet the Quality of Service (QoS) for the user applications [140]. To this aim, Mobile Edge Computing (MEC) [141] (and closely related concepts such as Cloudlet [142], Fog Computing [143], and Follow Me Cloud [144]) denote recent computational architectures aimed at running processing tasks in the proximity of the data

sources. A core component of MEC approaches is the seamless service offloading towards the edge/fog servers closer to the current user position [141]; this is typically implemented employing container/VM mobility [145][146]. On the other hand, several studies focused on the migration policies, e.g. to forecast the physical mobility of the IoT devices [147]. The WT migration proposed in this study can be considered a quite novel research problem compared to the literature on MEC systems, both in terms of potential and technical challenges. Regarding the latter, the mobility of a WT from one node to another might impact the operations of other WTs that were currently consuming it; hence, the WT handoff must be properly managed. At the same time, since uniform software interfaces (i.e. the TDs) describe the interactions among the WTs, it is possible to engineer fine-grained and adaptive allocation policies. Those policies can migrate groups of WTs to meet system-wide QoS requirements taking into account the actual network and computational load conditions.

To address the issues above, we propose in this Section the Migratable Web of Things (M-WoT), an architectural framework providing dynamic allocation of W3C WTs to the computational nodes in the edge/cloud spectrum. Specifically, we investigate how to support the stateful migration of WTs among different nodes by managing the handoff procedure on the consumer entities. At the same time, we envisage the presence of a WoT orchestrator, which is in charge of monitoring the interactions among the WTs, and of optimizing the WoT deployment at run-time by allocating the WTs to nodes according to high-level policies (e.g. data locality maximization, latency minimization, etc). More in detail, three main contributions are addressed:

- we present the components of the M-WoT software architecture (including the Thing Directory, the Thing Monitoring layer, and the Thing Orchestrator) and discuss the advantages of WoT migration mechanisms on two target IoT use cases. Moreover, we propose an implementation of the M-WoT framework which relies on Docker containers for Servient mobility.

- Although the framework is general and abstracts from the policy used to compute the WoT deployment, we formulate the WT allocation as a multi-objective optimization problem, by taking into account the inter-host communication load (generated by the interactions among WTs) and the computational load of each host. Then, we propose a centralized heuristic that allows to balance both the metrics above, i.e. to maximize the privacy/data locality while taking into account the fairness on the utilization of the computational resources.

- we validate the M-WoT operations through two testbeds. First, the performance of different allocation policies is evaluated when varying the number of WTs and the

interactions among them. Second, we investigate the effectiveness of the M-WoT framework on a generic IoT monitoring scenario, where real-time diagnostic services are dynamically migrated from cloud to edge nodes based on context conditions.

The evaluation analysis demonstrates that the proposed heuristic can effectively balance the inter-host communication and the computational load when compared to greedy policies. Moreover, in the IoT monitoring use case, the M-WoT solution is able to effectively reduce the diagnostic latency compared to a state-of-the-art, no-migrate approach.

IoT Service Migration A multitude of approaches has been proposed to enable the seamless service migration among nodes of a distributed IoT system. In most cases, the software mobility is aimed at supporting the physical mobility of IoT devices, by ensuring that the data management/processing is always occurring at the edge of the network, hence as close as possible to the current device location. Such a conceptual model is generally denoted as Mobile Edge Computing (MEC) [141], although it presents several overlaps with other state-of-the-art architectures, such as Cloudlet [142], Fog Computing [143], and Follow Me Cloud (FMC) [144]. A detailed illustration of service migration techniques and strategies can be found in [141]; here, the unique challenges of MEC compared to live migration for data centers and to handover management in cellular networks are highlighted. Similarly, in [148], the authors propose the concept of Companion Fog Computing (CFC), a software architecture composed of distributed layers, one running on the mobile device, and another on a fog server; the latter is dynamically allocated to nodes of the fog infrastructure in order to minimize the distance from the current device location. Generally speaking, MEC-related platforms must address two main issues: (*i*) how to define the service migration strategy, by taking into account the current resource utilization of the infrastructure nodes as well as the QoS of the IoT application; (*ii*) how to implement the software mobility, by also handling the migration of the execution state. Regarding the first issue (migration policy), most of QoS-aware service migration policies consider delay as the principal indicator of performance [149] and relies on multi-dimensional Markov Decision Process (MDP) models to capture the system evolution (i.e. the device mobility and consequential service mobility actions) over time (e.g. [147]). Since mobility patterns might be difficult to collect in advance, an increasing number of studies is investigating the application of Machine Learning (ML) techniques for the estimation of the optimal migration policy; an example is constituted by [139], where the usage of Deep Reinforcement Learning (DRL) technique is proved to maximize the users' reward, defined as the difference between the QoS and the migration cost. Among the non-delay oriented studies, we cite the self-organizing service management platform for smart-city proposed

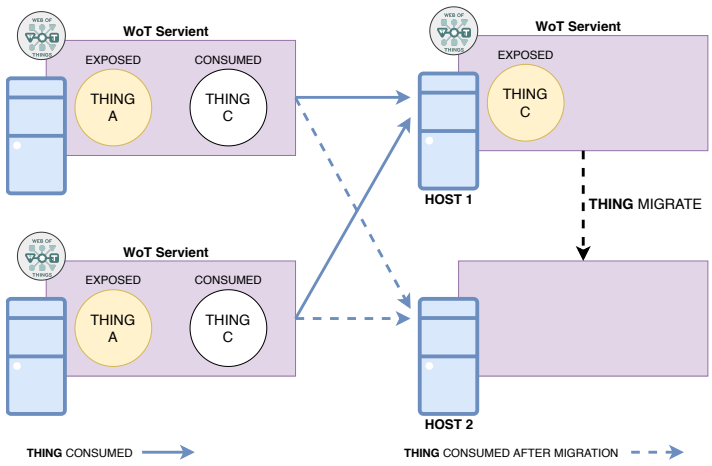


Fig. 5.4 The M-WoT migration environment.

in [150], wherein the ETX (Expected Transmission Count) metric is used to determine the optimal positioning of IoT services over the fog nodes. Regarding the second issue (i.e. software mobility), Virtual Machines (VMs) and containers represent the most investigated techniques to implement stateless or stateful service migration. Proactive migration of VMs according to predicted device mobility is considered in [145]; moreover, in order to reduce the network overhead induced by the VM transfer, a container synthesis technique is applied allowing a fog node to quickly resume the VM execution by applying deltas over a base image. The possibility to perform horizontal (roaming) and vertical (offloading) migration of IoT functions based on Docker containers is demonstrated in [146]. From a performance perspective, the container-based implementation is often considered more suitable for the virtualization at the network edge than the VM-based [151]. This is confirmed by several experimental studies, including [152] that investigates the implementation of Docker-based virtualization mechanisms for IoT data management and demonstrates that the energy impact on single-board computers is negligible. An alternative to the usage of VM/containers is constituted by the migration of active code: to this purpose, the ThingMigrate framework [153] enables the migration of active `Javascript` processes between different machines by employing injection mechanisms to track the local state of each function.

Before illustrating the technical contributions of this study, we introduce the concept of WT migration, and motivate its usefulness on target IoT/WoT use cases.
Let us consider a distributed scenario composed of a set of computing nodes distributed in the full stack of the IoT spectrum (from edge to the cloud), as depicted in Figure 5.4; each node is W3C WoT enabled, i.e. it can host one or more Servients (i.e. the run-time environment of the W3C WoT architecture), and each Servient contains one single WT in running state. WT migration can be defined as the *capability of dynamically offloading a WT between different*

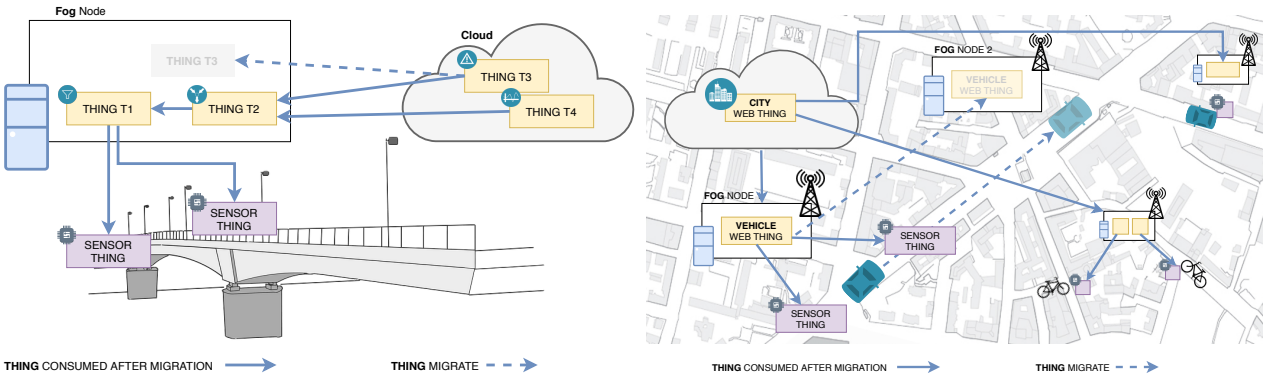


Fig. 5.5 Two possible M-WoT use cases: the data processing service migration (Figure 5.5(a)) and the Digital Twin migration (Figure 5.5(b)).

nodes, by stopping the execution on the source node and re-spawning it on a destination node. The migration process is assumed *stateful*, i.e. the internal state of a WT and its TD should be moved and updated together with the code. In particular, all the current values of its *Properties* and the information describing the current computational context of the WT should be considered as part of its *state* and hence should be migrated. Based on the definitions above, the WT Migration can be considered a particular instance of agent-based live migration [154], where the agents are the WTs and the run-time is represented by the Servient. At the same time, the WoT scenario introduces some unique issues and advantages that are not considered in classical migration approaches (VM/container/agent-based) previously reviewed, and that justify the need of radical different solutions:

- *Thing handoff management*. The W3C WoT allows seamless interactions among heterogeneous software through the operations of WT consuming; if a WT migrates to a different node, all the other WTs that were consuming it must be notified in order to update their Consumed objects and point to the new TD address. The case is depicted in Figure 5.4, where both WTs A and B are consuming WT C; the latter is migrated from Host 1 to Host 2 at some future instant. As a result, a proper signaling procedure must be employed in order to inform WTs A and B of when the activation of WT C at Host 2 has been completed, so that they could consume again the TD of WT C. Also, the migration process introduces a handoff interval, during which WT C might not be able to process remote invocations from WTs A and B; the duration of such handoff is clearly a critical parameter affecting the system performance.

- *Support to Edge-cloud continuum*. Although our implementation is based on the reference W3C WoT run-time [25], Servients might have been designed with minimal

requirements in terms of CPU/memory in order to be executed also on edge servers or even on the extreme edge (i.e. small devices, micro-controllers). As a result, an edge-cloud computing continuum can be devised by allowing WoT software to be seamlessly deployed and dynamically moved over all the nodes of the continuum, in order to guarantee system goals such as delay minimization, workload balancing, network traffic reduction, privacy maximization.

- *Advantage: Support to Group migrations.* As followup of the previous point, a WoT Migration framework could support the mobility of groups of software components (rather than of a single service like it occurs normally in MEC approaches [141]) as a consequence of the active data dependencies (i.e. interactions) among the WTs. Indeed, each WT exposes its Affordances through the TD in a standardized way; as a result, it is possible to build a real-time dependency graph among all the WTs of a distributed WoT system (as further detailed in Section 5.2.3) and consequently envisage allocation policies that determine group migrations of interacting subsets of WTs in order to maximize the data locality. Clearly, group-based migration policies could be deployed also on top of other micro-services architectures; however, for the case of M-WoT, this feature could be supported in a general, protocol-agnostic way since the interactions among the WTs occur according to a standardized interface, and hence they could be easily accounted through the M-WoT monitoring layer described in Section 5.2.2.

Figures 5.5(a) and 5.5(b) show two possible use cases of the WoT migration, related to slightly different conceptual models of WTs: the data processing service migration (Figure 5.5(a)) and the Digital Twin migration (Figure 5.5(b)). More specifically, Figure 5.5(a) depicts a Structural Health Monitoring (SHM) application based on IoT/WoT technologies [155][115], as proposed in the MAC4PRO project [41]. We assume that the monitoring system can work in two modes: *Normal* and *Critical*, denoting two different QoS requirements for risk detection. On the extreme edge there are the sensors (e.g. accelerometers) monitoring the vibrations of a building over time. The sensor data is made available through the Sensor Web Things (SWT) providing functionalities of data querying, and device status querying and updating. The sensor data processing is handled by migratable WTs T_1, T_2, T_3, and T_4 that implement respectively the functionalities of data fusion, data cleaning, data alerting, data forecasting. In Normal mode, T_1, T_2 are executed on a shelter/fog node in the proximity of the monitored structure, while T_3 and T_4 are hosted on a remote cloud; this introduces some network latency in detecting anomalous/dangerous situations (computed by T_3) but at the same time it minimizes the load on fog nodes. At one point of the system execution, we can assume that consecutive data anomalies are detected on the row data (T_2), and hence the monitoring system switches its mode from *Normal* to *Critic*; this action might also request a

higher degree of responsiveness for the diagnostic system. In the M-WoT environment, the mode change can be automatically handled by migrating the T_3 service from the cloud to fog nodes (or vice-versa when the mode switches again to *Normal*), without any manual need of configuration, and without introducing any explicit signaling mechanism among the involved WTs (i.e. T_2 and T_3).
Figure 5.5(b) depicts the second M-WoT use case where the migration involves WoT digital twins. The latter is defined in the W3C standard as a *virtual representation of a device or a group of devices that resides on a cloud or edge node (...) they can model a single device, or they can aggregate multiple devices in a virtual representation of the combined devices*" [24]. To this purpose, we consider a WoT application for the automotive industry like the one proposed in [102]; a WT is associated to each in-vehicle component in order to enable seamless access and interaction to car signals, as proposed in [102]. Like in the previous use case, the Sensor Web Things (SWTs) are in charge of acquiring the data from the hardware of the vehicle. In addition, we assume the presence of a Vehicle Web Thing (VWT), defined as the digital twin of the vehicle as a whole; the VWT is the unique point of access to a subset of the SWTs properties/actions/events, but it also exposes new Affordances derived from the processing and fusion of multiple sensor data, e.g. for real-time vehicle diagnostic. Due to the energy overhead, the VWT is hosted externally to the vehicle, on fog nodes owned by the municipality. While the vehicle moves within the scenario, its VWT is dynamically spawned on the closest fog node, similarly to the MEC applications [148][149], although here the physical mobility of a device induces the mobility of a WT digital twin. In addition, we conceive a city-wide scenario with many and heterogeneous VMTs, associated to different vehicle types (e.g. cars, bikes, buses); the VMTs are in turn consumed by cloud-based City Web Things (CWTs) that provide advanced mobility-related services, such as smart parking, traffic monitoring, multi-modal routing, just as example. We highlight that the number of VMTs can be highly dynamic over time, i.e. new Things might be created or disposed, as an effect of the ground mobility; similarly, the computational load needed for the execution of the VMTs and CWTs might vary over time. In our M-WoT environment, the VMTs are dynamically allocated among the cloud/fog nodes as they appear in the system; moreover, multi-goal load-balancing policies can be used, i.e. to minimize the distance from the data source (i.e. the vehicle) while maximizing the utilization of the computational resources of the fog/cloud nodes.

5.2.2 Architecture

The M-WoT software architecture is depicted in Figure 5.6. We assume a set of W3C WoT Servients, deployed on different nodes; each Servient hosts exactly one WT. Differently

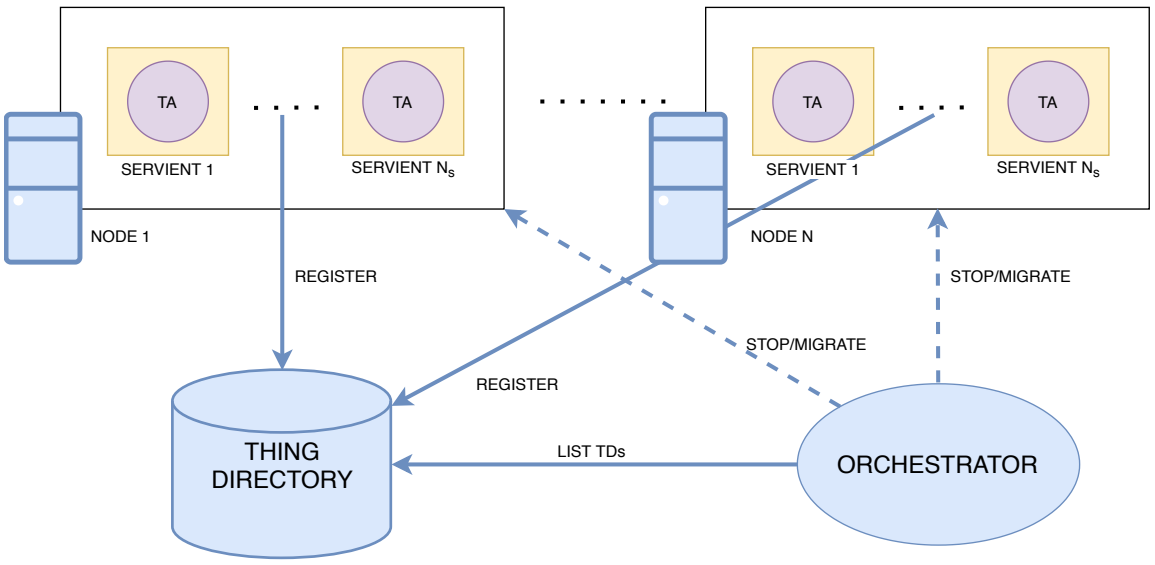


Fig. 5.6 The M-WoT software architecture.

from a legacy W3C WoT deployment, which is assumed static, the M-WoT enables WT mobility between different nodes. To this aim, the M-WoT features two novel components, respectively the Orchestrator and the Thing Directory; these modules do not migrate and can be deployed either on the edge (if the computational requirements are met) or on cloud servers. In addition, a Monitoring Layer has been added to the Servient's stack. In the following, We detail the internal structure of the three modules, while in Section 5.2.2 we clarify the modules' operations when a WT migration process occurs.

Thing Directory

The Thing Directory (TDir), as introduced in Section 3.1.2, serves as registry of the M-WoT resources, i.e. of the active Thing Descriptors (TDs). More in detail, we assume two types of TDs, one associated to WTs, and one with Servients; the latter describes the capabilities of the run-time environment, and it is used to enable the functionalities of the Monitoring Layer described in Section 5.2.2. Once activated, each Servient registers its TD and the TD of the hosted WT on the TDir. The TDir itself plays two main roles. First, it serves as discovery service, i.e. when queried by clients, it returns the list of TDs meeting the query parameters; as a result, the Orchestrator module can be aware of the list of Servients currently available in the WoT scenario. Second, it supports generic push notifications towards WTs/Servients once specific system events are detected, like for instance a WT handoff completion. To this purpose, let us assume that WT T_1 has been consumed by T_2, which is periodically accessing one of its properties. In case T_1 is migrated on a different node, the actual data pipeline

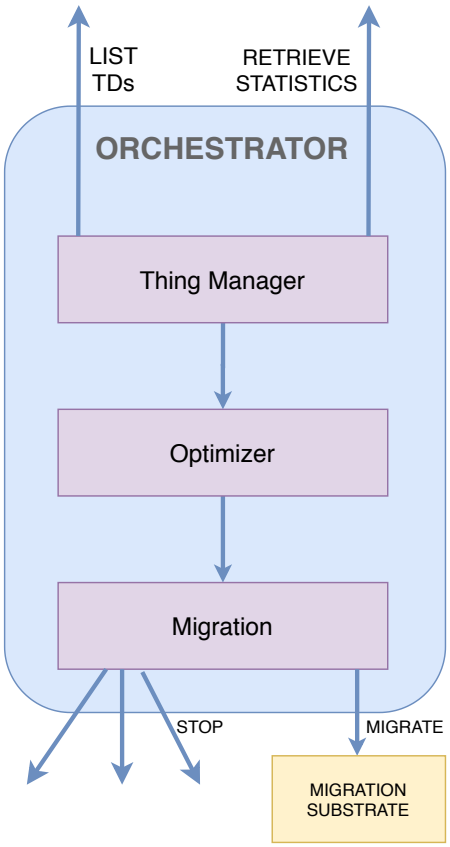


Fig. 5.7 The internal structure of the Orchestrator.

is broken unless T_2 is notified about the mobility event and the new service location. The notification process is illustrated in the sequence diagram of Fig. 5.9, discussed later in Section 5.2.2. Alternatively, a polling mechanism might be employed (involving T_1 and TDir in our example). However, this approach might introduce significant network overhead with consequent bandwidth wastage. Therefore it has not been considered in our solution.

WT Orchestrator

The Orchestrator constitutes the core component of the M-WoT architecture. As explained before, it exploits the TDir to retrieve the list of active Servients (i.e. of their TDs). Then, it periodically queries each Servient through its WoT interface in order to collect live statistics, like the utilization of the CPUs and the network traffic involved by the WT interactions. Based on the received metric values and on the optimization policy in use, the Orchestrator determines the proper allocation of WTs/Servients to nodes. The allocation plan is then transferred to an underlying layer (external to M-WoT), generically called here *Migration Substrate* which is in charge of implementing the physical software mobility

between the source and destination nodes. The steps above are continuously executed by the Orchestrator during the system lifetime; as a result, the dynamicity of the IoT/WoT environment concerning the WT creation/disposal, network bandwidth variation, policy update at run-time, is fully supported. Moreover, in order to favour the platform extensibility, the structure of the Orchestrator has been split into the three main submodules of Figure 5.7, reflecting the internal data pipeline:

1. *Thing Manager*: it periodically polls data from the TDir to manage the list of the active Servients/WTs and their TDs. The list is used to gather periodic reports from each Servient.

2. *Optimizer*: it runs the WT/Servient allocation policy. At the current stage of implementation, the module hosts the graph-based optimization algorithm defined in Section 5.2.3 and the other greedy policies evaluated in Section 5.2.6; however, we remark that any user-defined policy implementing the interface towards the upper (i.e. the Thing Manager) and lower (i.e. the Migration) layers can be installed and used.

3. *Migration*: it receives the deployment plan from the Optimizer, and it implements the WT handoff events. First, it stops the execution of the WTs to migrate at their actual nodes; then, through specific connectors, it issues actions towards the Migration Substrate to enable the physical transfer of the Servients (and of the hosted WTs) from the source to the destination nodes.

The M-WoT architecture does not depend on any specific software mobility technology. Instead, we have introduced an abstraction layer - called the Migration Substrate - which can employ any state-of-the-art solution (via proper migration connectors), such as Docker containers, VMs, or Javascript processes [141][153]. Those connectors will actuate the Optimizer output plan received as input. Concretely, the current implementation relies on Docker Swarm as a default migration connector, as better detailed in Section 5.2.5.

M-WoT Servient

Finally, the M-WoT framework introduces light modifications to the Servient runtime [25] in order to feed the Optimizer with real-time data about system performance. More specifically, a Monitoring API layer has been introduced between the WT application and the Scripting WoT run-time as depicted in Figure 5.8. The layer is in charge of intercepting the invocations to the Scripting API and of generating periodic Thing Reports (`TR`s). The latter can be considered a snapshot of the current Servient/WT execution, and it contains the metrics' values (both for the Servient and WT) required by the Optimizer. The Monitoring layer

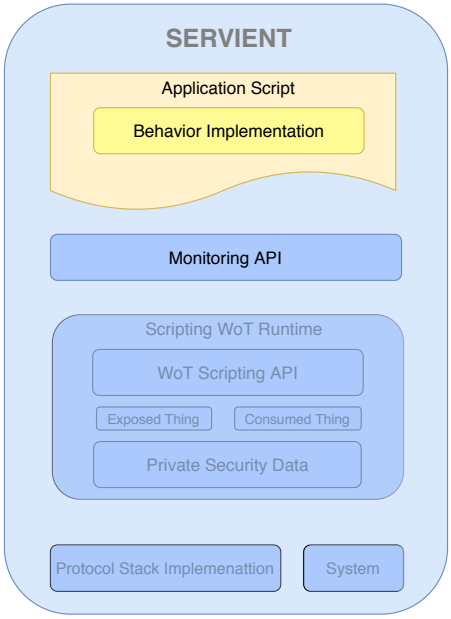


Fig. 5.8 The M-WoT Servient internal structure. The new Monitoring API module is highlighted in solid green.

exposes all the data collected through a proper Affordance action, which has been added to the Servient TD; by invoking it, the Orchestrator can issue a new request of TR generation to the Servient.

Migration example

To summarize the operations of the three components presented so far, we provide an example of the WT migration process. We consider two WTs/Servients, respectively T_A/T_B and S_A/S_B (with T_A running on S_A and T_B on S_B), hosted on nodes N_1 and N_2. We also assume that T_B has consumed T_A and it is periodically reading some of its properties. At time instant t, the Thing Manager queries S_A and S_B in order to collect the TRs; this is implemented by consuming the TDs of the Servients and issuing a `retrieveReport` command (details in Section 5.2.5). Then, the Optimizer is executed; a new allocation is produced where T_A must be moved to N_2. The sequence of operations performing the migration of T_A from N_1 to N_2 is shown in Figure 5.9. First, the current execution of T_A is stopped: this is performed by the Orchestrator (and more specifically by the Migration submodule) by invoking the `stop` action on S_A which, in sequence, stops the WT application, cleans the system resources, retrieves the application data context (i.e. the current state) and returns it. Hence, the application context of T_A is stored as metadata inside the TDir for later use. Next, the Orchestrator (through a proper Connector) issues a request to the Migration Substrate (e.g. Docker Swarm) in order to move

T_A/S_A to the destination node (N_2). After S_A has been respawned, it registers its new TD (with the updated network addresses of its Affordances) in the TDir. Consequently, it queries the TDir to retrieve the T_A's context; the latter is deserialized and injected as a global object inside the T_A's application script. Finally, T_A starts the initialization process and exposes itself by triggering the registration of its TD on the TDir. At this point, T_A resumes in the same state of when it has been stopped and it is considered fully migrated. The TDir pushes a notification to T_B regarding the handoff process; T_B retrieves the new TD of T_A from the TDir and consumes it again in order to point to the updated service location. Finally, T_B restarts interacting with T_A and accessing its affordances.

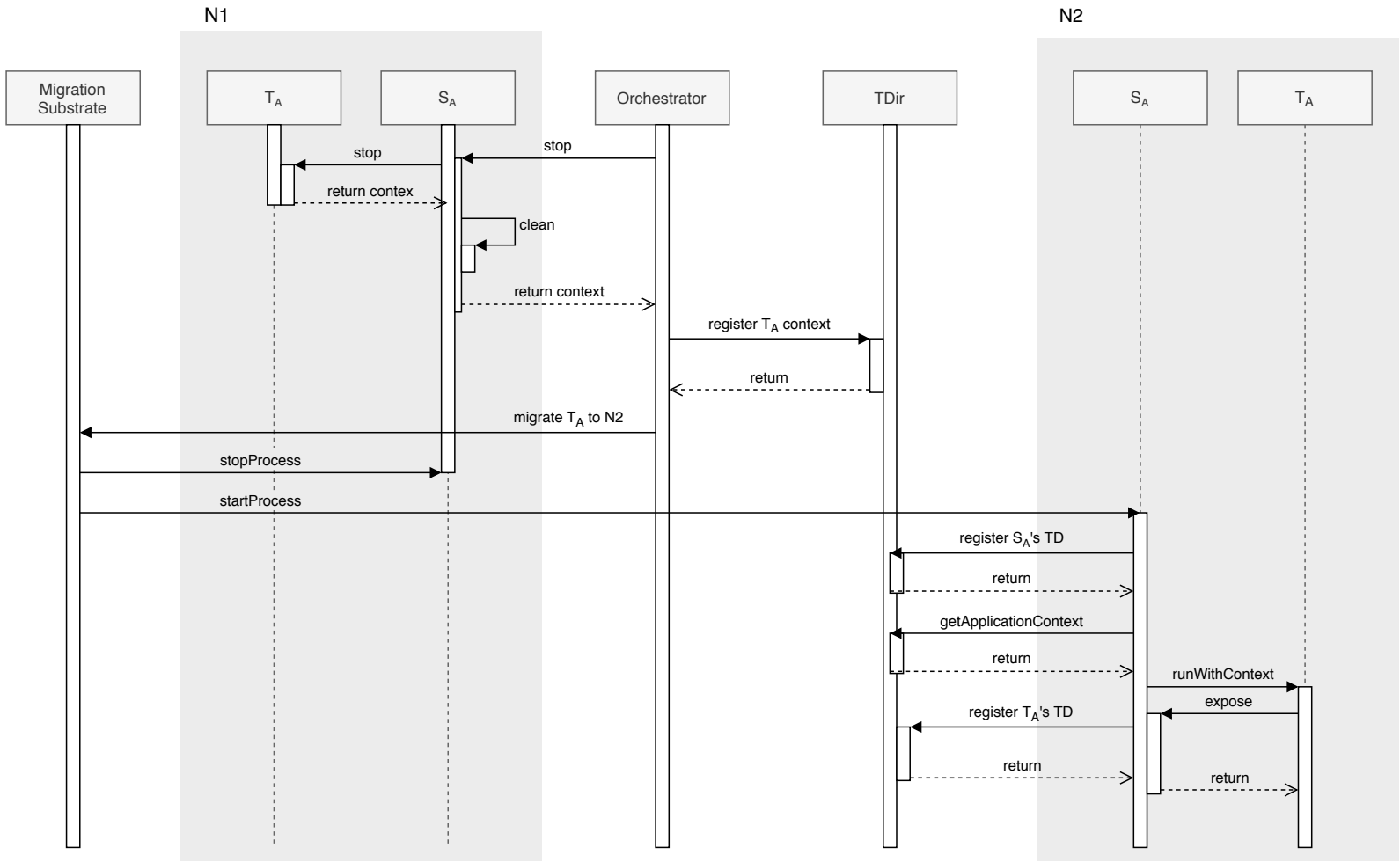


Fig. 5.9 Sequence diagram of a WT migration event.

5.2.3 Migration Policy

In the following, we formally characterize the operations of the Optimizer as a multi-objective optimization problem. For the purpose of this study, we consider a twofold optimization process that takes into account the load-balancing issue (i.e. how much each host[11] is loaded), and the network communication overhead (i.e. how much data is exchanged among hosts). The optimization problem is formally defined in Section 5.2.3. Next, a graph-based heuristic is proposed in Section 5.2.4; its computational complexity is calculated in Section 46.Table 5.5 reports the list of variables introduced in Section 5.2.3.

Problem formulation Regardless of the target use case, we consider a generic WoT deployment with N_{WT} active WTs. The system evolves over ordered time slots $T = \{t_0, t_1, \dots\}$; each slot has a duration of t_{slot} seconds and is equal to the interval between consecutive executions of the migration policy. Let $WT = \{wt_1, wt_2, \ ..., wt_{N_{WT}}\}$ be the set of WTs, which can be heterogeneous in terms of data model (e.g. the Affordances). Without loss of generality, let $A_i = \{a_i^1, a_i^2, ..., a_i^{N_{A_i}}\}$ be the Affordances exposed by wt_i in its TD; each Affordance can represent a property, an action, or an event. The set A_i is assumed static, i.e. wt_i cannot update its TD at run-time (e.g. by defining new properties). The set A_i is assumed static, i.e. wt_i cannot update its TD at run-time (e.g. by defining new properties Let H be the set of hosting nodes, with $H = \{h_1, h_2, ..., h_{N_H}\}$ and assumed heterogeneous in terms of hardware. Indeed, each node may have a different computational power; without loss of generality, this is modeled through a generic computational power index $\gamma(h_l), \forall h_l \in H$ which abstracts from the hardware details, and it is defined as the maximum number of Things that can be executed on that host. The allocation of WTs/Servients[12] to hosts is defined by the policy function $P : WT \times T \rightarrow H$; for each WT wt_i, the value $P(wt_i, t_k) = h_m$ specifies the machine (i.e. h_m) which is hosting it at time slot t_k. Based on the output of the allocation policy, the set $PT_{m,k} \subseteq WT$ denotes the list of WTs that are hosted by host h_m at time slot t_k, i.e.: $PT_{m,k} = \{wt_i \in WT |\ P(wt_i, t_k) = h_m\}$. According to the W3C WoT architecture presented in Section 2.5.5, each WT wt_i can interact with another WT wt_j by first consuming it. This is modeled by assuming that, at each time slot t_k, wt_i can issue a list of requests $R_{i,j,k}$=$\{r_{i,j,k}^1, r_{i,j,k}^2, ...\}$ on the consumed wt_j; each request $r_{i,j,k}^y$ refers to an Affordance of wt_j, and it consists in: a property reading/writing, action invoking or event processing. The mapping between Affordance and requests, i.e. which Affordance is activated by each request, is modeled through a function $f : R_{i,j,k} \rightarrow A_j$. The cost of a

[11] The terms hosts and nodes are used interchangeably in this study.

[12] For ease of disposition, we refer to WT migration in the following by meaning also the migration of the Servient hosting it. We do not model the Servient in the theoretical framework, since we assume that each Servient hosts exactly one WT.

request is defined as the cost of the corresponding Affordance activated on the consumed WT, i.e. $C(r^y_{i,j,k}) = C(a^x_j)$, with $a^x_i \in A_j$ and $a^x_j = f(r^y_{i,j,k})$. It is worth highlighting that the notation above assumes that the same Affordance a^x_i might be activated multiple times by wt_i during the same time slot, although they are considered different requests (e.g. WT wt_i reads twice the same property a^x_j on WT wt_j during time slot t_k). The implementation of each request $r^y_{i,j,k}$ involves some data exchange between WTs wt_i and wt_j; let $B(r^y_{i,j,k})$ be the data exchanged (in bytes) between the two WTs, including both the eventual parameters passed from wt_i to wt_j as well as the eventual return values from wt_j to wt_i. The $B(r^y_{i,j,k})$ value is included in the `TR` message, which is periodically sent by each WT to the Optimizer as previously described in Section 5.2.2. We denote with $B(i,j,k) = \sum B(r^y_{i,j,k})\ \forall r^y_{i,j,k} \in R_{i,j,k}$ the total communication load occurring between WTs wt_i and wt_j at time slot t_k. Clearly, $B(i,j,k) = 0$ whether wt_i is not consuming wt_j at time t_k, or no interaction occurs among them (i.e. $R_{i,j,k} = \emptyset$).

The goal of the Optimizer is to determine the policy which computes - at each time slot t_k- the optimal trade-off between computational resource utilization (i.e. load balancing over the hosts) and data locality (i.e. how much data is transferred among the hosts). To this purpose, we define the Network Overhead (NO) metric as the total inter-host communication load (in bytes) occurring due to interactions among WTs hosted by different nodes. More formally:

$$NO(t_k) = \sum_{wt_i \in WT, wt_j \in WT, P(wt_i,t_k) \neq P(wt_j,t_k)} B(i,j,k) \tag{5.1}$$

It is important to clarify that the $NO(t_k)$ metric above quantifies the *end-to-end*, application-layer, communication traffic between nodes of the M-WoT cluster, generated by the interactions among different WTs; it does not include the network-layer overhead (e.g. caused by multi-hop message forwarding among the routers) since the M-WoT framework is implemented at the application layer and the knowledge of the topology of the underlying network infrastructure is not assumed. Similarly, we introduce the Host Fairness (HF) metric defined as the difference between the most loaded and most unloaded host of the cluster, i.e.:

$$HF(t_k) = max_{h_m \in H} L(h_m, t_k) - min_{h_m \in H} L(h_m, t_k) \tag{5.2}$$

here, $L(h_m, t_k)$ defines the computational load ratio of h_m at time slot t_k, and it is related to the number of WTs hosted by it over its computational power, i.e.:

$$L(h_m, t_k) = \frac{|PT_{m,k}|}{\gamma(h_m)} \tag{5.3}$$

Let $p^{t_k}_{wt_i,h_m}$ be the binary variable indicating the WT allocations, defined $\forall t_k \in T$, $\forall wt_i \in WT$, and $\forall h_m \in H$ as follows:

$$p^{t_k}_{wt_i,h_m} = \begin{cases} 1 & \text{if } P(wt_i,t_k) = h_m \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

Through the NO and HF metric introduced above, the migration problem can be formally defined as follows:

$$\min_{p^{t_k}_{wt_i,h_m}} \quad NO(t_k) \tag{5.5}$$

$$\text{s.t.} \quad L(h_m,t_k) \leq 1 \quad \forall h_m \in H \tag{5.6}$$

$$HF(t_k) \leq \Delta \tag{5.7}$$

Constraint 5.6 ensures that the allocation on each host does not exceed the computational capabilities of that host ($\gamma(h_m)$). In Constraint 5.7, Δ is a user-defined parameter, which quantifies the trade-off previously mentioned. It is easy to notice that the HF and NO metrics are tightly coupled: minimizing the network load can be achieved by a policy that allocates all the WTs to the same host. However, this constitutes the worst-case for the load fairness. Hence, two extreme scenarios are possible:

- The system goal is to minimize the data exchanged over the network, regardless of the service latency; this might the case of an edge-cloud IoT scenario, where the stake-holder is interested in minimizing the amount of data transferred toward a remote infrastructure for privacy reasons. In this case, $\Delta = \infty$.
- The system goal is to minimize the service latency, by avoiding the presence of performance bottlenecks, i.e. overloaded hosts, while still mitigating the amount of inter-host communications. In this case, $\Delta \leq 1$.

All the intermediate situations are modeled through a proper tuning of the Δ parameter, which is assumed as input of the optimization problem.

5.2.4 Proposed Heuristic

We propose a graph-based heuristic that ensures the constraint 5.7, while relaxing the constraint 5.6 and addressing the goal function (Equation 5.5) through a greedy approach. The solution relies on the construction of a WT dependency graph $G(V,E,W,L)$ which models the interactions among the WTs:

Parameter	Description
t_{slot}	Slot Length (interval between consecutive migration policy executions)
$T = \{t_0, t_1, ...t_k\}$	Temporal evolution as sequence of fixed-length time slots
H	Set of hosting nodes composing the M-WoT cluster
$N_H = \|H\|$	Number of hosting nodes
WT	Set of Web Things (WTs)
$N_W = \|WT\|$	Number of Web Things (WTs)
$\gamma(h_l)$	Computational power of hosting node h_l
$P(wt_i, t_k)$	Allocation function, returning the hosting node of WT wt_i at time t_k
$p^{t_k}_{wt_i,h_m}$	Binary variable defining the allocation of the WTs over time depending on $P(wt_i, t_k)$
$PT_{m,k}$	Set of WTs hosted by node h_m at time wt_i
$A_i = \{a_i^1, a_i^2, ..., a_i^{N_{A_i}}\}$	List of Affordances exposed by WT wt_i
$r^y_{i,j,k} \in R_{i,j,k}$	Affordance request issued by wt_i toward wt_j at time t_k
$R_{i,j,k}$	Cumulative list of Affordance requests issued by wt_i towards wt_j at time t_k
$B(r^y_{i,j,k})$	Traffic load (in bytes) between WTs wt_i and wt_j to execute request $r^y_{i,j,k}$
$B(i,j,k)$	Total traffic load (in bytes) exchanged between wt_i and wt_j at time t_k
$NO(t_k)$	Total inter-host communication at time t_k in the M-WoT cluster
$L(h_m, t_k)$	Load Ratio of host h_m at time t_k: number of WTs hosted, normalized over $\gamma(h_m)$
$HF(t_k)$	Difference between most loaded and unloaded hosting nodes (loads expressed in terms of $L(h_m, t_k)$)
Δ	User-defined constraint on the $HF(t_k)$ value

Table 5.5 List of variables and parameters introduced in Section 5.2.3.

- V is the set of vertexes; each vertex represents a WT, hence $V = WT$ and $v_i = wt_i$ $\forall wt_i \in WT$.
- E is the set of edges; each edge $e_l(v_i, v_j)$ connects two vertexes $v_i, v_j \in V$ and models the interaction between the two WTs. More specifically, there exists the edge $e_l(v_i, v_j)$ only if $B(i,j,k) > 0$ or $B(j,i,k) > 0$.
- $W : E \to \mathbb{R}$ is a weight function, assigning a cost to each edge $e_l(v_i, v_j) \in E$. Here, the value $W(e_l(v_i, v_j))$ quantifies the total data exchanged among WTs, in case wt_i is consuming wt_j or vice versa, i.e. $W(e_l(v_i, v_j)) = B(i,j,k) + B(j,i,k)$.
- $L : V \to \mathbb{R}$ is a load function, assigning a cost to each vertex $v \in V$. If we assume to know the CPU load ($C(r)$) induced by each request received by wt_j, then $L(v_j)$ can be defined in a fine-grained way as $L(v_j) = \sum_{wt_i \in WT} \sum_{r \in R_{i,j,k}} C(r)$. In this study, we do not assume such knowledge, hence we generally set $L(v_i) = 1$ $\forall v_i \in V$, i.e. all WTs are assumed to produce the same load, while the total load of host h_m (denoted as $L(h_m)$ in the following) is simply the number of WTs hosted.

The graph G is built and continuously updated by the Optimizer by processing the `TR` messages received by each Servient. At the beginning of each time slot (e.g. t_k), the Optimizer visits the graph and allocates the WTs to hosts according to the policy output (i.e. the $PT(h_i, t_k)$ values); since the policy is computed once for each slot, we omit the temporal notation (i.e. the t_k) in the rest of this Section.

Algorithm 1: The graph-based heuristic

Input: Dependency graph $G(V,E,W,L)$, time slot t_k
Output: Allocation sets $PT(h_m,t_k)\forall h_m \in H$

$GC = \{G_1, G_2, ..., G_{N_C}\} \leftarrow$ `GetGomponent`(G)
forall $G_i \in GC$ **do**
  $L(G_i) \leftarrow \sum_{v \in G_i} L(v_i)$
end
$GC \leftarrow$ `Sort`(GC, L)
$H \leftarrow$ `Sort`(H, γ)
$cont \leftarrow 0$
while $GC \neq \emptyset$ **do**
  $G_h \leftarrow$ `Head`(GC)
  $PT(cont,t_k) \leftarrow PT(cont,t_k) \bigcup G_h$
  $L(h_{cont}) \leftarrow L(h_{cont}) + L(G_h)$
  $cont \leftarrow (cont+1)\%N_H$
end
$\langle balanced, h_{min}, h_{max} \rangle \leftarrow$ `CheckBalanced`(H, Δ)
while $balanced ==$ `false` **do**
  forall $v_i \in PT(h_{max},t_k)$ **do**
    $loss \leftarrow$ `TotInteractions`$(v_i, PT(h_{max},t_k))$
    $gain \leftarrow$ `TotInteractions`$(v_i, PT(h_{min},t_k))$
    $overhead(v_i) = loss - gain$
  end
  $v_s \leftarrow$ `argmin`$(overhead(v_i))\ \forall v_i \in PT(h_{max},t_k)$
  $PT(h_{min},t_k)) \leftarrow PT(h_{min},t_k) \bigcup \{v_s\}$
  $L(h_{min}) \leftarrow L(h_{min}) + L(v_s)$
  $PT(h_{max},t_k)) \leftarrow PT(h_{max},t_k) \setminus \{v_s\}$
  $L(h_{max}) \leftarrow L(h_{max}) - L(v_s)$
  $\langle balanced, h_{min}, h_{max} \rangle \leftarrow$ `CheckBalanced`(H, Δ)
end
return PT

Function `CheckBalanced`(H, Δ)**:**
  $h_{min} \leftarrow$ `argmin`$(L(h_i))\ \forall h_i \in H$
  $h_{max} \leftarrow$ `argmax`$(L(h_i))\ \forall h_i \in H$
  $n_{iter} \leftarrow n_{iter} + 1$
  if $L(h_{max}) - L(h_{min}) \leq \Delta$ **then**
    $balanced \leftarrow$ `true`
  else
    $balanced \leftarrow$ `false`
  end
  return $balanced, h_{min}, h_{max}$

Function `TotInteractions`(v_s, S)**:**
  $interactions \leftarrow 0$
  forall $v_j \in S$ **do**
    $interactions \leftarrow interactions + W(e(v_i, v_j))$
  end
  return interactions

The rationale of the proposed policy is the following. First, we compute the set of connected components on the dependency graph G. By construction, each component contains a closed set of interacting WTs; hence, the network overhead occurring among different graph components is equal to zero. The load of each component is defined as the sum of loads of its WTs; next, graph components are ordered based on their load values, and assigned to hosts in a round-robin way. In case the constraint 5.7 (load fairness) is satisfied, the algorithm stops its execution. Otherwise, we break the connected components computed so far (hence, introducing some network overhead at each iteration) by iteratively migrating one WT from the most loaded host to the most unused one, until the constraint 5.7 is satisfied (or alternatively a maximum round of interactions have been executed). The migrated WT wt_i is selected in a greedy way as the one which minimizes the network overhead, computed as the difference between: (*i*) the new overhead generated when detaching wt_i from the source host and (*ii*) the performance gain on the destination host, caused by the fact that wt_i has become a local service on that host.

Algorithm 1 shows the pseudo-code of the proposed heuristic. First, we build the dependency graph $G(V,E,W,L)$ and we compute its set of connected components, denoted as GC at line 1. The load of each component G_i (i.e. $L(G_i)$) is estimated as the sum of the loads of its vertices (line 3). Then, we order the set GC based on the load values, and the set of hosts H based on the computational power of it, represented by the γ metric. To this purpose, at lines 5-6, the function `Sort` (not reported here) is sorting a set passed as the first argument in descending order according to the metric values given by the second argument. The loop at lines 8-13 assigns sub-graphs to hosts in a round-robin, by also updating the load for each host as the load of its vertexes/WTs (line 11). Next, we check whether constraint 5.7 is satisfied through the `CheckBalanced` function (lines 30-39), which also returns the hosts associated to the highest and lowest load values, respectively h_{max} and h_{min}. If the load difference is lower than the user-threshold Δ, than the current allocation is returned. Vice-versa, a greedy mechanism is implemented through the loop at lines 15-27; here, at each iteration, a candidate WT v_s is migrated from h_{max} to h_{min} (lines 22-25) (by consequently updating the per-host load information) and the load balance condition is evaluated again at line 26. The WT/vertex to migrate (v_s) is selected as the one that minimizes the overhead function at line 21. This latter takes into account: (*i*) the total amount of network communications (in bytes) between v_s and any other WT hosted by h_{max}, which will now become inter-host communications and hence will constitute a network overhead after the WT migration (the value is stored within the $loss$ variable at line 17); (*ii*) the total amount of network communications (in bytes) between v_s and any other WT hosted by h_{min}, which will now occur locally (intra-host communication) and hence will reduce the network overhead (the value is stored within

the *loss* variable at line 18). The computation of gain/loss values is performed through the helper function `TotInteractions` (lines 41-46) that returns the total number of interactions occurring between a target vertext/thing (v_s) and a set of vertexes (S) provided as inputs, over the dependency graph G.

Computational Complexity

The computational complexity is expressed in terms of N_W (number of WTs) and N_H (number of nodes) for the worst-case scenario. At line 1 of Algorithm 1, the connected components of graph G are computed; this operation is completed in time O(N_W) through a DFS graph visit. Then, from line 8 to line 13, the connected components are assigned to the computational nodes; again this is performed in O(N_W). The complexity of the balancing loop (from line 15 to line 26) depends on the Δ value and on the L function definition. We assume that We assume that all hosts are homogeneous ($\gamma(h_m)$=1 $\forall h_m \in H$), hence $L(h_m, t_k) = PT_{m,k}$. The assumption is compliant with the experimental analysis presented in Section 5.2.6. By considering a totally unbalanced allocation of WTs to nodes, the loop is executed for $N_W - \Delta$ times; the internal loop (lines 16-20) has a complexity of $O(\frac{N_W}{N_H})^2$ since we visit each WT hosted by the most used node, and for each WT we compute the total NO with the WTs hosted on the most unused node. Finally, the `CheckBalance` function loops over the N_H set hence it has a complexity of $O(N_H)$. Since we expect that $N_W >> N_H$, the overall complexity of Algorithm 1 is $\sim O(N_W^2)$.

5.2.5 Implementation

The implementation of the architecture components presented in Section 5.2.2 is detailed here. This solution can be considered an extension of the Thingweb *node-wot*[25], the official reference implementation of the W3C WoT Working Group, to which we added specific primitives in order to support the WT migration process.

Thing Directory & Orchestrator

The TDir is implemented as a dedicated (non-migratable) WT, which is hosted by a WoT Servient exposing a specific API for managing TDs and contexts. Among the most important interaction affordances we cite: the `registerThing` action that takes a TD as input, and makes it globally available to the other M-WoT components; the `getThingById` and `listThings` actions, which respectively return one or more TDs based on the id or on a semantic filter; the `getContextById` action which returns the context associated to a WT,

and the `thingRegistered` event, which is triggered each time a WT registers itself on the TDir, and causes its TD to be broadcasted to all the subscribers. The Orchestrator is implemented as a `Node.js` application written in `TypeScript` and using the *Nest*[13] (v6) framework in *standalone application* mode. The Orchestrator includes several modules working in synergy, and corresponding to the three components presented in Section 5.2.2:

- *Thing Manager*: it provides a `TasksManager` capable of executing generic tasks at a specific schedule; the functionality is implemented by the `@nestjs/schedule` package, which in turn uses the `node-cron`[14] package. Among the others, we cite the the `collectReports` task that periodically retrieves the list of active WTs through the TDir and invokes the `retrieveReport` action on each one of them in order to get the corresponding `TR`s.
- *Optimizer*: it provides the data structures representing the current status of the M-WoT deployment. In particular, it records the live metrics of WTs (i.e. interactions with other WTs) and the list of the hosting nodes. Moreover, it provides the `Policy` abstract class, with a `getAllocation` method that returns the planned allocation of WTs to nodes (i.e. the $PT(h_m, t_k)$ sets of Algorithm 1). Any new policy installed in the Optimizer must implement the method above.

WoT Servient

The default *node-wot* [25] framework has been extended in two directions: the script run time has been proxied with a monitoring module, and the default CLI implementation has been modified to handle WT state injection and retrieval.

Monitor APIs The Monitoring API is a collection of `Typescript` classes and functions collecting the data needed by the Optimizer. More specifically, the Monitoring API intercepts any invocation of the WTs to the underlying WoT scripting functions and updates the number of activations of each property/action/event as well as the total time of completion. Then, it stores such data inside the `TR`.The main fields of the `TR` include: the `id` of the WT being monitored, the `hostID` of the node hosting the WT/Servient, the `serviceID` used to map the WT to the corresponding docker swarm service, the average CPU and memory utilization of the node, and the Interaction List. The latter contains statistics related to the interaction with each consumed WT, and more specifically the number of times a specific Affordance has been activated, and the latency involved in the request-response.

[13] https://nestjs.com/
[14] https://github.com/kelektiv/node-cron

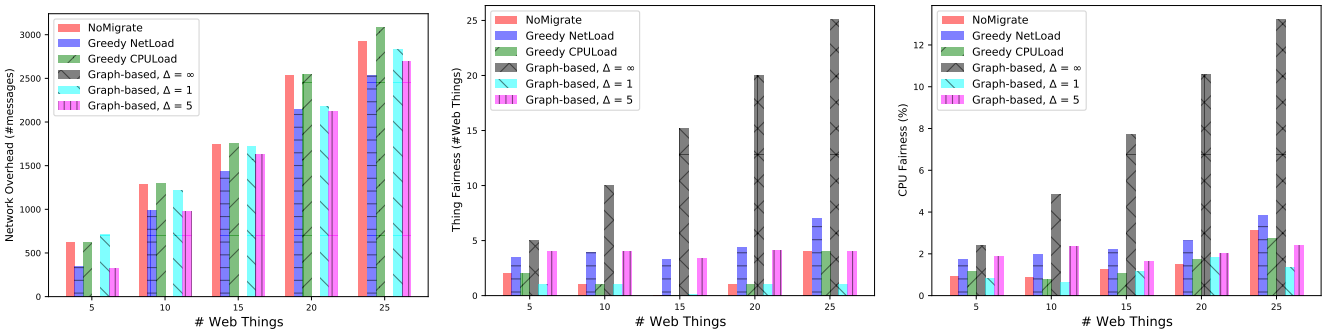
Context migration In the case of active WT, the M-WoT framework supports the migration of its context, i.e. all the information characterizing the internal *state* and including: `Global variables` of the Thing Application, the `Properties values` and the current `State` of eventual external libraries in use. Before migration can start, all the possible running operations should be interrupted and the `context` must be collected. This has been implemented by adding the `stop` method to the TD of the WT, which disables all its Affordances in order to avoid a possible state change during the context saving process. After that, it collects the WT `Context` and returns it to the Servient; the context is then stored on the TDir as described in Section 5.2.2. After the new Servient has been deployed, and before running the migrated WT, it makes a request to the TDir (by using the `Thing ID`) for retrieving the context. The latter is then passed to the WT to be loaded, hence restoring the state at the time of migration. For sake of simplicity, and to ease the programmers' tasks, we automatized the process of adding all the auxiliary functionalities inside the WT behaviour. More in detail, the methods for stopping the WT Affordances and returning the context are automatically injected into the code of the WT application by the Servient before exposing it. The servient searches for a specific comment in the script (*/*INIT*/*) to understand whether and where the M-WoT code should be inserted. The only operation required to the programmer in order to make a WT migration-enabled is to add such comment to the application code.

5.2.6 Validation

In this Section, the performance of the M-WoT framework via a twofold experimental evaluation is tested. First, we compare different migration policies, including multiple variants of the graph-based heuristic presented in Section 5.2.3, on ad-hoc edge scenarios. Then we investigate the effectiveness of the WT migration mechanisms on the edge-cloud continuum for an IoT monitoring use case. More in detail, we evaluate a concrete IoT structural monitoring application inspired by one of the use cases presented in Section 5.2.1 (see Figure5.5(a)). The characteristics and parameters of each scenario are discussed separately in Sections 5.2.6 and 5.2.6.

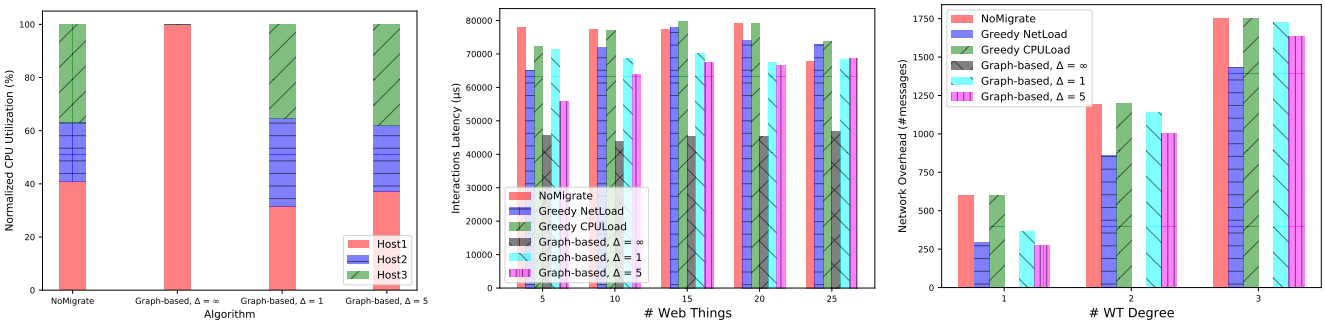
Policy Analysis

We consider a distributed setup composed of three edge servers (i.e. $N_H = 3$), physically located at the DISI/ARCES departments of the University of Bologna, and connected through an Ethernet LAN, at one hop distance one from each other. Specifically, two servers are equipped with 4-core 2 GHz CPUs and 4 Gb of RAM, while the third server is equipped



(a) (b) (c)

Fig. 5.10 The *NO*, *TF* and *CF* metrics for the six policies when varying the number of active WTs are shown respectively in Figures 5.10(a), 5.10(b) and 5.10(c).



(a) (b) (c)

Fig. 5.11 The average utilization of each computational node is shown in Figure 5.11(a). The *IL* metric when varying the number of active WTs is shown in Figure 5.11(b). The *NO* metric as a function of the WT degree is reported in Figure 5.11(c).

with an Intel Xeon E5440 processor with 32 Gb of RAM. Moreover, the Orchestrator and the TDir have been installed on a different node within the same data center. Therefore, in total, the experimental setup is composed of 4 nodes, three of which constitute the M-WoT deployment space, and can be used to host the WTs. On this space, we deployed N_{WT} Servients, each hosting exactly one WT; at the startup, the Servients are randomly allocated over the available nodes. The WT interactions are modeled as follows. We abstract from the physical meaning of the WT and the correspondence to specific real-world applications since the focus is on the assessment of the migration operations and on the evaluation of the policies' performance. Hence, each WT exposes exactly one action in its TD (e.g. `test`), which computes a sequence of trigonometric operations (mainly *tan* and *atan*) in order to generate some CPU load. Each WT (e.g. wt_i) consumes exactly other N_C WTs, chosen randomly among the N_{WT} available. On each consumed thing wt_j, wt_i issues a request for

the `test` action every 1.5 seconds. In order to automatically apply the test configurations on each WTs, we implemented a *Mashup* application, i.e. a WoT client that is in charge of consuming the WTs involved in each test instance and of passing them the proper setup (e.g. the list of WTs to consume). Every 45 seconds, the Orchestrator collects the Thing Reports (`TR`) produced by each Servient; every 190 seconds, a new WT allocation is computed by the Optimizer according to the current policy, and implemented through proper WT migrations among the edge servers. The latter is also the duration of one time slot (i.e. t_{slot}=190 seconds), in accordance with the problem formulation presented in Section 5.2.3. The setting of t_f and t_{slot} parameters allows the Optimizer to collect at least three reports from each WT and hence to estimate the WT interactions before computing a new allocation of WTs to nodes. The performance analysis is based on the following metrics:

- *Network Overhead* (*NO*): this is the performance index defined by Equation 5.1 and quantifying the amount of inter-host network communications produced by remote WT interactions. Differently from the theoretical model, we compute the *NO* in terms of number of interactions rather than of bytes, since all the WT interactions refer to the same affordance (i.e. the `test` action); this is the equivalent to set $B(i,j,k)$=1 in Equation 5.1, $\forall wt_i, wt_j \in WT, t_k \in T$.

- *CPU Fairness* (*CF*): this is the performance index defined by Equation 5.2 and quantifying the fairness unbalance in terms of max-min difference of the average CPU occupation loads among the N_H nodes of the cluster. We set $\gamma(h_l) = 1, \forall h_l \in H$.

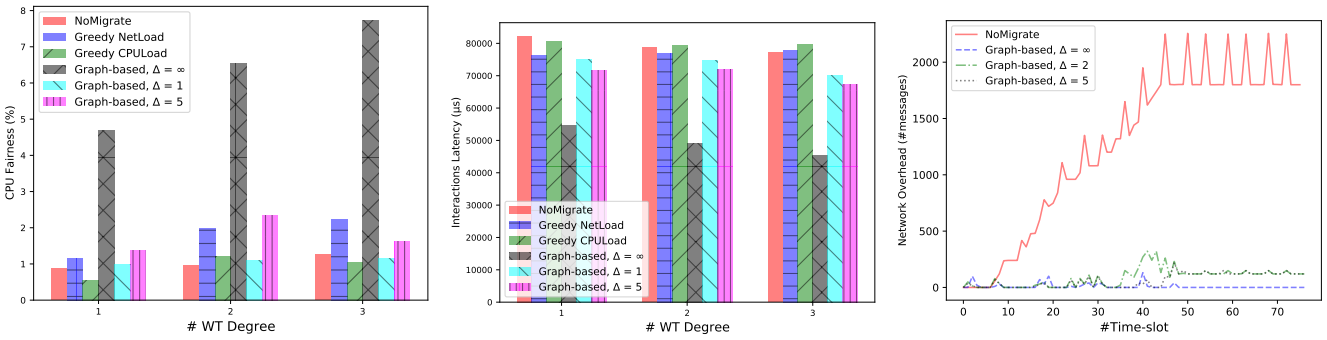
- *Thing Fairness* (*TF*): this is similar to the *CF* metric, however the fairness unbalance is expressed in terms of number of WTs hosted respectively by the most loaded and unloaded node (rather than of average CPU values).

- *Interaction Latency* (*IL*): this is the average latency required to perform a WT action invocation issued by an external WT; more explicitly, this is the average time lapsed from when wt_i issues a `test` action on wt_j to when the corresponding reply is received. Hence, it takes into account both the processing delay and the network delay in case wt_i and wt_j are executed on different nodes of the cluster.

We compared the following policies:

- *NoMigrate*: this is the state-of-the-art WoT solution, i.e. the WTs are statically deployed on nodes and they are not migrated during the whole system lifetime.

- *Greedy NetLoad*: this is a greedy policy that aims at minimizing the *NO* metric. At each time slot, it selects the WT generating the highest NO, and moves it towards the same node of the consumer WT.

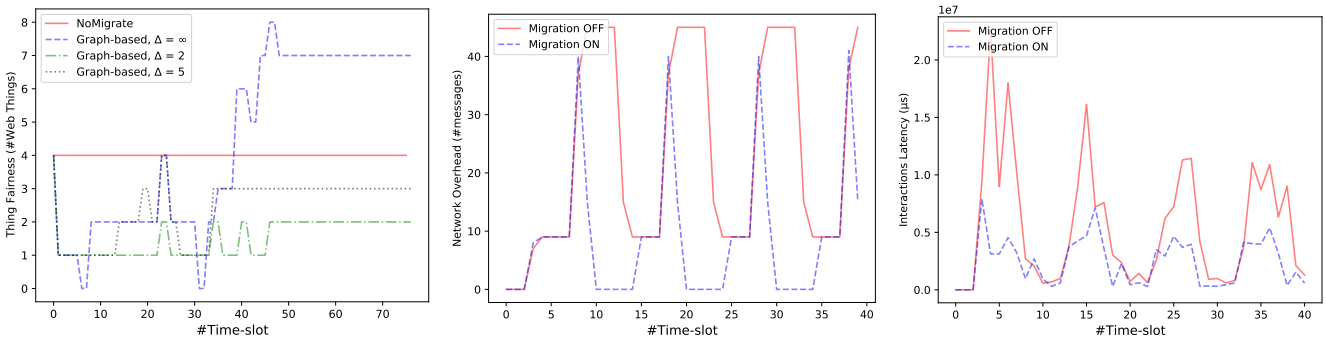


(a) (b) (c)

Fig. 5.12 The *CF* and *IL* metrics when varying the WT degree are shown respectively in Figures 5.12(a) and 5.12(b). The *NO* over time slots in a dynamic WoT deployment where the number of WTs is varied over time is reported in Figure 5.12(c).

- *Greedy CPULoad*: this is a greedy policy that aims at minimizing the *CF* metric. At each time slot, it selects the edge node of the cluster associated with the highest average CPU load, detaches one WT and moves it towards the node with the lowest CPU load.
- *Graph-based,* $\Delta = \infty$: this is the WT dependency-graph policy presented in Section 5.2.3; we set $\Delta = \infty$, hence the policy aims exclusively at minimizing the *NO* metric, while no load-balancing action is executed (i.e. lines 16-26 of Algorithm 1 are skipped).
- *Graph-based,* Δ=*5*: this is again the policy of Section 5.2.3, where the balance parameter is put into action. The latter is expressed in terms of number of WTs, hence the policy computes a minimal *NO* solution ensuring that TF metric cannot exceed the Δ threshold equal to 5.
- *Graph-based,* Δ=*1*: this is similar to the previous policy, however we request the maximum balancing of the WT allocations over the nodes of the cluster.

For each configuration, we ran 10 repetitions, and then averaged the metric values; on each repetition, a random initial allocation of WTs to nodes, and random dependencies among the WTs are considered.
Figure 5.10(a), 5.10(b), 5.10(c) and 5.11(a) show the metrics previously introduced when varying the policy in use and the N_{WT} configuration, i.e. the number of WTs in the scenario. The N_C value is fixed and equal to 3, i.e. each WT consumes exactly 3 peers, randomly selected. From the *NO* values of Figure 5.10(a), we can notice that the amount of inter-host communications increases with the number of active WTs, as expected. At the same time, the *Graph-based* and the *NetLoad* policies are more effective than the *NoMigrate* and the



(a) (b) (c)

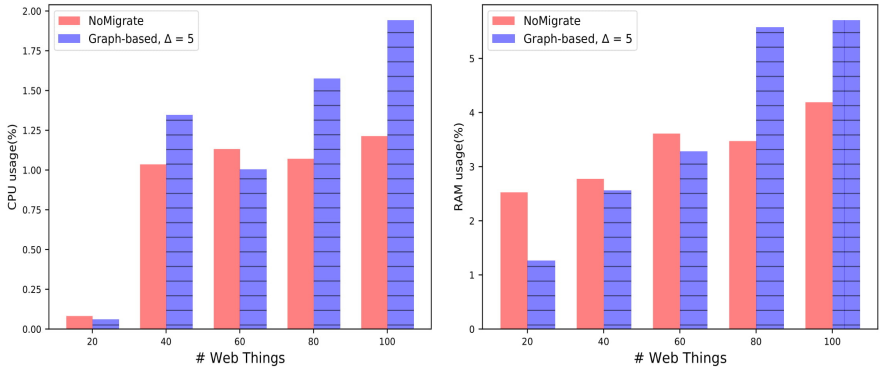
Fig. 5.13 The TF over time slots in a dynamic WoT deployment where the number of WTs is varied over time is reported in Figure 5.13(a). The *NO* over time in the IoT monitoring use case is shown in Figure 5.13(b); the processing latency for the same scenario is reported in Figure 5.13(c).

CPULoad since they both aim at allocating interacting WTs on the same node; the *NO* performance gain of the *Graph-based* policy can be tuned through the Δ parameter. For $\Delta = \infty$, the *NO* is always zero, since the WT dependency graph is likely connected (this is also due to N_C=3); as a result, all the WTs are moved to the same edge node, as better highlighted below. For $\Delta = 1$ and $\Delta = 5$, the *Graph-based* policy introduces some *NO* due to the load-balancing constraint, but still lower than the *NoMigrate*, hence it is preferable to a random allocation. The load-balancing capabilities of the six policies are investigated in Figure 5.10(b) which shows the TF metric as a function of the number of WTs; for the *Graph-based* with $\Delta = \infty$, the TF is always equal to the number of WTs in the scenario, since all the WTs are allocated to the same node. Vice versa, we can notice that, for $\Delta = 1$ and $\Delta = 5$, the TF value is always lower than the required threshold, demonstrating the effectiveness of the load-balancing mechanism. The fairness in terms of WTs translates into a better utilization of computational resources, as investigated in Figure 5.10(c). Here, the CF metric is shown for the six policies; we can notice that the *Graph-based* heuristic with $\Delta = \infty$ and $\Delta = 1$ are respectively the worst and optimal cases, once again demonstrating the versatility of the proposed approach. By comparing Figures 5.10(a) and 5.10(c), we can also appreciate that the *Graph-based* policies (with $\Delta \neq \infty$) are able to achieve a better trade-off between *NO* and CF metrics when compared to the two Greedy policies; based on the system requirements (i.e. data locality or resource utilization), the administrator can achieve the wanted performance trade-off by properly tuning the Δ parameter, whose optimal setting is clearly scenario-dependant. Figure 5.11(a) provides additional insights on the WT allocation, by showing, for the *Graph-based* policies and different values of Δ, the

average CPU utilization of each node of the cluster (denoted by the colors on each bar); the CPU values are normalized between 0 and 100%. It is easy to notice that lower values of Δ correspond to more balanced utilization of the computational resources of the cluster, while for $\Delta = \infty$ only one node is used. Finally, Figure 5.11(b) shows the *IL* metric for the six policies; we highlight that the latency is not taken into account in the optimization framework of Section 5.2.3, although delay-aware policy can be designed and installed in the Optimizer as future work. Nevertheless, the *Graph-based* with $\Delta = \infty$ overcomes the other competitors for all the configurations of WTs; this is due to the reduction of communication latency since all the WT interactions occur locally on the same node. In Figures 5.12(a), 5.12(b), 5.12(c) we expand the evaluation by considering the impact of different WT interaction amounts on the system performance. More specifically, we consider a fixed number of WTs (N_{WT}=15), while on the x-axis we vary the WT degree (N_C), i.e. the number of peers consumed by each WT, again selected in a random way. Figure 5.12(a) depicts the *NO* metric for the six policies; as expected, the amount of inter-host communication increases with the N_C values on the x-axis. The only exception is the *Graph-based* with $\Delta = \infty$: similarly to the previous analysis, the *NO* is zero since interacting WTs are allocated to separate nodes, however more than one connected component is found on the dependency graph for N_C=1 and N_C=2. As a result, the *CF* metric of the *Graph-based* with $\Delta = \infty$ shows the increasing trend of Figure 5.12(a); for N_C=1 and N_C=2, a more balanced allocation is achieved since the graph components are allocated to different nodes, while for N_C=3 the graph is fully connected hence the whole workload is allocated to the same node. Comparing 5.11(c) and 5.12(a), we can appreciate a gain how the *Graph-based* policies (with $\Delta \neq \infty$) are able to capture a better *NO-CF* tradeoff than the *NoMigrate* and greedy policies. This translates into a relevant performance gain of the *Graph-based* policies for the *IL* metric in Figure 5.12(b); for N_C=1, the latency reduction provided by the *Graph-based* policy over the *NoMigrate* is up to 37% with $\Delta = \infty$, 13% with $\Delta = 5$.

In the analysis presented so far, we considered WoT scenarios where the number of WTs is fixed at startup, hence the WT discovery process can be considered static over time. In Figures 5.12(c) and 5.13(a) we analyze the performance of M-WoT in a dynamic environment where the number of active WTs (and hence the amount of traffic and computational loads) is varying over time. More specifically, we set up the system with N_{WT}=0. Every 360 seconds, a new WT is created and added to the scenario; each WT consumes exactly one peer (N_C=1). Figure 5.12(c) shows the *NO* metric over system evolution, expressed in time slots; we remind that each time slot corresponds to the execution of the Optimizer policy, and this event occurs every 190 seconds. It is easy to notice that the *NO* metric increases significantly over time for the *NoMigrate* policy as a consequence of the creation of the new WTs, and

hence of the additional inter-host communication introduced in the system; vice versa, the *Graph-based* policies are able to adapt the WT allocation so that the *NO* minimization goal is continuously met. The adaptiveness of M-WoT to network load conditions is further demonstrated by Figure 5.13(a) which shows the TF metric over time slot; for the case of *Graph-based* with $\Delta = \infty$, the TF increases over time as a consequence of the fact that - by adding new WTs in the system - larger connected components could be created and migrated to the same node. Vice versa, the *Graph-based* policies with $\Delta = 5$ and $\Delta = 2$ dynamically allocate the WTs so that the load-balancing constraint (reflected by the Δ value) is continuously satisfied.



(a) (b)

Fig. 5.14 CPU load (Figure 5.14(a)) and RAM consumption (Figure 5.14(b)) of the Orchestrator for different numbers of deployed WTs.

Finally, we evaluated the scalability of the proposed solution by monitoring the CPU and RAM consumption on the Orchestrator and Thing Directory node. Figures 5.14(a) and 5.14(b) show our findings. The results were obtained by sampling the container metrics every second, and then averaging the results for different numbers of deployed WTs. It is possible to notice that the consumption grows linearly but it is pretty negligible even with 100 WTs. Also, the overhead introduced by the *Graph-based* policy is only slightly higher than a *NoMigrate* policy, although M-WoT must execute the WT allocation procedure and the handoff procedure detailed in section 5.2.2. Clearly, despite such positive results, the centralized Orchestrator might still become a performance bottleneck in large-scale WoT deployments; to address the issue, we can envisage the usage of a federated network of Orchestrators, each controlling a specific region of nodes. Such distributed M-WoT framework would require proper data replication, load-balancing, and gossiping mechanisms, which can be investigated as future works.

Use case Analysis

Let us consider an IoT monitoring application, which mimics the operations of the SHM use case presented in Section 4.2 and briefly introduced with 5.5(a). Specifically, we assume that a W3C WoT system has been designed to acquire and process the IoT data of a smart building. The WoT system involves three WTs:

- A *Sensing* WT, which performs data acquisition from an IoT sensor device (e.g. an accelerometer) through a Serial connection. More specifically, we assume that the *Sensing* WT can run in two modes, which differ from the sensor query frequency (qf), respectively the *Normal* mode (with 1 sample every 5 seconds) and *Warning* mode (with 1 sample every second); the mode switch (i.e. from Normal to Warning and vice versa) occurs when the last consecutive three readings are higher or lower than a static threshold; in other words, the granularity of the monitoring system is adjusted according to the detection of possible data anomalies.

- A *Processing* WT, which continuously receives the real-time measurements from the *Sensing* and applies a statistical method (i.e. the ARIMA regression) to forecast the next sensor values.

- A *Reporting* WT, which produces a notification (e.g. an alarm) based on the output of the *Processing* WT.

We abstract from the specific physical meaning of the IoT sensing values, while we focus on the capabilities of the WoT system to minimize the latency of processing especially in *Warning* mode, i.e. the time from when the data is acquired to when the forecast value is produced in the output. We consider an initial setup with two nodes (N_C=2), respectively an edge server (connected to the IoT sensor device) and a remote cloud server on the Internet. Two scenarios are configured and compared in the evaluation analysis:

- *Migration OFF*. This represents the state-of-the-art WoT environment, where the WT migration is not enabled. The *Sensing* and *Reporting* WTs are deployed on the edge node, while the *Processing* WT is deployed on the cloud due to its higher computational power.

- *Migration ON*. This corresponds to the M-WoT environment, where the *Processing* WT is configured as migratable, i.e. it can be dynamically moved on the edge or on the cloud node based on the actual sensing mode. To this purpose, we deployed in the Optimizer a scenario-specific policy that checks the number of interactions between the *Sensing* and *Processing* WTs at each time slot; in case such value is higher

than a threshold (set equal to the sf configuration in Normal Mode), the Optimizer realizes that the *Sensing* WT is working in Warning mode, and hence it migrates the *Processing* WT on the edge node, i.e. closer to the acquisition in order to minimize the communication latency. Otherwise, the *Processing* WT is allocated to the cloud node.

In the test-bed, the *Sensing* WT starts in *Normal* mode for 5 seconds, then it switches to *Warning* mode for 1 second, then again it repeats the same sequence for other two times. Figure 5.13(b) shows the NO metric over the time slots; for the *Migration OFF* configuration, the NO value at each slot is equal to the number of messages exchanged by the *Sensing* and *Processing* WTs, since they are hosted by different nodes. The peaks correspond to intervals where the *Sensing* WT switches to the *Warning* mode. It is interesting to notice that: (*i*) the *Migration ON* configuration follows the same curve of the *Migration OFF* when the inter-host communication load is below a threshold; (*ii*) the NO of the *Migration ON* is zero in correspondence of *Warning* periods, since the *Processing* WT is migrated to the edge node, and hence all the communication occurs locally. Such action impacts the utilization of computational resources on the cloud/edge nodes as well as the processing latency. we report only the latter in Figure 5.13(c). We can notice the effectiveness of the M-WoT framework in terms of latency reduction for the *Migration ON*, which is more evident during the *Warning* periods since the edge-cloud communication delay is canceled.

Part III

Conclusions

Chapter 6

Conclusions

The main goal of this thesis was to experiment and investigate the suitability of the Web of Things as a means for countering the fragmentation of the IoT, in particular by taking advantage of the recent appearance of the new W3C Web of Things standard. Despite being still under definition, the latter can be considered as one of the most promising efforts to tackle the interoperability problem in several IoT scenarios. More specifically, we addressed the following research questions:

- How to map heterogeneous IoT systems into the WoT?
- How to easily design and deploy WoT scenarios?
- Is the W3C standard effectively complete and covering all requirements of IoT scenarios? How to improve and contribute to it?

The answers to these questions have been provided in the WoT Store tool (Section 3.1), a novel platform for managing and deploying resources on the W3C WoT, with a particular focus on the dynamic discovery of Things, the possibility to easily manage their software, and to collect, control and analyze their data. The WoT store represents a concrete solution for system maintainers and users that need to deploy WoT Scenarios, fully respecting the new W3C WoT Standard. In order to speed up and facilitate the transition to the WoT Store, as well as to enable the possibility to use its registered Things in other legacy IoT systems, we proposed a software architecture (Section 3.2) to turn each Web Thing into an IoT service, and, on the contrary, to bring each IoT legacy service onto the WoT Store as a Web Thing.

That being said, the WoT Store has been validated on two different use case scenarios that can be considered as representative examples; first, it has been deployed for a heterogeneous environmental monitoring, where different Wireless Sensor Networks (WSNs) (Section 4.1) have been mapped into Web Things and brought to the WoT Store, in order to ease and

manage the sensing orchestration directly from the WoT Store interface. Second, the WoT Store has been customized for a structural health monitoring (SHM) scenario (Section 4.2), where different kinds of ad-hoc sensors collect important data that is then processed and analyzed through custom applications deployed directly in the WoT dashboards. In both cases, the effort of mapping the Sensor Networks and the ad-hoc sensors to Web Things has been precisely described. This kind of validation highlighted some lacks and improvements to be made both in the standard as well as in the official WoT implementation. In particular, we made a proposal for bringing the TSN - with strict QoS parameters - of industrial contexts (Section 5.1) in the W3C standard. Furthermore, we also designed a process for the automatic configuration of such scenarios, taking into account both the QoS requirements of the applications and the QoS capabilities offered by the Things. Finally, an additional study has been conducted for the migration of WoT services and Web Things (Section 5.2). The main objective of such research was to dynamically redistribute the WoT services on different hosts based on precise policies. This study was useful to understand that the WoT official implementation [25] needs to be augmented by adding a monitoring layer that, in our case, can be used to collect usage statistics through a service orchestrator.

The final results of this thesis can be summarized in the following: given the Web of Things paradigm, and given the new W3C WoT Standard, we deeply investigated how to concretely apply the WoT in IoT scenarios that are affected by interoperability issues, analyzing all the possible obstacles that can be encountered and hence proposing dedicated solutions. We strongly believe that the availability of support tools constituting the WoT SECO can facilitate the adoption of the W3C standard by academic and industrial communities, as well as the definition of novel use cases, and in this sense WoT Store could be considered as a reference point.

6.1 Current and future research directions on the WoT

The W3C Web of Things standard is quite new and several other studies need to be conducted on different scenarios and with different kinds of devices/things. A particular example can be represented by those devices that cannot be brought to the WoT or directly turned into Web Things. In fact, differently from sensors used in the heterogeneous sensing use case, there are devices that cannot take advantage of the *System API* layer in the *WoT Servient* to enable the communication with the WoT layer: not all low-network technologies can be mapped into Web Things directly through the System APIs. LORA [156], for instance, that we already deeply investigated in some of our previous works [157][158], uses LoRaWAN protocol to send data to the Cloud. Since data is encrypted in the communication between

device-gateway-cloud, it is not possible to capture the data directly from the sensors in order to map it to the respective Web Things. Hence, for this specific case, we are working on an auxiliary software architecture to bring this kind of device onto the WoT Store, that is able to work mainly at the Application Layer. More in detail, a dedicated component deployed in the Cloud is in charge of collecting data arriving from sensors, mapping it to semantic data and making it available for the Thing instance (this last can also be instantiated on a different layer). For this use case, Thing Descriptions for sensors must be already available and hence *exposed* by that component. LORA sensors are particularly suitable for smart agriculture scenarios, since the characteristics of such technology perfectly match the application requirements [43]. In particular, the data throughput is quite small, and both the sensing frequency and the power capability of the devices are very limited. Having success in this kind of research will also open the door to the possibility of employing the WoT Store in yet another scenario.

Directly related to this work, and aiming to enable the porting of other kinds of devices into the WoT, we designed and implemented a Micro WoT Servient, i.e., a servient capable of running on micro-controllers. This can be particularly useful for those devices that already offer networking features, like NodeMCU or ESP32, but that at the moment need to use an external device that runs a Servient, and hence require the *System API* to communicate with the WoT layer. Instead, through a native servient running on the micro-controller, this can natively expose and consume Things, avoiding intermediate steps in the communication. Clearly, the biggest challenges in this research are related to the hardware architecture and the limited capabilities - both in terms of power and computational resources - of the devices. In this sense, an accurate and precise design work has been made to adapt the functionalities of the Servient to such hardware limitations.

Finally, another interesting improvement we are currently working on, especially for the migration work presented in Section 5.2, is represented by the *dynamic aggregator*. This is a component in charge of collecting data directly from the Things, with the additional duty of aggregating it based on different kinds of policies. Furthermore, once it has obtained the data, it also saves it to the *persistor* component - basically, to a database. It is interesting to note that, in particular scenarios - like the SHM presented in Section 4.2, it would be very useful and efficient to dynamically move the aggregator on different layers, depending on the current conditions. This would allow analysts to be informed as soon as the danger has been detected on the edge and to observe aggregated data on the cloud, implying also a faster response of the system and a considerable reduction in terms of data exchange.

References

[1] Antonio J. Jara, Alex C. Olivieri, Yann Bocchi, Markus Jung, Wolfgang Kastner, and Antonio F. Skarmeta. Semantic Web of things: An analysis of the application semantics for the IoT moving towards the IoT convergence. *International Journal of Web and Grid Services*, 10(2-3):244–272, 2014. ISSN 17411114. doi: 10.1504/ijwgs.2014.060260.

[2] W3C Working Group. WoT Reference Architecture (Proposed Recommendation 30 January 2020), 2019. URL http://www.w3.org/TR/wot-architecture/.

[3] Dominique D Guinard and Vlad M Trifa. *Building the web of things*, volume 3. Manning Publications Shelter Island, 2016.

[4] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE communications surveys & tutorials*, 17(4):2347–2376, 2015.

[5] Beniamino Di Martino, Massimiliano Rak, Massimo Ficco, Antonio Esposito, Salvatore Augusto Maisto, and Stefania Nacchia. Internet of things reference architectures, security and interoperability: A survey. *Internet of Things*, 1:99–112, 2018.

[6] Federico Montori, Luca Bedogni, Marco Di Felice, and Luciano Bononi. Machine-to-machine wireless communication technologies for the internet of things: Taxonomy, comparison and open issues. *Pervasive and Mobile Computing*, 50:56–81, 2018.

[7] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. Interoperability in internet of things: Taxonomies and open challenges. *Mobile Networks and Applications*, 24(3):796–809, 2019.

[8] Emiliano Sisinni, Abusayeed Saifullah, Song Han, Ulf Jennehag, and Mikael Gidlund. Industrial internet of things: Challenges, opportunities, and directions. *IEEE Transactions on Industrial Informatics*, 14(11):4724–4734, 2018.

[9] Muhammad Intizar Ali. From Raw Data to Smart Manufacturing: AI and semantic Web of Things for Industry 4.0. *IEEE Intelligent Systems*, 33(August):79–86, 2018. doi: 10.1109/MIS.2018.043741325.

[10] Interoperability and the Internet of Things, 2017. URL https://www.ndpanalytics.com/report-interoperability-and-iot.

[11] James Manyika. *The Internet of Things: Mapping the value beyond the hype*. McKinsey Global Institute, 2015.

[12] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990. doi: 10.1109/IEEESTD.1990.101064.

[13] Hasan Derhamy, Jens Eliasson, Jerker Delsing, and Peter Priller. A survey of commercial frameworks for the internet of things. In *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–8. IEEE, 2015.

[14] Open Connectivity Foundation (OCF). IoTivity, 2015. URL https://iotivity.org/.

[15] AllSeen Alliance. Allseen alliance wiki. URL https://wiki.allseenalliance.org/.

[16] Arrowhead Framework, 2015.

[17] Mina Younan, Sherif Khattab, and Reem Bahgat. Wotsf: A framework for searching in the web of things. In *Proceedings of the 10th International Conference on Informatics and Systems*, pages 278–285, 2016.

[18] Andreas Kamilaris and Muhammad Intizar Ali. Do "web of things platforms" truly follow the web of things? In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 496–501. IEEE, 2016.

[19] Michele Ruta, Floriano Scioscia, Agnese Pinto, Eugenio Di Sciascio, Filippo Gramegna, Saverio Ieva, and Giuseppe Loseto. Resource annotation, dissemination and discovery in the semantic web of things: a coap-based framework. In *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pages 527–534. IEEE, 2013.

[20] SOSA (Sensor, Observation, Sample, and Actuator) W3C onthology. URL https://www.w3.org/TR/vocab-ssn/.

[21] Federica Paganelli, Stefano Turchi, and Dino Giuli. A Web of Things Framework for RESTful Applications and Its Experimentation in a Smart City. *IEEE Systems Journal*, 10(4):1412–1423, 2016. ISSN 19379234. doi: 10.1109/JSYST.2014.2354835.

[22] Luca Mainetti, Vincenzo Mighali, and Luigi Patrono. A software architecture enabling the web of things. *IEEE Internet of Things Journal*, 2(6):445–454, 2015. ISSN 23274662. doi: 10.1109/JIOT.2015.2477467.

[23] Enzo Mingozzi, Giacomo Tanganelli, and Carlo Vallati. Coap proxy virtualization for the web of things. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 577–582. IEEE, 2014.

[24] W3C. WoT Reference Architecture (W3C Recommendation 9 April 2020), 2020. URL http://www.w3.org/TR/wot-architecture/.

[25] W3C. Eclipse Thingweb node-wot, 2020. URL https://github.com/eclipse/thingweb.node-wot.

[26] Ioan Szilagyi and Patrice Wira. Ontologies and semantic Web for the internet of things - A survey. In *IECON Proceedings (Industrial Electronics Conference)*, pages 6949–6954, 2016. ISBN 9781509034741. doi: 10.1109/IECON.2016.7793744.

[27] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[28] Auto-ID Labs, 2020. URL http://www.autoidlabs.org/.

[29] M Presser and A Gluhak. The internet of things: Connecting the real world with the digital world, eurescom mess@ ge–the magazine for telecom insiders, vol. 2, 2009, 2012.

[30] Maarten Botterman. Internet of things: an early reality of the future internet. In *Workshop Report, European Commission Information Society and Media*, volume 15. sn, 2009.

[31] Lara Srivastava, T Kelly, et al. The internet of things. *International Telecommunication Union, Tech. Rep*, 7, 2005.

[32] IPSO Alliance. Internet protocol for smart objects (ipso).

[33] David Culler, Samita Chakrabarti, and IP Infusion. 6lowpan: Incorporating ieee 802.15. 4 into the ip architecture. *IPSO Alliance, White paper*, 2009.

[34] Ioan Toma, Elena Simperl, and Graham Hench. A joint roadmap for semantic technologies and the internet of things. In *Proceedings of the Third STI Roadmapping Workshop, Crete, Greece*, volume 1, pages 140–53, 2009.

[35] Artem Katasonov, Olena Kaykova, Oleksiy Khriyenko, Sergiy Nikitin, and Vagan Y Terziyan. Smart semantic middleware for the internet of things. *Icinco-Icso*, 8: 169–178, 2008.

[36] Iñaki Vázquez. Social devices: Semantic technology for the internet of things. *Week@ ESI, Zamudio, Spain*, 2009.

[37] Li Da Xu, Wu He, and Shancang Li. Internet of things in industries: A survey. *IEEE Transactions on industrial informatics*, 10(4):2233–2243, 2014.

[38] Min Zhang, Tao Yu, and Guo Fang Zhai. Smart transport system based on "the internet of things". In *Applied mechanics and materials*, volume 48, pages 1073–1076. Trans Tech Publ, 2011.

[39] Sameer Kumar, Eric Swanson, and Thuy Tran. Rfid in the healthcare supply chain: usage and application. *International Journal of Health Care Quality Assurance*, 2009.

[40] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *IEEE Internet of Things journal*, 1(1):22–32, 2014.

[41] Bric 2018 inail mac4pro project, https://site.unibo.it/mac4pro/it, 2019.

[42] Simon Elias Bibri and John Krogstie. The emerging data–driven smart city and its innovative applied solutions for sustainability: the cases of london and barcelona. *Energy Informatics*, 3(1):1–42, 2020.

[43] Swamp EU-BR H2020 Project, 2020. URL http://swamp-project.org.

[44] Hans Van Der Veer and Anthony Wiles. Achieving Technical Interoperability: the ETSI Approach. *European Telecommunications Standards Institute*, (3):29, 2008. URL https://portal.etsi.org/CTI/Downloads/ETSIApproach/IOPwhitepaperEdition3final.pdf.

[45] Martin Serrano, Payam Barnaghi, Francois Carrez, Philippe Cousin, Ovidiu Vermesan, and Peter Friess. Internet of things iot semantic interoperability: Research challenges, best practices, recommendations and next steps. , European Research Cluster on the Internet of Things, 2015.

[46] Arrowhead - Ahead the Future., 2018. URL https://www.arrowhead.eu.

[47] Big IoT - Bridging the Interoperability Gap of the Internet of Things. URL http://big-iot.eu.

[48] Wise IoT - Worldwide Interoperability for Semantics IoT. URL http://wise-iot.eu/en/home/.

[49] IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN). URL https://tools.ietf.org/html/rfc8138.

[50] Garvita Bajaj, Rachit Agarwal, Pushpendra Singh, Nikolaos Georgantas, and Valerie Issarny. A study of existing ontologies in the iot-domain. *arXiv preprint arXiv:1707.00112*, 2017.

[51] Luca Roffia, Paolo Azzoni, Cristiano Aguzzi, Fabio Viola, Francesco Antoniazzi, and Tullio Salmon Cinotti. Dynamic linked data: A sparql event processing architecture. *Future Internet*, 10(4):36, 2018.

[52] Andrés García Mangas and Francisco José Suárez Alonso. WOTPY: A framework for web of things applications. *Computer Communications*, 147(September):235–251, 2019. ISSN 1873703X. doi: 10.1016/j.comcom.2019.09.004. URL https://doi.org/10.1016/j.comcom.2019.09.004.

[53] Open Connectivity Foundation (OCF). Ocf core specification, version 2.0.1. , Open Connectivity Foundation (OCF), 2019. URL https://openconnectivity.org/specs/OCF_Core_Specification_v2.0.1.pdf.

[54] Open Mobile Alliance (OMA). Lightweight machine to machine technical specification, version 1.0.2. , Open Mobile Alliance (OMA), 2019. URL http://www.openmobilealliance.org/release/LightweightM2M/V1_0_2-20180209-A/OMA-TS-LightweightM2M-V1_0_2-20180209-A.pdf.

[55] European Telecommunications Standards Institute (ETSI). onem2m: Functional architecture, version 2.10.0. , European Telecommunications Standards Institute (ETSI), 2016. URL https://www.etsi.org/deliver/etsi_ts/118100_118199/118101/02.10.00_60/ts_118101v021000p.pdf.

[56] Carlos Kamienski, Marc Jentsch, Markus Eisenhauer, Jussi Kiljander, Enrico Ferrera, Peter Rosengren, Jesper Thestrup, Eduardo Souto, Walter S. Andrade, and Djamel Sadok. Application development for the Internet of Things: A context-aware mixed criticality systems development platform. *Computer Communications*, 104(2017): 1–16, 2017. ISSN 01403664. doi: 10.1016/j.comcom.2016.09.014. URL http://dx.doi.org/10.1016/j.comcom.2016.09.014.

[57] Lu Tan and Neng Wang. Future internet: The internet of things. In *2010 3rd international conference on advanced computer theory and engineering (ICACTE)*, volume 5, pages V5–376. IEEE, 2010.

[58] Behailu Negash, Tomi Westerlund, and Hannu Tenhunen. Towards an interoperable Internet of Things through a web of virtual things at the Fog layer. *Future Generation Computer Systems*, 91:96–107, feb 2019. ISSN 0167739X. doi: 10.1016/j.future.2018.07.053.

[59] Timothy Chou. *Precision-Principles, Practices and Solutions for the Internet of Things*. McGraw-Hill Education, 2017.

[60] Dave Raggett. The web of things: Challenges and opportunities. *Computer*, 48(5): 26–32, 2015. ISSN 00189162. doi: 10.1109/MC.2015.149. URL www.w3.org/communitywww.dali-ag.org.

[61] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, and Bill Serra. People, places, things: Web presence for the real world. *HP Laboratories Technical Report*, (16):19–28, 2000.

[62] Erik Wilde. Putting Things to REST. *Transport*, 15(November):1 – 13, 2007. URL http://dret.net/netdret/publications{#}wil07n.

[63] Dominique D Guinard and Vlad M Trifa. What is the web of things? ApacheCon Europe, Presentation, April 2008. URL https://webofthings.org/2017/04/08/what-is-the-web-of-things/.

[64] Dominique Guinard and Vlad Trifa. Towards the Web of Things : Web Mashups for Embedded Devices. *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009)*, pages 1–8, 2009. doi: 10.1.1.155.3238.

[65] Dominique Guinard, Iulia Ion, and Simon Mayer. In search of an internet of things service architecture: Rest or ws-*? a developers' perspective. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 326–337. Springer, 2011.

[66] Foughali Karim, Fathalah Karim, and Ali Frihida. Monitoring system using web of things in precision agriculture. In *Procedia Computer Science*, volume 110, pages 402–409. Elsevier B.V., 2017. doi: 10.1016/j.procs.2017.06.083.

[67] Tein-Yaw Chung, Ibrahim Mashal, Osama Alsaryrah, Van Huy, Wen-Hsing Kuo, and Dharma P Agrawal. Social Web of Things: A Survey. 2013. doi: 10.1109/.101.

[68] Andrei Ciortea, Olivier Boissier, Antoine Zimmermann, and Adina Magda Florea. Reconsidering the social web of things. Position paper. *UbiComp 2013 Adjunct - Adjunct Publication of the 2013 ACM Conference on Ubiquitous Computing*, pages 1535–1544, 2013. doi: 10.1145/2494091.2497587.

[69] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 284(5):34–43, 2001.

[70] Jörg Heuer, Johannes Hund, and Oliver Pfaff. Toward the web of things: Applying web technologies to the physical world. *Computer*, 48(5):34–42, 2015. ISSN 00189162. doi: 10.1109/MC.2015.152.

[71] Sujith Samuel Mathew, Yacine Atif, Quan Z. Sheng, and Zakaria Maamar. Web of things: Description, discovery and integration. *Proceedings - 2011 IEEE International Conferences on Internet of Things and Cyber, Physical and Social Computing, iThings/CPSCom 2011*, (Contribution 3):9–15, 2011. doi: 10.1109/iThings/CPSCom.2011.165.

[72] Andreas Kamilaris, Semih Yumusak, and Muhammad Intizar Ali. WOTS2E: A search engine for a Semantic Web of Things. *2016 IEEE 3rd World Forum on Internet of Things, WF-IoT 2016*, pages 436–441, 2017. doi: 10.1109/WF-IoT.2016.7845448.

[73] Michael Mrissa, Lionel Médini, Jean-Paul Jamont, Nicolas Le Sommer, and Jérôme Laplace. Building Internet of Things Software An Avatar Architecture for the Web of Things. Technical report, 2015. URL https://ifttt.com/].

[74] Vlad Trifa, Dominique Guinard, and David Carrera. Web Thing Model, 2015. URL https://www.w3.org/Submission/wot-model/.

[75] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002. ISSN 1533-5399.

[76] T. Bray (Ed.). The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259 (Internet Standard), December 2017. ISSN 2070-1721.

[77] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[78] Sebastian Käbisch, Takuki Kamiya, Michael McCool, Victor Charpenay, and Matthias Kovatsch. Web of things (wot) thing description. W3C recommendation, April 2020. https://www.w3.org/TR/wot-thing-description.

[79] Gregg Kellogg, Pierre-Antoine Champin, and Dave Longley. JSON-ld 1.1. W3C proposed recommendation, May 2020. https://www.w3.org/TR/json-ld11/.

[80] Donald A. Norman. *The Design of Everyday Things*. Basic Books, Inc., USA, 2002. ISBN 9780465067107.

[81] Roy T. Fielding. A little rest and relaxation. ApacheCon Europe, Presentation, April 2008. URL https://www.slideshare.net/royfielding/a-little-rest-and-relaxation.

[82] Dominique Guinard, Vlad Trifa, Thomas Pham, and Olivier Liechti. Towards physical mashups in the web of things. *INSS2009 - 6th International Conference on Networked Sensing Systems*, pages 196–199, 2009. doi: 10.1109/INSS.2009.5409925.

[83] Edward A Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, 2008.

[84] Dominique Guinard, Vlad Trifa, and Erik Wilde. A resource oriented architecture for the web of things. *2010 Internet of Things, IoT 2010*, 2010. doi: 10.1109/IOT.2010.5678452.

[85] Cesare Pautasso and Erik Wilde. Why is the web loosely coupled? a multi-faceted metric for service design. In *Proceedings of the 18th international conference on World wide web*, pages 911–920, 2009.

[86] Vlad Trifa, Samuel Wieland, Dominique Guinard, and T.M. Bohnert. Design and implementation of a gateway for web-based interaction and management of embedded devices. *Proceedings of the 2nd International Workshop on Sensor Network Engineering (IWSNE 09)*, pages 1–14, 2009. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.155.4806{&}rep=rep1{&}type=pdf.

[87] Dominique Guinard, Mathias Mueller, and Jacques Pasquier-Rocha. Giving RFID a REST: Building a web-enabled EPCIS. *2010 Internet of Things, IoT 2010*, 2010. doi: 10.1109/IOT.2010.5678447.

[88] I. Nadim, Y. Elghayam, and A. Sadiq. Semantic discovery architecture for dynamic environments of Web of Things. *Proceedings - 2018 International Conference on Advanced Communication Technologies and Networking, CommNet 2018*, pages 1–6, 2018. doi: 10.1109/COMMNET.2018.8360269.

[89] Simon Mayer and Dominique Guinard. An extensible discovery service for smart things. In *Proceedings of the Second International Workshop on Web of Things*, pages 1–6, 2011.

[90] Michele Ruta, Floriano Scioscia, and Eugenio Di Sciascio. Enabling the semantic web of things: Framework and architecture. *Proceedings - IEEE 6th International Conference on Semantic Computing, ICSC 2012*, pages 345–347, 2012. doi: 10.1109/ICSC.2012.42.

[91] Muhammad Shoaib, Wang-Cheol Song, Rashid Ahamd, and Do-Hyeun Kim. Architecture of push service based on sns for sharing sensor information. In *Green and Smart Technology with Sensor Applications*, pages 342–346. Springer, 2011.

[92] M Baqer. Enabling collaboration and coordination of wireless sensor networks via social networks. In *2010 6th IEEE International Conference on Distributed Computing in Sensor Systems Workshops (DCOSSW)*, pages 1–2. IEEE, 2010.

[93] Thomas Schmid and Mani B Srivastava. Exploiting social networks for sensor data sharing with senseshare. 2007.

[94] Dominique Guinard, Mathias Fischer, and Vlad Trifa. Sharing using social networks in a composable Web of Things. *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops, PERCOM Workshops 2010*, pages 702–707, 2010. doi: 10.1109/PERCOMW.2010.5470524.

[95] Andreas Kamilaris and Andreas Pitsillides. Social networking of the smart home. *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC*, pages 2632–2637, 2010. doi: 10.1109/PIMRC.2010.5671783.

[96] Chunhong Zhang, Cheng Cheng, and Yang Ji. Architecture design for social web of things. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2012. doi: 10.1145/2346604.2346608.

[97] Antonio Pintus, Davide Carboni, and Andrea Piras. Paraimpu: A platform for a social Web of Things. *WWW'12 - Proceedings of the 21st Annual Conference on World Wide Web Companion*, pages 401–404, 2012. doi: 10.1145/2187980.2188059.

[98] Mari Carmen Domingo. A context-aware service architecture for the integration of body sensor networks and social networks through the ip multimedia subsystem. *IEEE Communications Magazine*, 49(1):102–108, 2011.

[99] Md Abdur Rahman, Abdulmotaleb El Saddik, and Wail Gueaieb. Data visualization: From body sensor network to social networks. In *2009 IEEE International Workshop on Robotic and Sensors Environments*, pages 157–162. IEEE, 2009.

[100] Andrés García Mangas and Francisco José Suárez Alonso. Wotpy: A framework for web of things applications. *Computer Communications*, 147:235–251, 2019.

[101] Youngmin Ji, Kisu Ok, and Woo Suk Choi. Web of things based iot standard interworking test case: demo abstract. In *Proceedings of the 5th Conference on Systems for Built Environments*, pages 182–183, 2018.

[102] Benjamin Klotz, Soumya Kanti Datta, Daniel Wilms, Raphael Troncy, and Christian Bonnet. A car as a semantic web thing: Motivation and demonstration. In *2018 Global Internet of Things Summit, GIoTS 2018*. Institute of Electrical and Electronics Engineers Inc., nov 2018. ISBN 9781538664513. doi: 10.1109/GIOTS.2018.8534533.

[103] Michael McCool and Elena Reshetova. Distributed Security Risks and Opportunities in the W3C Web of Things. Internet Society, aug 2018. doi: 10.14722/diss.2018.23008.

[104] Michele Blank, Sebastian Kaebisch, Haifa Lahbaiel, and Harald Kosch. Role models and lifecycles in IoT and their impact on the W3C WOT thing description. *ACM International Conference Proceeding Series*, 2018. doi: 10.1145/3277593.3277908.

[105] JV Joshua, DO Alao, SO Okolie, and O Awodele. Software ecosystem: features, benefits and challenges. *International Journal of Advanced Computer Science and Applications*, 2, 2013.

[106] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[107] Andrea Capponi, Claudio Fiandrino, Burak Kantarci, Luca Foschini, Dzmitry Kliazovich, and Pascal Bouvry. A survey on mobile crowdsensing systems: Challenges, solutions, and opportunities. *IEEE communications surveys & tutorials*, 21(3):2419–2465, 2019.

[108] Jerker Delsing. *Iot automation: Arrowhead framework*. CRC Press, 2017.

[109] Hongyu Pei Breivold and Magnus Larsson. Component-based and service-oriented software engineering: Key concepts and principles. In *33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007)*, pages 13–20. IEEE, 2007.

[110] Pal Varga, Fredrik Blomstedt, Luis Lino Ferreira, Jens Eliasson, Mats Johansson, Jerker Delsing, and Iker Martínez de Soria. Making system of systems interoperable–the core components of the arrowhead framework. *Journal of Network and Computer Applications*, 81:85–95, 2017.

[111] Pál Varga and Csaba Hegedus. Service interaction through gateways for inter-cloud collaboration within the arrowhead framework. *5th IEEE WirelessVitae, Hyderabad, India*, 2015.

[112] Youngmin Ji, Kisu Ok, and Woo Suk Choi. Web of things based IoT standard interworking test case. pages 182–183. Association for Computing Machinery (ACM), nov 2018. doi: 10.1145/3276774.3281012.

[113] Charles R Farrar and Keith Worden. An introduction to structural health monitoring. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 365(1851):303–315, 2007.

[114] Paolo Barsocchi, Pietro Cassará, Fabio Mavilia, and Daniele Pellegrini. Sensing a city's state of health: Structural monitoring system by internet-of-things wireless sensing devices. *IEEE Consumer Electronics Magazine*, 7(march):22–31, 2018.

[115] C. Jr. Arcadius Tokognon, Bin Gao, Gui Yun Tian, and Yan Yan. Structural Health Monitoring Framework Based on Internet of Things: A Survey. *IEEE Internet of Things Journal*, 4(3):619–635, 2017.

[116] Limin Sun, Zhiqiang Shang, Ye Xia, Sutanu Bhowmick, and Satish Nagarajaiah. Review of bridge structural health monitoring aided by big data and artificial intelligence: From condition assessment to damage detection. *Journal of Structural Engineering*, 146(5):04020073, 2020.

[117] Limin Sun Sun, Zhiqiang Shang, Ye Xia, Sutanu Bhowmick, and Satish Nagarajaiah. Review of bridge structural health monitoring aided by big data and artificial intelligence: From condition assessment to damage detection. *Journal of Structural Engineering*, 146(5), 2020.

[118] P. Pierleoni, M. Conti, A. Belli, L. Palma, L. Incipini, L. Sabbatini, S. Valenti, M. Mercuri, and R. Concetti. IoT Solution based on MQTT Protocol for Real-Time Building Monitoring. *Proceeding of the 2019 IEEE 23rd International Symposium on Consumer Technologies (ISCT 2019)*, pages 57–62, 2019.

[119] Yizheng Liao, Mark Mollineaux, Richard Hsu, Rebekah Bartlett, Anubhav Singla, Adnan Raja, Ravneet Bajwa, and Ram Rajagopal. SnowFort: An open source wireless sensor network for data analytics in infrastructure and environmental monitoring. *IEEE Sensors Journal*, 14(12):4253–4263, 2014.

[120] F. Lamonaca, C. Scuro, P. F. Sciammarella, R. S. Olivito, D. Grimaldi, and D. L. Carní. A layered iot-based architecture for a distributed structural health monitoring system. *Acta IMEKO*, 8(2):45–52, 2019.

[121] Md Anam Mahmud, Kyle Bates, Trent Wood, Ahmed Abdelgawad, and Kumar Yelamarthi. A complete Internet of Things (IoT) platform for Structural Health Monitoring (SHM). *Proceedings of the IEEE World Forum on Internet of Things (WF-IoT 2018)*, 2018-January:275–279, 2018.

[122] Rih-Teng Wu and Mohammad Reza Jahanshahi. Data fusion approaches for structural health monitoring and system identification: Past, present, and future. *Structural Health Monitoring*, 19(2):552–586, 2020. doi: 10.1177/1475921718798769.

[123] Nicola Testoni, Cristiano Aguzzi, Valentina Arditi, Federica Zonzini, Luca De Marchi, Alessandro Marzani, and Tullio Salmon Cinotti. A sensor network with embedded data processing and data-to-cloud capabilities for vibration-based real-time shm. *Journal of Sensors*, 2018, 2018.

[124] Federica Zonzini, Michelangelo Maria Malatesta, Denis Bogomolov, Nicola Testoni, Alessandro Marzani, and Luca De Marchi. Vibration-based shm with up-scalable and low-cost sensor networks. *IEEE Transactions on Instrumentation and Measurement*, 2020.

[125] Shu Li, Jeong Geun Kim, Doo Hee Han, and Kye San Lee. A survey of energy-efficient communication protocols with qos guarantees in wireless multimedia sensor networks. *Sensors*, 19(1):199, 2019.

[126] Matthias Kovatsch, Ryuichi Matsukura, Michael Lagally, Toru Kawaguchi, Kunihiko Toumura, and Kazuo Kajimoto. Web of things (wot) architecture. W3C recommendation, April 2020. https://www.w3.org/TR/wot-architecture/.

[127] Plattform Industrie 4.0. Details of the asset administration shell. Specification, BMWi, November 2018.

[128] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC Unified Architecture*. Springer-Verlag GmbH, Germany, 2009. ISBN 9783540688983.

[129] OPC Foundation. Initiative: Field level communications (flc) – opc foundation extends opc ua including tsn down to field level. Brochure, OPC Foundation, 2019.

[130] IEEE. Ieee standard for local and metropolitan area network–bridges and bridged networks. IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014), 2018.

[131] Jan Eveleens. Ethernet avb overview and status. In *SMPTE*, pages 1–11, Hollywood, CA, USA, 2014.

[132] R. Enns (Ed.), M. Bjorklund (Ed.), J. Schoenwaelder (Ed.), and A. Bierman (Ed.). Network Configuration Protocol (NETCONF). RFC 6241 (Proposed Standard), June 2011. ISSN 2070-1721. Updated by RFCs 7803, 8526.

[133] M. Bjorklund (Ed.). The YANG 1.1 Data Modeling Language. RFC 7950 (Proposed Standard), August 2016. ISSN 2070-1721. Updated by RFCs 8342, 8526.

[134] Astrit Ademaj. Ptcc - tsn endstation centralized configuration interface for opc ua pubsub systems. IEC/IEEE 60802 Contribution, September 2019. URL http://www.ieee802.org/1/files/public/docs2019/60802-ademaj-PTCC-0918-v05.pdf.

[135] Martin Boehm, Jannis Ohms, Manish Kumar, Olaf Gebauer, and Diederich Wermser. Dynamic real-time stream reservation for ieee 802.1 time-sensitive networks with openflow. In *Proc. ICAIIT*, pages 7–12, Koethen, Germany, 2020.

[136] A. Bierman, M. Bjorklund, and K. Watsen. RESTCONF Protocol. RFC 8040 (Proposed Standard), January 2017. ISSN 2070-1721. Updated by RFC 8527.

[137] Hansong Xu, Wei Yu, David Griffith, and Nada Golmie. A survey on industrial internet of things: A cyber-physical systems perspective. *IEEE Access*, 6:78238–78259, 2018.

[138] Fatemeh Jalali, Timothy Lynar, Olivia J. Smith, Ramachandra Rao Kolluri, Claire V. Hardgrove, Nick Waywood, and Frank Suits. Dynamic Edge Fabric EnvironmenT: Seamless and Automatic Switching among Resources at the Edge of IoT Network and Cloud. *Proceedings of the IEEE International Conference on Edge Computing (EDGE 2019)*, pages 77–86, 2019.

[139] Cheng Zhang and Zixuan Zheng. Task migration for mobile edge computing using deep reinforcement learning. *Future Generation Computer Systems*, 96:111–118, 2019. ISSN 0167739X. URL https://doi.org/10.1016/j.future.2019.01.059.

[140] X. Sun and N. Ansari. EdgeIoT: Mobile Edge Computing for the Internet of Things. *IEEE Communications Magazine*, 54(12):22–29, 2016. ISSN 01636804.

[141] S. Wang, J. Xu, N. Zhang, and Y. Liu. A Survey on Service Migration in Mobile Edge Computing. *IEEE Access*, 6:23511–23528, 2018. ISSN 21693536.

[142] Kiryong Ha, Yoshihisa Abe, Zhuo Chen, Wenlu Hu, Brandon Amos, Padmanabhan Pillai, and Mahadev Satyanarayanan. Adaptive vm handoff across cloudlets. *Technical Report CMU-CS-15-113*, 2015.

[143] Carlo Puliafito, Enzo Mingozzi, Francesco Longo, Antonio Puliafito, and Omer Rana. Fog computing for the internet of things: A survey. *ACM Transactions on Internet Technology (TOIT)*, 19(2):1–41, 2019.

[144] Wei Bao, Dong Yuan, Zhengjie Yang, Shen Wang, Wei Li, Bing Bing Zhou, and Albert Y Zomaya. Follow me fog: Toward seamless handover timing schemes in a fog computing environment. *IEEE Communications Magazine*, 55(11):72–78, 2017.

[145] P. Bellavista, A. Zanni, and M. Solimando. A migration-enhanced edge computing support for mobile devices in hostile environments. In *Proceedings of the 13th International Wireless Communications and Mobile Computing Conference (IEEE IWCMC 2017)*, pages 957–962, 2017.

[146] Corentin Dupont, Raffaele Giaffreda, and Luca Capra. Edge computing in IoT context: Horizontal and vertical Linux container migration. *Proceedings of the Global Internet of Things Summit (GIoTS 2017)*, pages 2–5, 2017. doi: 10.1109/GIOTS.2017.8016218.

[147] Shiqiang Wang, Rahul Urgaonkar, Ting He, Murtaza Zafer, Kevin Chan, and Kin K Leung. Mobility-induced service migration in mobile micro-clouds. In *2014 IEEE military communications conference*, pages 835–840. IEEE, 2014.

[148] Carlo Puliafito, Enzo Mingozzi, Carlo Vallati, Francesco Longo, and Giovanni Merlino. Companion fog computing: supporting things mobility through container migration at the edge. *Proceedings of the 2018 IEEE International Conference on Smart Computing, (IEEE SMARTCOMP 2018)*, pages 97–105, 2018. doi: 10.1109/SMARTCOMP.2018.00079.

[149] Hadeel Abdah, Joao Paulo Barraca, and Rui L. Aguiar. QoS-aware service continuity in the virtualized edge. *IEEE Access*, 7:51570–51588, 2019. ISSN 21693536.

[150] Kai Kientopf, Saleem Raza, Simon Lansing, and Mesut Güneş. Service management platform to support service migrations for IoT smart city applications. *Proceedings of the IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (IEEE PIMRC 2018)*, 2017-Octob:1–5, 2018.

[151] Flávio Ramalho and Augusto Neto. Virtualization at the network edge: A performance comparison. In *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–6. IEEE, 2016.

[152] Roberto Morabito and Nicklas Beijar. Enabling Data Processing at the Network Edge through Lightweight Virtualization Technologies. *Proceedings of the IEEE International Conference on Sensing, Communication and Networking, SECON Workshops 2016*, (607728):1–6, 2016. doi: 10.1109/SECONW.2016.7746807.

[153] Kumseok Jung, Julien Gascon-Samson, and Karthik Pattabiraman. Demo: ThingsMigrate - Platform-independent live-migration of javascript processes. *Proceedings of the 2018 3rd ACM/IEEE Symposium on Edge Computing (SEC 2018)*, pages 356–358, 2018. doi: 10.1109/SEC.2018.00044.

[154] Fabio Bellifemine, Giovanni Caire, Agostino Poggi, and Giovanni Rimassa. Jade: A software framework for developing multi-agent applications. lessons learned. *Information and Software Technology*, 50:10–21, 01 2008.

[155] Luis Alonso, Javier Barbarán, Jaime Chen, Manuel Díaz, Luis Llopis, and Bartolomé Rubio. Middleware and communication technologies for structural health monitoring of critical infrastructures: A survey. *Computer Standards and Interfaces*, 56(March 2017):83–100, 2018.

[156] LoRa Alliance. URL https://lora-alliance.org/.

[157] Luca Sciullo, Federico Fossemo, Angelo Trotta, and Marco Di Felice. Locate: a lora-based mobile emergency management system. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2018.

[158] Luca Sciullo, Angelo Trotta, and Marco Di Felice. Design and performance evaluation of a lora-based mobile emergency management system (locate). *Ad Hoc Networks*, 96:101993, 2020.