Alma Mater Studiorum - Università di Bologna

DOTTORATO DI RICERCA IN

MONITORAGGIO E GESTIONE DELLE STRUTTURE E DELL'AMBIENTE - SEHM2

Ciclo 33

Settore Concorsuale: 09/H1 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

Settore Scientifico Disciplinare: ING-INF/05 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

STRUCTURAL HEALTH AND ENVIRONMENTAL MONITORING WITH THE WEB OF THINGS

Presentata da: Cristiano Aguzzi

Coordinatore Dottorato

Alessandro Marzani

Supervisore

Tullio Salmon Cinotti

Co-supervisore

Luca De Marchi

Esame finale anno 2021

This page intentionally left blank.

Keywords:

Web Of Things

Structural Health Monitoring

Smart Agriculture

This page intentionally left blank.

To whom have devoted their lives to science

Once you stop learning,
you start dying

- Albert Einstein -

Abstract

Structural health and Environmental monitoring are recently benefiting from the advancement in the digital industry. Thanks to the emergence of the Internet of Things (IoT) paradigm, monitoring systems are increasing their functionalities and reducing development costs. However, they are affected by a strong fragmentation in the solution proposed and technologies employed. This stale the overall benefits of the adoption of IoT frameworks or IoT devices since it limits the reusability and portability of the chosen platform. As in other IoT contexts, also the structural health and environmental monitoring domain is suffering from the negative effects of, what is called, an interoperability problem. Recently the World Wide Web Consortium (W3C) is joining the race in the definition of a standard for IoT unifying different solutions below a single paradigm. This new shift in the industry is called Web Of Things (WoT) or in short WoT. Together with other W3C technologies of the Semantic Web, the Web of Things unifies different protocols and data models thanks to a descriptive machine-understandable document called the Thing Description. This work wants to explore how this new paradigm can improve the quality of structural health and environmental monitoring applications. The goal is to provide a monitoring infrastructure solely based on WoT and Semantic technologies. The architecture is later tested and applied on two concrete use-cases taken from the industrial structural monitoring and the smart farming domains. Finally, this thesis proposes a layered structure for organizing the knowledge design of the two applications, and it provides evaluation comments on the results obtained.

Abstract (Italian)

Le pratiche di monitoraggio strutturale e dell'ambiente stanno recemente beneficiando degli avanzamenti nella industria digitale. Grazie alla nascita di tecnologie basate sull'Internet of Things (IoT), i sistemi di monitoraggio hanno migliorato le loro funzionalità base e ridotto i costi di svilippo. Nonostante ciò, queste soluzioni hardware e software sono affette da una forte fragmentazione sia riguardo ai tipi dispositivo sia alle tecnologie usate. Questa fenomeno fa si che i benifici ottenuti utilizzando tecnologie IoT si riducano poichè spesso tali soluzioni mancano di portabilità e adattabilità. Come in altri contesti IoT, anche nel monitoraggio strutturale e ambintale possiamo incorre nel problema tipico della mancanza di interoperabilità tra diverse piattaforme. Recemenete il World Wide Web Consortium (W3C) ha iniziato a lavorare ad uno standard per unificare le maggiori tecnologie IoT sotto un unico paradigma. Questo nuova corrente è chiamata il Web of Things o in breve WoT. Assieme ad altre tecnologie del W3C come il Semantic Web, il Web of Things astrae differenti protocolli e middleware grazie ad un documento descritivo interpretabili dalle macchine chiamato Thing Description. Questo documento vuole esplorare come questo nuovo paradigma influenzi il mondo del monitoraggio strutturale e ambientale. In particolare vuole verificare se l'utilizzo di tecnologie puramente basate su WoT e Semantic Web possa migliorare la portabilità di un applicazione di monitoraggio. In concreto propone un architetuttura software poi implementata in due casi d'uso reali presi dal mondo dello smart farming e monitoraggio di strutture industriali. Infine, la tesi, propone un organizzazione a layer del modello dei dati e una valutazione dei risultati ottenuti.

This page intentionally left blank.

Contents

1 Introduction **1**
1.1 Problem statement . . . 5
1.2 Contributions . . . 8
1.3 Thesis outline . . . 9

2 Background **11**
2.1 Structural Health Monitoring . . . 11
2.1.1 Global damage detection . . . 16
2.1.2 SHM sensors . . . 17
2.2 Internet of Things . . . 18
2.2.1 Internet of Things protocols . . . 22
2.3 Web of Things . . . 27
2.3.1 Thing Description . . . 34
2.3.2 Protocol bindings and Templates . . . 38
2.3.3 Servient . . . 41
2.3.4 Scripting API . . . 42

3 Open WoT Monitoring platform **45**
3.1 Related works . . . 46
3.1.1 Service migration . . . 48
3.2 Requirements . . . 50
3.3 Architecture outline . . . 53
3.3.1 Migration . . . 62
3.3.2 Discovery . . . 74
3.4 Implementation . . . 91

3.4.1 Structural Health Monitoring platform 93
3.4.2 Smart Agriculture 101
3.4.3 Tools 107

4 Evaluation and Discussion **111**
4.1 Software Architecture Analysis 111
4.1.1 Interoperability 112
4.1.2 Extensibility 114
4.1.3 Openness 114
4.1.4 Final comments 115
4.2 Migration 116
4.2.1 Policy Analysis 117
4.2.2 Use-case Analysis 124
4.3 WoT open points 127

5 Future work and conclusions **135**
5.1 Future work 135
5.2 Conclusion 137

6 Acknowledgements **139**

Appendices **143**

A Code listings **145**

Chapter 1

Introduction

Since the beginning of time, humanity has sought some level of mastery over the physical world. The first step of control is the understanding of natural and artificial phenomenons. Understanding, as the scientific method taught, means to study and observe extracting meaningful models and parameters. Therefore, the act of watching something over a period, also known as monitoring, became a crucial step in expanding our knowledge of the universe. We employed this technique to analyze the stars and unveil the secrets of the smallest particles. Even when we do not have a proper comprehension of something, monitoring can help to foresee future behaviors. For example, we monitor the status of the snow to estimate avalanches probability, or we watch climatic conditions to predict the wheater in the next days. The more the monitor process is accurate, the more the predictions are close the reality. Precise forecasts can have a real impact on our day to day life. In fact, they could prevent life losses (e.g., inform about the arrival of a hurricane) or economic advantages (e.g., reducing the maintenance cost of an airplane). Among the countless objects and parameters that could be monitored this work will focus on two particular domains: Civil structures and the Environment.

Every civil structure is subject to a slow decay caused by time, usage stress, and environmental conditions. Often a degradation of critical constructions like bridges or power plants can cause in-extremes interventions with service discontinuances and high financial losses by private and pub-

lic institutions. It is clear that this procedure, also known as corrective maintenance, is not suitable for this kind of problem. Therefore, usually, engineers define a maintenance plan to hamper critical damages to the objects and to contain costs of service failures. This practice is called preventive maintenance, and it has the benefit of reducing in-extreme intervention and severe damages. Nevertheless, preventive maintenance is usually based on previous experience and cannot be employed on novel structure types or materials. Furthermore, it is a challenge to plan and design due to the high number of hyperparameters that came to play. Even when it is perfectly crafted, there is still some probability that an unexpected event occurs which causes critical damage and the consequent unplanned repairment operation. Finally, it is not cost-effective, because the regular inspection campaigns usually involve human intervention and service termination. For example, airplanes are decommissioned every 400-600 flight [1] hours to go under an inspection intervention that disassembles the entire vehicle and assess the status of each component. On the other hand, predictive maintenance starts to be employed in the field to solve the high financial requirements of the previous techniques. Predictive Maintainance leverage on monitoring technologies to assess construction status. In literature, this process is called structural health monitoring. More in detail, this procedure consists of the usages of a set of sensors installed on the construction. Those sensors acquire what could be called the virtual signs of the structures like its vibration frequencies, tilt, displacements, temperature, and cracks length. As doctors use blood pressure, temperature, blood oxygen levels, etc. engineers diagnose structural health from those sensor data.

When they found the structure in bad conditions a maintenance intervention could be scheduled to stabilize the structure and stop further decay. The difference with preventive maintenance is that those repairment calls are on-demand and not based on a regular schedule. Therefore, a maintenance operation has a much higher probability to find a failure in the structure.

Like civil structures, the environment is a well-known subject of monitoring procedures. Environmental monitoring is a large domain that contains a wide number of practices and methodologies. Specifically, it can be defined as "the observation and study of the environment". As the Oxford

dictionary defines, the environment is "the natural world in which people, animals, and plants live" or generally speaking: "the conditions in which a person, animal or plant lives or operates or in which an activity takes place". Therefore, examples of environmental monitoring are:

- Air or water quality assessment
- Soil components extraction
- Plant health observation
- Climate studies
- Sea level or glacier monitoring
- Weather forecasts

Those operations come to play a critical role in different human activities. For example, in the agricultural domain where weather forecasts, plant growth assessment, and soil studies are crucial to optimize the yield and conserve resources. Furthermore, precise pollution monitoring can also improve health conditions. [2] reports that continuous air quality monitoring together with other commercially available technologies can reduce the emissions of CO2 levels in industrial plants by 60%. This would cause direct stress relief in the inhabitants of industrialized cities or regions [3].

As previously mentioned, modern monitoring methodologies involve the deployment of a set of sensors. In some applications, this batch of devices could be large. In [4] forty sensors were deployed to monitor a long bridge and in [5] a network of 150 nodes was simulated to verify an underwater pipeline monitoring. Managing this huge set of the sensor comes with his own challenges. Usually, these problems go under a specif computer science domain: the Internet of Things (IoT). The IoT extends the internet to the physical world dimension. In particular, it envisages the connection of every digital device to the internet and the progressive digitalization of physical objects. Thanks to the recent advancements in electronic manufacturing and computer science, its estimated market value is about 700 billion dollars [6]. Currently, even if we reached the vision of the ability to connect

everything to the internet, there still open issues that need to tackle to fulfill the dream of a digitalized world. Most of the potential of the Internet of Things resides in the level of interoperability between IoT applications and services [7]. Unfortunately, the IoT landscape is characterized by a very heterogeneous technology solution spectrum. Nowadays, there are several different protocols, stacks, and cloud ecosystems claimed to be THE preferred solution for IoT [8]. Although cloud ecosystems mitigate some of the interoperability issues, through web technologies (i.e. REST APIs, JSON, Web Sockets, etc.); vendor lock-in and silos architectures impact in a negative fashion the overall value of the IoT application. Furthermore, such solutions imply a sensor-to-cloud paradigm where sensors are connected and exploited through cloud connectivity. However, it is not always feasible to transport data from sensors to the cloud, due to security and energy consumption reasons. Even an edge computing solution does not solve some criticalities such as protocol interoperability, deployment, and maintenance costs. One of the emerging solutions capable of addressing the above issues is the Web of Things (WoT) [9]. In particular, in this thesis, we refer to the standard WoT definition proposed by World Wide Web Consortium (W3C) in 2020 [10]. As the Web standardized the Internet, the WoT paradigm aims to create a shared interaction model within the IoT world. Therefore, it addresses interoperability issues at the Application level creating a standard interface and a description for IoT devices. This description, known as the Thing Descriptor (TD), employs semantic tags to state the device capabilities or affordances. Using a common analogy, the Thing Descriptor is like a machine-understandable user-manual where functionalities and operations are described. The new solution proposed by W3C promotes a more collaborative shift inside the IoT as opposed to the sensor-to-cloud paradigm. In fact, even if it supports natively a sensor-cloud interaction, it foresees a more device-centric approach; where sensors, actuators, data, and applications are distributed in the full-stack spectrum.

To the best of the author's knowledge, current industrial monitoring solutions either use obsolete human-based techniques or when using IoT technologies does not leverage on WoT paradigm flexibility. Furthermore, especially in the monitoring of the environment and public civil structures, those solutions do no take into mind the possibility to provide open tools

that can be used by the population to gain insights about the monitoring process. For example, qualitative information about bridge health status could be conveyed in smart cities portals, so that tragedies like the Genova bridge collapse could be avoided thanks to the public awareness of the problem. In conclusion, for the above reasons, this thesis will explore the design of a multi-purpose open monitoring platform built with the Web of Things paradigm.

1.1 Problem statement

Building civil structures requires a relevant economic and engineering effort, but they are still subject to deterioration due to weather and usage. For example, a bridge can face environmental corrosion, persistent traffic, wind loading, and material aging, as well as unexpected seismic events, which may result in incremental structural deficiencies. At worst, this could lead to collapses causing tremendous additional costs and threatening human lives. Therefore, monitoring their structural health status is crucial to maintain their services and safety as long as possible.

At the same time, harsh environmental conditions can cause financial losses and health problems. Elements like extreme weather, strong air pollution, wildfires, earthquakes, invasive species uncontrollable growth, species extinctions, etc. have real impacts on the worldwide economy and human toll [11] [12] [13] [14]. Fortunately, novel monitor technologies, when employed together with prediction models, can stem the possible damages. For example, [12] reviews different techniques for hurricane forecast that could save up to 15% ($10.8 million of 1971 US$) in total and $2 million per hurricane warning per sector. Another work estimate that modern hurricane prediction could drop mortality by about 90% [15].

Other businesses that benefit from weather observation are those involved in the agri-food sector. In recent years, agriculture companies had shifted their gathering and management processes to use IoT technologies. The goal is to optimize human and economical resources with respect to yield quality. Weather monitoring associated with plant growth, plant health, and soil monitoring can provide vital information about the optimization

techniques with a relatively small budget. Although the importance of continuous structural health and environmental monitoring, those systems are rarely employed in our daily life. Just a small fraction of civil structures are really monitored and the one monitored are key bridges or dumps. Full-scale monitoring is needed to prevent deaths and assure that the high investments of building big civil structures are not wasted. For example, the reconstruction of the collapsed Genoa bridge in 2020 cost about 200 million euros 7% of the total budget reserved for the construction of roads and railways in the biennium of 2018-2020 of Italy [16]. Without mentioning the yearly cost of the absence of that key structure in the Genoa region which some estimating around 700 million euros [17]. At the same time, current state-of-the-art solutions represent a considerable investment related to the overall construction cost. The reason for their high value resides in the intrinsic challenges present when building a monitoring platform. Usually, monitoring demand the deployment of complex large scale systems like satellites, ground sensor networks, and drones. The clear physical differences between the objects subject to the observation complicate the design of the process and the platform. For example, monitoring a massive highway bridge is different from assessing the growth stage of a pear in a smart farm. Even similar structures require different types of sensors. Taking again the bridge example, a concrete viaduct may require strain gauges to observe crack evolution, while a steal railway bridge may employ only accelerometers or piezoelectric sensors. Having this heterogeneity in the type of sensors, increase the need to manage a multi-vendor set of devices. Unfortunately, in the era of the IoT, these circumstances increase the probability of encounter interoperability issues. Since the high number of different IoT protocols, there is a high chance to have different communication protocols involved in just one particular monitoring setup. Moreover, even when sensors convey the information using the same protocol, they might use different data formats. In practice, it is similar to when two people communicate in foreign languages. They use the same protocol (i.e. voice), but they cannot mutually understand (i.e., incompatible data models). Furthermore, the subtle differences in the software platform requirements might lead to different architectures. For example, one particular application might result in a centralized stack (e.g., typically a sensor-to-cloud architecture),

one other to a distributed design (e.g., a fog computing scenario). Those technical challenges together with business interests lead to ad hoc solutions, also known as silos applications, that can not be really reused in other fields. Even when a software solution is claimed to be general-purpose, they are discarded because of their incompatibility with some aspects required in a particular monitoring process. Silos architectures not only limit the portability of a particular monitoring software product but also close the opportunity to create innovative unexpected services from monitored data. If in some applications there's a clear end-user, like when design an industrial plan monitoring, others might have a more fuzzy definition of the interested stakeholders. When the object monitored is a public interest (e.g, bridges, highways, public buildings, local weather, air quality, etc.), municipalities might want direct access to monitoring data. At the same time, developers might come up with novel ideas to exploit those data, providing services to the community. Moreover, engineers might want to track the status of construction and provide timely intervention plans. Finally, a regular citizen can also play a role both as a consumer or producer of monitored data. Crowdsourcing the monitoring sensing process is a new compelling field with remarkable results that could end up plunge the economical effort of monitoring metropolises and medium-sized cities.

In conclusion, the development of a software monitoring platform has its unique challenges. In particular, we have defined 5 different sources of heterogeneity that can increase the complexity of such a solution:

- Sensors and methods
- Observed properties
- Computing architecture
- Data formats
- Users

For these reasons, despite the economical advantages of having a general one fit all solution, a clear unified open monitoring platform has yet to emerge.

1.2 Contributions

The lack of compelling general-purpose monitoring platforms limits the adoption of predictive maintenance techniques and the creation of innovative cross-domain services. Therefore, this document explores the requirements of a possible open monitoring platform that servers as a general framework to assess the physical properties of an observed object. Additionally, the thesis presents a software architecture derived from those requirements. Then, the architecture is implemented in two specific use-cases. The first defines a typical structural health monitoring application applied to both bridges and pipelines. The latter, it coming from the smart agriculture domain, where the monitoring is the first step in the control of the long production process that transforms the energy sources in vegetables and fruit. In these contexts, the platform data model was extended to address domain-specific requirements. Specifically, in the agriculture domain, state of the art ontologies were found insufficient. Therefore, an IoT inspired smart agriculture ontology was defined and published, as later described in Chaper 3. On the other hand, in the SHM domain, integration strategies and ontology alignments were employed to leverage on the state of the art data models. Moreover, as the W3C WoT is still undergoing the standardization process, this work identifies weak spots and provide possible solutions to be evaluated in the future. Finally, the core contributions of this work can be summirzed as follows:

- Define an open architecture for structure health and environment monitoring.Particularly,this study concentrates around the open requirements, but without forgetting security and safety issues.

- Discuss weak spots in the WoT standard related to the use case and propose possible solutions.

- Design an IoT inspired ontology for Smart Agriculture. The ontology is divided into four subspaces that represent different aspects of smart agriculture data space. Where possible, those subspaces were aligned with existing vocabularies to maintain a high interoperability level.

- Platform integration with industry data models in the SHM domain. In particular, the work focused on the integration of two emerging standards: Building Topology Ontology (BoT) and Industry Foundation Classes ontology.

Besides scientific contributions, this document also contains the description of two tools, namely WoT Application Manager (WAM) and WoT Farm, that were developed and employed during the evaluation of the proposed platform.

1.3 Thesis outline

The remainder of the document is composed as follows. Chapter 2 introduce the background and related works, focusing on the web of things standard and IoT monitoring technologies. Chapter 3 describes the main contributions in detail, starting from the requirements of the monitoring platform. Then the software architecture is described with insights on the implementation challenges. Finally, the chapter presents a possible data model alignment within the agriculture and building monitoring use cases. Chapter 4 evaluates relevant aspects of the platform and discusses gaps between the WoT standard and the platform requirements. At last, Chapters 5 and 6 draw the conclusion, discuss future works, and acknowledge external contributions for this work.

This page intentionally left blank.

Chapter 2

Background

This chapter will introduce the history of Structural Health and Environmental monitoring, together with the fundamental techniques and concepts. Moreover, the second section will describe the evolution of distributed open computation technologies till the born of the Web of Things. Additionally, section 2.2 provides the main definitions of WoT concepts that will be used later on in the description of the proposed platform. In conclusion, this chapter serves as a foundation of the research ideas discussed in Chapter 3.

2.1 Structural Health Monitoring

With the term Structural Health Monitoring (SHM), we refer to the strategies adopted when assessing and measuring damages to any human construction. It could be a civil structure, an aerospatial vehicle, or a manufactured object. Specifically, when we speak about structural damage, we indicate any unwanted internal change suffered by a system. Consider that every single damage has its origin from micro imperfections in a material. These imperfections are always present even at the beginning of the life of a system. Those defects could later grow and became substantial damages because of stress, overloading, corrosion, and passing of time. Therefore, it is crucial to observe the complex system during a period and study changes to identify criticalities that can later evolve into structural deficiencies. Besides, SHM objectives can also cover other aspects of the construction process like

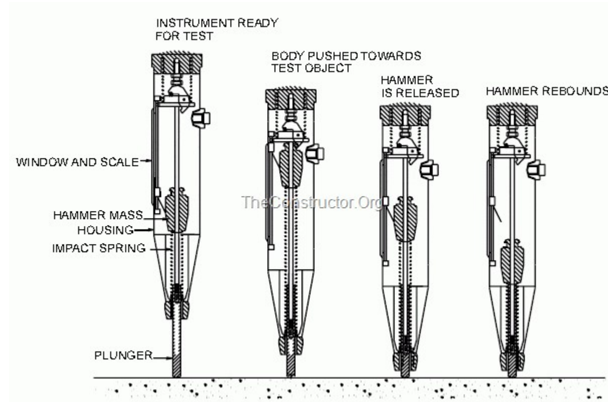


Figure 2.1: Different phases of the rebound hammer test

quality control, excessive loading warnings, and condition assessment [18]. This shifts the paradigm from the routine or critical event (e.g., earthquake) based inspections to continuous monitoring of the object. Based on the real-time data collected, engineers can program repairment of strengthening works to expand the functional life of the monitored construction. Consequently, health monitoring of civil structures has been attracting much attention from the research community and practicing engineers since the 40s. Initially, civil engineers had to answers practical questions like when the concrete was sufficiently ready to be removed from formworks. Specifically, the principal need was to determine the homogeneity and the compressive strength of fresh concrete. Therefore, different experimental methods were employed, usually based on a visual inspection and other non-descrictive techniques (NDE). For example, the rebound hammer test (Figure 2.1), where a mass hit with a specific energy a spring in contact with the concrete surface. The level of the rebound provoked by the spring measures the strength and the stiffness of the concrete. More in detail, a low stiffness concrete absorbs more energy and it results in a small rebound of the mass. When the urban expansion was at its decline, structures started to age. Consequently, engineers were more interested in the real mechanical properties of old constructions as well as the characterization of hidden defects. The previously mentioned methods were not sufficient to inspect

precisely the internal physical properties of a material. Therefore, new techniques were born during the 70s to satisfy these novel needs:

- Acoustic emission
- Electro-magnetic field methods
- Eddy current methods
- X-ray
- Thermal field methods

Even if the new methods were more advanced and precise, they required physical access to the inspected object. Furthermore, they could not assess the general health status of complex constructions or composite materials like concrete. On the contrary, they focused on specific damages types localized in a finite spot of the structure. For this reason, modern SHM techniques leverage on global damage detection methods (See Section 2.1.1). Nowadays, preserving civil infrastructures nationwide is dependent on the successful implementation of structural health monitoring concepts. According to [19], the engineering structural health process consists of four distinct steps :

- Sensor allocation and measurements
- Structural identification
- Damage or degradation detection
- Decision making

Every one of these steps is an entire research field inside SHM. First is sensor development and deployment. As Section 2.1.2 will detail, SHM methodologies required the design of new special sensors that took in mind the size of the objects monitored and their physical properties in interest. Then structural identification and damage detection cover the mathematical challenges to correctly model and predict the behavior of composite constructions. In this context [20] and [21] identify four accuracy levels for damage detection:

- Identify that damage has occurred
- Identify that damage has occurred and determine the location of the damage
- Identify that damage has occurred, locate the damage, and estimate its severity
- Identify that damage has occurred, locate the damage, estimate its severity, and determine the remaining useful life of the structure

Of course, researchers strive to achieve the highest level of accuracy. However, a generic algorithm is yet to find. Hopefully, long term monitoring of a multitude of buildings and civil works could generate a consistent dataset. This knowledge could be then fed to one or more machine learning algorithms that could obtain the highest accuracy in most of the observed objects. Early experiments are already carried out with promising results [22].

Finally, the decision-making level that maps sensed knowledge to real-world actions. For example, considering the location of damage in a bridge, should we intervene for a repair? Should we close half of the bridge immediately to reduce the loading stress? etc.

Figure 2.2 shows the usual architecture of an SHM system. Sensors measure physical properties and usage levels which later processing units elaborate to extract local deficiencies. Commonly, these sensors are connected to cabled or wireless networks. One or more networks can cover one single structure. Since the monitored object might have considerable sizes, it is a common practice to monitor different components with different sensor networks. Once data is processed it is safely stored inside a historical database. The prognosis uses this temporal data together with the real-time information to evaluate the current structural health status of the observed object. This status is the combination of the different sources described above. It summarizes the local deficiencies sensed by the sensors, the global behavior obtained by usage information, and the historical parameters.

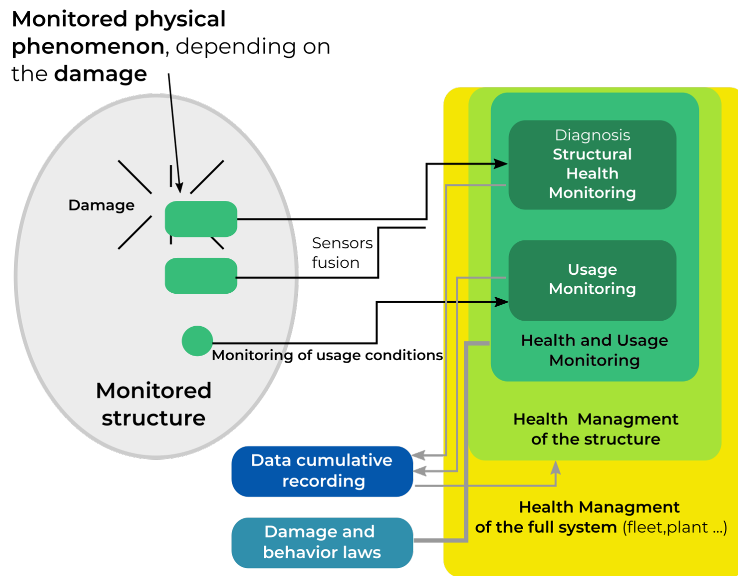


Figure 2.2: A schematic view of a typical Structural health monitoring system.

2.1.1 Global damage detection

Global damage identification methods (GDMs) are novel techniques to assess the health of a complex structure or a building (i.e., a construction composed of different sub-parts). As the name entitles, they differentiate from the local damage identification methods (LDMs) because of their scope. LDMs use non-destructive methods to identify damage in a specif structural component, whereas GMDs reason about the behavior of the overall structural complex system. Global damage identification algorithms can be classified into two categories: model-based and response-based. The model-based methods assume that a physical numerical model of the structure is defined. Engineers then exploit the model to simulate the nominal behavior. Consequently, a substantial deviation from this ground truth suggests possible damage. Hence, the accuracy of this process yields down to the accuracy of the model itself. However, accurate physical models for most known structures are unavailable yet. This circumstance confines the usage of model-based methods to reticular steel structures that have a well-known model. On the other hand, response-based methods are more versatile but much harder to configure and initialize. A typical procedure for response-based damage detection involves the extraction of behavioral parameters. Physically speaking, dynamic and static responses to external solicitations define the behavior of a structure. Consequently, these responses are monitored and stored in a historical database. The dynamic responses are commonly measured using the so-called vibrational analysis. The basic premise of vibration-based SHM is that changes in structural characteristics, such as mass, stiffness, and damping, will affect the global vibrational response of the structure. on the other side, the static responses are obtained from the displacement of key points in a structure. If damage occurs changes in the local stiffness of a structure determine a change in one of the two responses. Therefore, response-based algorithms exploit historical data to infer possible structural deficits. This method has the advantage to be applicable to a wide range of different objects since it does not require a modeling phase. However, understanding of the relationships between the damage and the corresponding changes in the dynamic properties is an open research problem.

2.1.2 SHM sensors

The SHM field has its unique challenges, hence during the years, custom sensor types were developed to measure the desired physical properties. This section will introduce the most common one giving an overview of the heterogeneity of this field.

- Strain Gauges.This type of sensor measures the physical displacement of two points. They were the first sensor employed to measure static and dynamic structure behaviors. Their basic principle is that the resistance of a conductor change when it is subject to strain. In the market, this sensor type is specialized for different applications. There are strain gauges that measure structural stress (stress gauges), material bending (flexagauges), etc.

- Gyroscopes. Gyroscopes measure the angular displacement of an object, usually on multiple axis. They can also provide angular velocity but this information is usually not relevant in an SHM process.

- Accelerometers. An accelerometer measures the acceleration alongside three axes. This sensor is closely related to the gyroscope so that they are commonly packed in a single sensor. In SHM they are used in the structural characterization process as a measurement of its frequency response to solicitations.

- Piezoelectric Transducers. This sensor type is the most popular and widely-used means of measuring the internal structural properties of a material. Piezoelectric sensors function like a stethoscope earning the internal micro-vibration of a material. They are far more sensible than strain gauges and accelerometers. In fact, they can measure up to the Mhz range against the Hz range of an accelerometer. Thanks also to their physical properties they are the only sensor that does not require any power for functioning. The energy comes from the deformation of the material itself and it is later transformed into a digital signal. One caveat is that they need to firmly attach close to the location of possible damage which is usually unknown. Furthermore, if not properly attached or if corrosion and weather agent causes a partial

detached, they lose accuracy and sometimes they became defat to new inputs. In SHM they are used both as listening devices or as active entities that send an acoustic signal and wait for the reflected response. They are also used as a localization device for cracks. Deploying at least three piezoelectric sensors it is possible to obtain the location of a punctual energy emission(i.e. a crack) thanks to trigonometry techniques.

- Laser Doppler Vibrometer. Laser Doppler Vibrometers measures the instantaneous velocity of the surface of the structure. They are optical devices that use laser interferometry to detected micro-vibrations in the surface of a material. The advantages with respect to piezoelectric sensors are that they do not require physical contact with the surface. The only requirement is the line of sight. This makes LDV a game-changing sensor when analyzing the vibrational responses of an inaccessible object. For example, when the surface of interest is particularly curved, or when it has high temperatures that can melt piezoelectric sensors. However, they are more expensive and sensible to noise.
- Fiber Optic Sensors(FOS). These sensors are actually a whole family of different measuring devices. They are characterized by the usage of fiber optic materials as the mean for the measure. There are fiber optic accelerometers, gyroscopes, vibrometer, stress gauges, flexagauges, etc. In SHM, FOS sensors are appreciated for their resilience. They have a low susceptibility to electromagnetic events (e.g. storms) and a strong corrosion resistance. Moreover, they have a small size factor, a wide range of operational temperatures, and a long lifetime. On the other hand, they found to be sensible to micro-variations of the light source and ambient conditions (dust, moisture, smoke, etc.).

2.2 Internet of Things

The Internet of Things (IoT) is a novel term to refer to the interaction between the Internet and the physical world. Although it is now a commonly

Figure 2.3: A sample of different shm sensors. From the top right corner we found: accelerometers, piezoelectric, strain gauge, laser vibrometer, and fiber optic sensor

used label, its vision, digital systems that blend into everyday life, has roots in modern history. In the computer of the 21th century [23], the year 1991, Mark Weise uses the term *Ubiquitous computing* when explaining how information technology will evolve during this century. He imagined a profound bond between humans and computing devices:

> Machines that fit the human environment, instead of forcing humans to enter theirs, will make using a computer as refreshing as taking a walk in the woods.

As pioneering as this idea could have been, it was still a vague prediction of what the world of computers will look like in the coming decades. It was only in the early 2000s that Kevin Asthon concretized the idea of Ubiquitous computing in something more specific. He deliberately envisioned that the means to achieve Weise's dream was throughout the Internet. Electronic devices were becoming smaller and smaller every year; therefore, it was possible to install computing power in something different than personal computers and mainframes. Nonetheless, it was not sufficient to provide real

value to everyday users. Only giving the ability to communicate with other peers opened the possibility to a whole set of new applications. Ashton, in how to fly a horse [24] recalls one concrete application for all that gave him the idea of the Internet of Things. When working for Procter & Gamble, he was asked to figure out why half of the company resellers were always out of stock of a shade of one of their lipsticks. After researching, Ashton noticed that the problem was in the missing information. In practice, resellers were not ready to communicate the real count of the lipsticks on the shelf. To solve this problem, Ashton placed a radio microchip in every lipstick and an antenna receiver on the shelf, but he did not stop there. He finally connected the shelf antenna device to the Internet, opening the possibility to remotely track the real lipstick count in every reseller's store. It was this simple act that inspired him to coin the term Internet of Things.

As time passes, ideas evolve and the IoT was not immune to the changes of time. After the first timid applications in the retail industry, the Internet of Things became famous for its applicability in a wide range of domains. Examples of IoT applications can be found in agriculture and food manufacturing, healthcare, home automation, in industrial plants Such flexibility led to a proliferation of a wide range of technology solutions. At first, IoT was about connectivity; existing protocols were not able to handle long-range wireless connections and high data rates. Consequently, different protocols emerged to cope with the increased demand for reliable wireless solutions. Not only, but also the Internet Protocol (IP[1]) address was extended to cover the expanded address space (See IPv6[2]). Currently, the protocol spectrum of the IoT has plenty of solutions for every IoT domain. Refer to Section 2.2.1 for further details. Nonetheless, mere connectivity, as vital as it is, does not implies communication and coordination between agents. Applications at IoT scale requires means to specify the interactions between those distributed nodes and orchestrate devices to follow the desired business logic. Therefore, the ICT market answered with a proliferation of different middlewares or full-stack frameworks. The most favorite architecture design was to leverage the recent paradigm of cloud

[1] https://en.wikipedia.org/wiki/Internet_Protocol
[2] https://en.wikipedia.org/wiki/IPv6

computing. The bound between IoT and Cloud became so strong that they are often mistaken for one the other. However, the so-called sensor-to-cloud architecture has its shortcomings. First of all, it is not always feasible to transfer data from sensors to remote datacenters due to bandwidth or security limitations. Furthermore, employing the cloud as the control center limits the response time of the system. Even with a high transfer network local computation has always a lower latency for simple computational inexpensive actions. Finally, it does not take into account the lower level optimizations of IoT protocols and devices. For example, sensors might have a long sleep-active time cycle to conserve energy; frequent readings from cloud computing applications would disrupt this optimization routine leading to to poor battery life performance. In recent years, a new shift in the IoT landscape strives to solve the above issues: the Edge Computing. Edge computing moves critical services and applications from remote datacenters close to the devices. The extremization of this procedure is Fog/Mist computing, which takes the services right in network switches, gateways, or even sensors. Even if these new software designs solve sensor-to-cloud issues, they open new ones. In particular, due to the variety of protocols and communication standards, the more we descend the stack (i.e., from data centers to sensors), the more interoperability between protocols became challenging. While between cloud services Web technologies (e.g. HTTP, WebSocket, SSE, etc.) are the defacto standard for communication, at the edge (and even more at the fog/mist level), the protocol spectrum is quite erratic. These circumstances create ad-hoc solutions aimed at solving one particular application at a time: silos applications. Engineers select one or two protocols well-suited for the application domain, connect the devices to an edge server (if needed), and implement the application business logic at the cloud level. Silos applications, as the name suggested, enclose the value provided by the IoT solution and restrict the possible "unpredictable usages" of the information extracted. In conclusion, IoT is an ever-evolving field with wide adoption in every economic and scientific domain. Nonetheless, the heterogeneity of the proposed market solutions limits its true potential restraining the ability to create multi-domain applications and reusing existing resources.

2.2.1 Internet of Things protocols

Given the high adoption of the IoT paradigm in different application domains, specific communication protocols arose to satisfy particular application requirements. This section will describe a list of the most notable protocols, their qualities, and the problems they are trying to solve. Moreover, the following will serve as a glossary for protocols that are mentioned in the next chapters. The reader is invited to refer to this section if needed.

- Message Queuing Telemetry Transport (MQTT). MQTT[3] is one of the most successful IoT protocols on the market. It is a publish-subscribe network protocol that runs over TCP/IP or any other protocol that provides ordered, lossless, bi-directional connections. It is designed to be lightweight and transmits a large quantity of real-time data with low energy consumption. One of its first applications was monitoring a remote oil pipeline, but it was later also largely applied in mobile applications. The protocol defines two software agent roles: clients and brokers. A broker is the core component of the protocol, and its prime duty is to deliver messages to clients. On the other hand, clients are broker users that can publish messages or subscribe to a specific topic. Its quasi-stateless architecture allows MQTT to scale at an impressive number of connected nodes. For example, RabbitMQ (one opensource broker implementation) is claimed to handle one hundred thousand devices guaranteeing a message latency under 10 ms [25]. This feature, among others, renders MQTT the preferred solution for monitoring applications and real-time automation (e.g., it is employed in smart homes and agriculture domains).

- Constrained Application Protocol (CoAP).CoAP[4] is a request/response web protocol specially designed for constrained devices. It is closely inspired by the most famous web protocol: HTTP. As RFC 7252 describes this retains key concepts from the web like URIs and content negotiations but provides unique features suited for lossy networks and small microcontrollers. For instance, it runs on UDP

[3] `https://mqtt.org/mqtt-specification/`
[4] `https://tools.ietf.org/html/rfc7252`

instead of the TCP and it includes support for discovery and multicast. CoAP reuses the same HTTP actors and nomenclature, but it prescribes a subset of HTTP operations. In particular, CoAP does support only GET, POST, PUT, and DELETE request types with the same HTTP semantics. It is claimed to provide interoperable machine to machine communication and thanks to its versatility it can be used in any IoT domain, preferably at mist/fog level.

- HyperText Transfer Protocol (HTTP).HTTP[5] is an application protocol born with the dawn of the Web. Even if it was originally designed for exchanging hypermedia contents, nowadays it evolved to a general-purpose interaction model (i.e., RESTFull applications). Its adoption in the IoT landscape is mostly related to cloud environments. It is rather used as an application protocol between microservices that store, process, and present IoT data streams. However, there are some examples of its adoption in smart home domains where the device capabilities are not too restrictive. Although in such contexts CoAP is usually preferred.

- WebSocket[6]. WebSocket is a computer protocol over TCP providing full-duplex communication. Its main feature is to be compatible with the HTTP protocol, using the Upgrade header mechanism. Due to this ability, the most known browsers adopted this protocol as the favorite solution for bidirectional communication between servers and JavaScript scripts. In IoT, it is employed as a protocol to convey device events or real-time data. Since it does not define any particular interaction model, sometimes it is used as the transport layer for other IoT application protocols like MQTT. Thanks to it interoperability with browsers is, therefore, employed for dynamic web dashboards in the Smart Home and mobile domain.

- Long Range Wide Area Network (LoRaWAN)[7].LoRaWAN is MAC based (no-IP) networking protocol implemented over LoRA physical

[5] `https://tools.ietf.org/html/rfc2616`
[6] `https://tools.ietf.org/html/rfc6455`
[7] `https://lora-alliance.org/resource-hub/lorawanr-specification-v11`

layer, a long-range low power physical layer protocol developed by Semtech. LoRaWAN acts mainly as a network layer protocol for managing communication between gateways and end-node devices. Typical implementations of LoRaWAN protocol adopt a cloud computing architecture and more interoperable application protocols like MQTT and HTTP. For example, ChirpStack translates LoRaWAN messages to MQTT messages and vice-versa. This enables the development of Web applications that use remote devices transparently thanks to the MQTT protocol. Its long-range and low-power capabilities are appreciated in the environmental monitoring and smart agriculture domains. In such application contexts, sensors are deployed in remote locations and measure slow physical phenomenons; all requirements that LoRaWAN satisfies. On the other hand, its low bandwidth capabilities limit its adoption for telemetry application that requires high data rates (i.e. industry 4.0, automotive, etc.)

- DASH7[8].DASH7 Alliance Protocol (D7A) is an open-source Wireless Sensor and Actuator Network protocol DASH7, which provides multi-year battery life. Similar to LoRaWAN, it is designed for long-range communication, although it reaches 2 Km at maximum. It also leverage on the same cloud-based architecture, where clients interact with the devices thanks to a cloud translation of the DASH7 protocol to MQTT. On the other hand, it has lower latency even with moving objects and real bidirectional communication between peers nodes. Thanks to these features it can be employed in smart cities, industrial settings, and logistics.

- 6LoWPAN[9].6LoWPAN is an extension of IP to devices with lower computational capabilities. It is based on IEEE 802.15.4, which is a standard wireless communication protocol for low rate personal area networks. The main benefit of this technology is that it allows a lighter implementation of the IP stack. In practice, this translates into a lower entry barrier for the participation of the Internet.

[8] https://dash7-alliance.org/download-specification/
[9] https://tools.ietf.org/html/rfc8138

- ZigBee[10]. Zigbee is also an IEEE 802.15.4-based specification for a suite of high-level networking protocols used to build lightweight, low-power digital radio personal area networks, such as home automation, data collection for medical devices, and other low-power low-bandwidth specifications, designed for small-scale projects that involve wireless connectivity.Zigbee is therefore a wireless ad hoc network with low power, low data rate and close proximity (i.e. personal area). The Zigbee specification-defined technology is intended to be easier and less expensive than other wireless personal area networks (WPANs), such as Bluetooth or more general wireless networking such as Wi-Fi. Among the possible applications we cite wireless light switches, home energy monitors, and traffic management systems.

- Bluetooth[11]. Bluetooth is a standard for wireless technology used to transmit data over short distances between fixed and mobile devices using UHF radio waves in the commercial, science and medical radio bands, from 2,402 GHz to 2,480 GHz, and to create private area networks (PANs).It is mainly adopted in the communication of small battery devices and appliances. One of its characteristics is the definition of application profiles that enables a standard communication protocol and configuration. Most known applications are smart lock control, headsets for audio reproduction and recording, transfer files or contacts, motion controllers in VR headsets, smart treadmills and exercise bikes.

- Lightweight M2M[12]. OMA Lightweight M2M (LwM2M) is an application protocol from the Open Mobile Alliance designed for Machine to Machine device management and service communication. It provides an IoT device management strategy and enables devices and systems from multiple vendors to co-exist in an IoT ecosystem. LwM2M was

[10] https://zigbeealliance.org/wp-content/uploads/2019/11/docs-05-3474-21-0csg-zigbee-specification.pdf

[11] https://www.bluetooth.com/specifications/bluetooth-core-specification/

[12] http://www.openmobilealliance.org/wp/Overviews/lightweightm2m_overview.html

originally built on CoAP, but later LwM2M versions also support additional transfer protocols. The system management capabilities of LwM2M include remote security certificate provisioning, firmware upgrades, management of networking (e.g. cellular and WiFi), diagnostics of remote devices, and troubleshooting.

- Modbus[13] over TCP. Modbus is a de facto standard for the communication of industrial PLC devices. It gained its popularity thanks to its simplicity and open specification. Moreover, it was originally developed for serial communication lines but nowadays it is extended to the Internet: Modbus over TCP. One downside of this protocol is the lacking of an encryption or security layer.

- CAN[14] bus over TCP. Controller Area Network is a protocol originally designed for the smart car industry. It is a message-based protocol supporting the common publish/subscribe pattern. Even if it was originally convinced as an automobile protocol with a low copper impact it was later adopted in other contexts (e.g., Structural Health Monitoring). CAN TCP is an extension of the protocol over the Internet.

- OPC-UA[15]. OPC Unified Architecture (OPC UA) is an industrial automation machine to a machine communication protocol. The protocol has two forms: binary and web based. The binary version uses TCP as the transport protocol and has its own defined URL schema (opc.tpc://server). It is usually the preferred solution in constraint devices because of its parsing speed. Another important factor is that the binary protocol has less degree of freedom in the data exchanged. As such, it has a higher interoperability level regards to its web companion. The web protocol is based on HTTP; as a consequence is more firewall-friendly and expressive. Unfortunately, the high complexity of the protocol specification leads to a fragmentation in

[13] `https://modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf`

[14] `https://www.kvaser.com/software/7330130980914/V1/can2spec.pdf`

[15] `https://opcfoundation.org/developer-tools/specifications-unified-architecture`

the Server implementation of the protocol which in turn diminished its concrete adoption.

- Netconf The Network Configuration Protocol (NETCONF)[16]. NETCONF is a network management protocol developed and standardized by the IETF. The NETCONF protocols specify mechanisms to install and configure devices. . It is bulti on top of high-level transportation and application protocols like TLS and HTTPS. Its operations leverage on a simple Remote Procedure Call (RPC) layer alongside a notification protocol. As a message data format, the NETCONF protocol uses an Extensible Markup Language (XML); configuration data is also encoded in XML. One of the core features of NETCONF is its security which relies on the TLS protocol. It is used in industrial IoT applications but it had a wide adoption in the implementation of smart gateways and routers.

2.3 Web of Things

While the Internet of Things was in infancy, Tim Berners Lee and Ora Lassita visioned an evolution of the World Wide Web to a new dimension [26]. They imagined that all the information enclosed in static HTML pages as written text could be disclosed to machines. Therefore, they coined a new paradigm: the Semantic Web, intended as a global Web of Data, which software agents can interpret and process directly. This possibility would open the creation of next-generation web applications able to seamlessly scrape data from the new immense source of information. For example, they open their paper with:

> The entertainment system was belting out the Beatles' "We Can Work It Out" when the phone rang. When Pete answered, his phone turned the sound down by sending a message to all the other local devices that had a volume control.

[16] https://tools.ietf.org/html/rfc6241

Interestingly enough the example is not really far from the description of the Ubiquitous computing given in Section 2.2 by Weise. However, in the early years, the Semantic Web was focused more on describing knowledge in a machine-understandable way. Was during these early stages that the Resource Description Framework[17] was standardized by the World Wide Web Consortium. RDF is a general metadata model and it is base on statements composed of a subject, predicate, and object (also known as triples). In practice, the subject denotes a particular Web resource or abstract entity, the predicate one of its properties, finally the object the value of this property. For example, consider that a user is identified with urn:people:user12345 (a Uniform Resource Identifier or, in short, URI[18]) the fact that he is named "Francesco" can be expressed with the following RDF triple:

urn:people:user12345 http://schema.org/givenName "Francesco"

Notice that RDF is completely transparent about the URI used as the predicate; in the example, we could have used urn:dev:name or urn:dev:isCalled RDF terms as well. However, writing an RDF statement with commonly understood terms is critical to assure a certain level of machine-to-machine interoperability. A machine has some difficulties to understand that urn:dev:isCalled has the same semantic as http://schema.org/givenName. It should be manually instructed to threaten two vocabulary terms in the same way. Therefore, for most common terms in a domain, the Semantic Web community is developing a set of vocabularies or ontologies. Ontologies describe those terms both in human and machine understandable forms so that developers can encode these special URIs in their programs. Even if it was originally conceived as a means to describe human knowledge, thanks to its intrinsic expressivity, RDF becomes also widely adopted to define services capabilities. Having services represented as machine-readable data has been found useful when it comes to discovery and self-documentation. For instance, in the European Union founded Smart Objects For Intelligent Applications (SOFIA) project, intelligent objects and virtual services were

[17] https://www.w3.org/RDF/
[18] https://tools.ietf.org/html/rfc3986

represented as RDF triples in a shared database called the Knowledge base. Here smart agents or Knowledge Processors (KP) could infer real-world facts and act intelligently. Moreover, they could discover functionalities during runtime thanks to the uniform and machine-understandable RDF description. Another interesting concept detailed in the SOFIA project was Smart Space. A Smart Space is a physical environment augmented with those RDF descriptions and a set of KPs or software services. Therefore, a Smart Space is where the semantic web meets the physical world proving new functionalities and commodities to the users.

The idea of Smart Spaces brings us back to the world of the Internet of Things. As outlined in Section 2.2 one of the major challenges in the IoT domain is the interoperability between applications and services. Could RDF descriptions of services mitigate those issues?

In 2011 Dominque Guinard, in his Ph.D. thesis, proposed the Web of Things: a web architecture for the Internet of Things [9]. His idea was that every device should be able to interact with its peers through an HTTP interface over IEEE 802 (Ethernet) or IEEE 802.11 (WiFi) network. More in detail, the architecture defines four layers:

- Accessibility. This first layer provides basic connectivity to the internet and assures that device services are available as Web APIs. This interface follows the RESTFull principles [REF] allowing interactions with universally supported web methods. When it is not possible to model a device using Web standards and protocols a Smart Gateway[Leveraging the Web to Build a Distributed Location-aware Infrastructure for the Real World] can act as a bridge between the smart object and web clients (e.g., when the device uses non-web protocols like ZigBee or Bluetooth). Thanks to this layer, web clients can access physical resources using Uniform Resource Identifiers and exchange data using HTTP content negotiation (i.e., clients can choose their preferred serialization format).

- Findability. On this level, smart objects are contextualized and described with metadata. Thanks to their formal description, typically in RDF, search engines can record the physical resources and web clients can infer their capabilities.

- Sharing. the previous two layers allow unrestricted access to physical devices and data from the web. If sometimes public information access is welcomed (i.e., an Air Pollution sensor for a smart city), in other circumstances this could violate user privacy and safety. Therefore the Sharing layer prescribes a set of rules and technology stacks (e.g., OAuth 2.0) to control the information sharing in WoT.

- Compositions. This final layer consists of a set of tools and frameworks to create services from the WoT open ecosystem. As the previous layers build an open distributed system the Compositions layer provides useful abstractions to create complex Mashup applications integrated with other web services.

If we focus on the first two layers we can understand that in Dominque's architecture the IoT interoperability issues are solved by two strategies. The first is to mandate only one protocol, HTTP; the second is to define a formal interaction pattern (RESTFull) described by a semantic description. The semantic description is critical to assess application-level interoperability. Consider one application that turns on the lights of a room if a presence sensor detects a person inside. Application-interoperability assures that the application can understand from device description which is the light and which is the sensor. Not only, but it also allows the smart agent to understand how to interact with them. Devices from different vendors might have different APIs and a formal description can instruct the application on how to use them. Much like, when a human reads a user manual to understand how to wash its favorite clothes in a washing machine. Consequently, yes, as WoT demonstrate, RDF descriptions can mitigate some open issues of the IoT. On the other hand, prescribing HTTP (or similar web protocols) as the only protocol allowed, limited the software market adoption of WoT architecture. The IoT ecosystem is full of corner cases where high-level protocols like HTTP does not satisfy stringent domain-specific requirements. Nevertheless, some companies believed in Dominque's vision and create various versions of its architecture. Above all, Mozilla IoT[19] is one of the most known. Mozilla IoT targets Smart Buildings scenarios

[19] `https://iot.mozilla.org/`

where there are no hard requirements for esoteric protocols or performance. On the contrary, users need well-designed dashboards to control their HVAC systems or to create automation patterns. Developers in this field value more easy-to-use and interoperability rather than performance and real-time capabilities; all features that a web platform can support. In summary, Mozilla IoT architecture provides for a similar structure to the original Dominque's idea. For the Accessibility and Findability layers, Mozilla provides a Smart Gateway implementation, middleware for devices capable to handle a web stack, and implementation of a semantic description for smart objects. For the Compositions layer, Mozzila Smart Gateway provides means to create simple mashup applications, whereas the middleware defines low-level APIs to access device resources from code. Sadly, although its success, Mozilla IoT was shut down for internal policies and left to the open-source community.

As other versions of WoT emerged, in 2016 the World Wide Web Consortium (W3C) step in with the quest to standardize the definition of the WoT ecosystem, finally bringing the web down to the physical world. The W3C is an international non-governmental organization founded by Tim Berners Lee with the goal to develop standards related to the Web. For example, his in charge of the definition of HTTP, HTML, and CSS technologies. For the Web of Things standardization, W3C assembled one working group which published four different major recommendation documents: Web of Things Architecture [10], Web of Things Thing Description [27], Web of Things Discovery [28], and Web of Things Profile [29]. Each one of this document specifies normative requirements for implementers to guarantee a certain level of interoperability between software solutions. The remainder of this section will summarize the W3C Web of Things focusing on the architecture specification from the architecture. Notice that later in this thesis, we refer to this W3C interpretation when using the term Web of Things or WoT. The core component of the WoT architecture is a Web Thing (sometimes shortened as Thing). A Thing is a virtual or physical entity described by metadata serialized as W3C Thing Description and it is logically composed of five aspects: behavior, capabilities, data schemas, security configurations, and protocols. Those concepts can be directly mapped with the 2011's architecture shown in Figure 2.4. The behavior defines the internal business

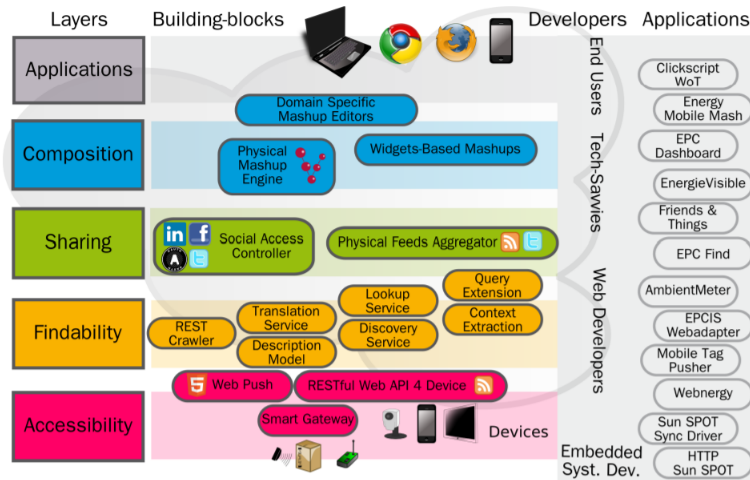


Figure 2.4: Original Web of Things architecture [9]

logic of Thing software agents; for example, the algorithm to correctly operate a robot arm in the desired position. Then capabilities are the set of abstract operations that a Thing can fulfill. In the W3C WoT, they are called Interaction Affordances (or simply Affordances) as formally defined by Donald Norman in the field of human-computer interaction:

> 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.

The affordances define an abstract model of how a client may interact with a Web Thing. However, they do not explain which protocols or data encoding should be used. Protocols and Data Schema logically components define exactly that. Data schemas are semantic descriptions of the data exchanged for a particular affordance. For instance, they define that the temperature affordance provides a floating-point number in Celsius in the range of -30.0 and 100.0. On the other hand, protocols called protocol bindings in the WoT specification are the list of supported network protocols by a Web Thing. Here we can notice the first difference with the original WoT architecture

Figure 2.5: Interaction schema between WoT agents.

proposed by Guinard. W3C does not impose HTTP as the only allowed network protocol but it welcomes other non-web communication standards. The only requirement is that they are IP-based otherwise they are not usable by other services on the internet. This renders the architecture more flexible and capable of handling more heterogeneous applications and IoT domains. Finally, the Security configurations describe the mechanisms for control access Thing Affordance. They can be public security metadata (i.e., which security protocol should be used) and private metadata (e.g., private keys of a PKI system). Thing behavior is described as machine code with the help of Scripting APIs whereas the other four aspects are expressed declaratively in the Thing Description (See next section). Software agents can then read a Thing Description to understand how to interact with the Web Thing and its capabilities. This action is referred to as ”consume a Thing Descriptio” and a client that performs it is called a Consumer, as depicted in Figure 2.5. Furthermore, Figure 2.6 represents the general overview of WoT architecture distributed on different IoT nodes. It is possible to notice how WoT spread into all the segments of a typical sensor to cloud architecture. For instance, a Web Thing can be implemented directly on a small smart device, but also more complex Things can be installed on edge gateways or even in the cloud. Moreover, even existing devices can participate in a WoT application as long as their protocols and data schemas are describable with a TD. One final important aspect of the proposed architecture is that agents can interact seamlessly in the different layers with the same paradigm. A concrete example could be a smart radio that wants to access local speakers. With the current market solution, the interaction between those devices can happen or via a common intermediary

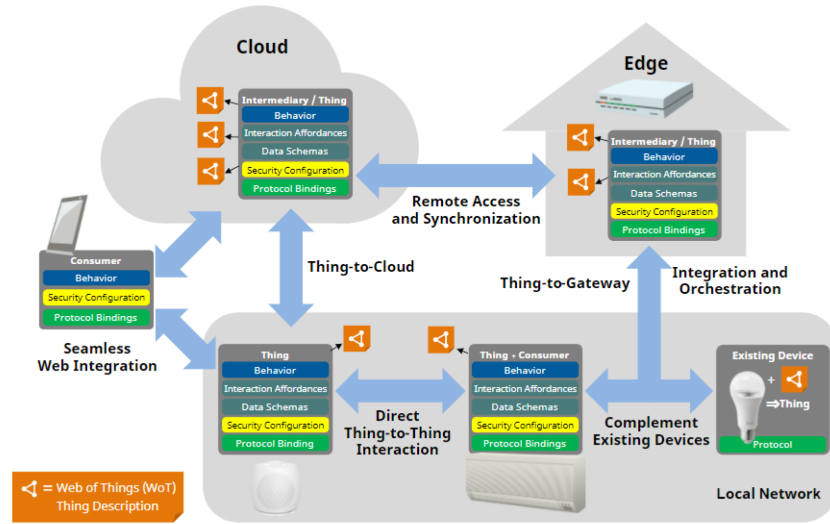


Figure 2.6: An overview of a WoT deployment. Image taken from [10]

(i.e., a smart gateway) or using property protocols diminishing the set of compatible devices. With WoT technologies, the smart radio would discover local Thing Descriptions of installed speakers and interact directly with their preferred protocol. Consequently, it is possible to have direct communication (Figure 2.7) between WoT software agents, or more specifically Servients (See Section 2.3.3). In conclusion, W3C WoT architecture has overcome some interoperability shortcomings of the early ideas, and it is more flexible. For further details refer to W3C WoT Architecture 1.1 document [10]. In the next section, we will describe more in detail how a thing description is defined, the fundamental structure of a servient, and the scripting API.

2.3.1 Thing Description

The Thing Description (TD) is a "formal model and a common representation for a Web of Things" (See [27]). As Hypertext Markup Language describes the content and the shape of a page, the TD model describes physical or virtual interactable resources. In every sense is the entry point

Figure 2.7: Consumed and Exposed Thing diagram. See [10] for futher details

of every Web Thing, as it introduces its purpose and context. The model consists of a set of vocabulary terms that can be understood by machines and humans. Those terms define declaratively a thing instance-specific configuration of the core architectural aspect: affordances, protocols, data schemas, and security configurations. It also provides general metadata like Thing id, name, human-readable description, and links to other web resources. Figure 2.8 portrays vocabulary terms organized as a Unified Model Language classes and properties. In particular, we can find that Thing affordances are categorized into three different types:

- Properties express the status of a Thing or one of its configuration parameters. For example, the last temperature measured or the position of a bistable switch. Properties can be read or written but also observed if the underlying protocol allows it.

- Actions indicate an operation that triggers a process on a Thing. The function may change Thing's internal stat or just provide useful information to the caller. An example of action could be a move operation for a robotic arm.

- Events describe state transitions pushed asynchronously to consumers. For instance, notify users when the temperature rises above a critical threshold. State transitions might not reflect in Thing Property changes, like a Thing that provides an alarm when detecting a robbery.

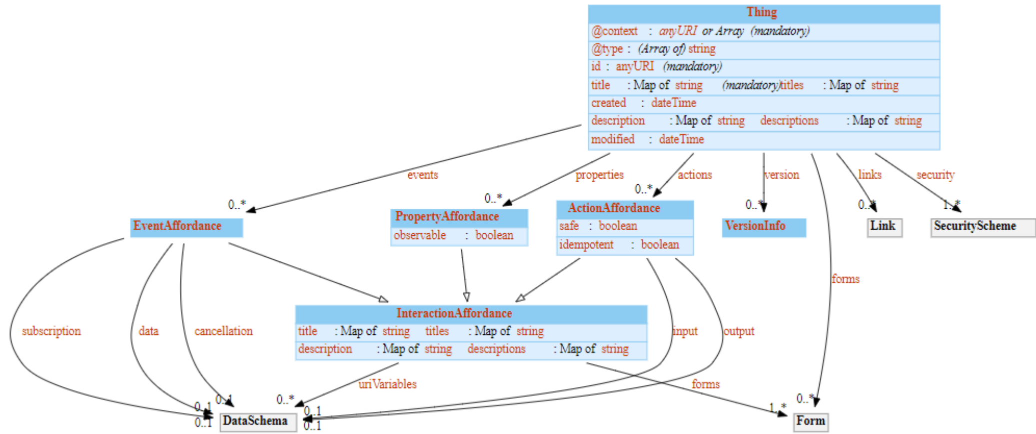


Figure 2.8: The Thing Description data model [30]

Another notable term in the TD model is Form. A form has two goals: it specifies which protocol should be employed to use a particular affordance, and it configures protocol-specific parameters to access it. Forms describe those two aspects thanks to domain-specific vocabularies defined as Protocol Bindings templates (See Section 2.3.2). This allows a seamless extension of the Web Things to other protocols and IoT stacks.

The default serialization format of the TD model is JSON, but it can also contain semantic tags borrowed from the JSON-LD specification. This bi-serialization form allows constraint devices to optimize parsing of TDs and trait them as simple JSON documents. On the other hand, it enables more capable nodes to leverage semantic tags to infer knowledge about the IoT set-up or Thing context. As JSON-LD 1.1 serialization format designates, a JSON document can be semantically enriched using reserved keywords that contextualize its properties. For example, the term "@type" indicates which ontology defined class this TD document implements. Moreover, the keyword "@context" specifies which ontology/vocabulary should be used when interpreting the description. This feature allows translating a TD in an RDF representation similarly to other previous semantic web IoT solutions, but at the same time maintaining the simplicity of JSON.

Listing 2.1 shows a full example of a Thing Description for a smart Lamp.

```
1  {
2    "@context": [
3       "http://www.w3.org/ns/td",
4       {
5       "cov": "http://www.example.org/coap-binding#"
6       }
7     ],
8     "id": "urn:dev:ops:32473-WoTLamp-1234",
9     "title": "MyLampThing",
10    "description": "MyLampThing uses JSON
   serialization",
11    "securityDefinitions": {"psk_sc":{"scheme": "
   psk"}},
12    "security": ["psk_sc"],
13    "properties": {
14        "status": {
15            "description": "Shows the current
   status of the lamp",
16            "type": "string",
17            "forms": [{
18                "op": "readproperty",
19                "href": "coaps://mylamp.example.com
   /status",
20                "cov:methodName": "GET"
21            }]
22        }
23    },
24    "actions": {
25        "toggle": {
26            "description": "Turn on or off the lamp
   ",
27            "forms": [{
```

```
28                  "href": "coaps://mylamp.example.com
   /toggle",
29                  "cov:methodName": "POST"
30              }]
31          }
32      },
33      "events": {
34          "overheating": {
35              "description": "Lamp reaches a critical
    temperature (overheating)",
36              "data": {"type": "string"},
37              "forms": [{
38                  "href": "coaps://mylamp.example.com
   /oh",
39                  "cov:methodName": "GET",
40                  "subprotocol": "cov:observe"
41              }]
42          }
43      }
44  }
```

Listing 2.1: A Thing Description example of smart lamp.

2.3.2 Protocol bindings and Templates

As previously discussed in Section 2.2, various application domains and use cases involve IoT technologies. It is this intrinsic heterogeneity that has sprout specialized protocols and frameworks tailored to one (or few) use case(s). Since no fit-all solution has yet emerged, the Web of Things wants to be as inclusive as possible, avoiding being the N + 1 middleware in the already crowded IoT tech space. Consequently, WoT requirements for protocols are loose: it must be IP based, it must have a defined URL protocol scheme, and it should be mapped to at least one basic operation of the WoT network interaction model. The WoT network interaction model is composed of twelve functions associated with one affordance type.

Figure 2.9: Main WoT network interface operations grouped per affordance type

For instance, "readproperty", "writeproperty", "observeproperty", and "unobserverproperty" belongs to Property affordances and defines what operation can be requested remotely. Figure 2.9 shows the complete list of the available operations for each affordance. The mapping between this abstract interface and the concrete protocol or stack happens throughout the form property of a Thing Description. This declarative method allows instructing implementations on which protocol to use and how to configure it. The instructions are given using a vocabulary defined for that particular IoT solution known as Protocol Binding Template, whereas a particular description given in a form is called Protocol Binding (i.e., binding as a synonym for mapping). As an example take Listing 2.2. In this form is described how a readproperty operation should be performed using the HTTP protocol. The term "htv:method" is a protocol specific concept defined in the HTTP Protocol Binding Template. Specifically, this keyword command consumers to use the HTTP method indicated by its value. Consenquently, for this form example, a software agent will issue an HTTP GET request to `http://thing.example.it/test` when reading the property test.

To conclude, the WoT solution to protocol heterogeneity is addressed with the Protocols Bindings Templates and the Thing Description as shown in Figure 2.10.

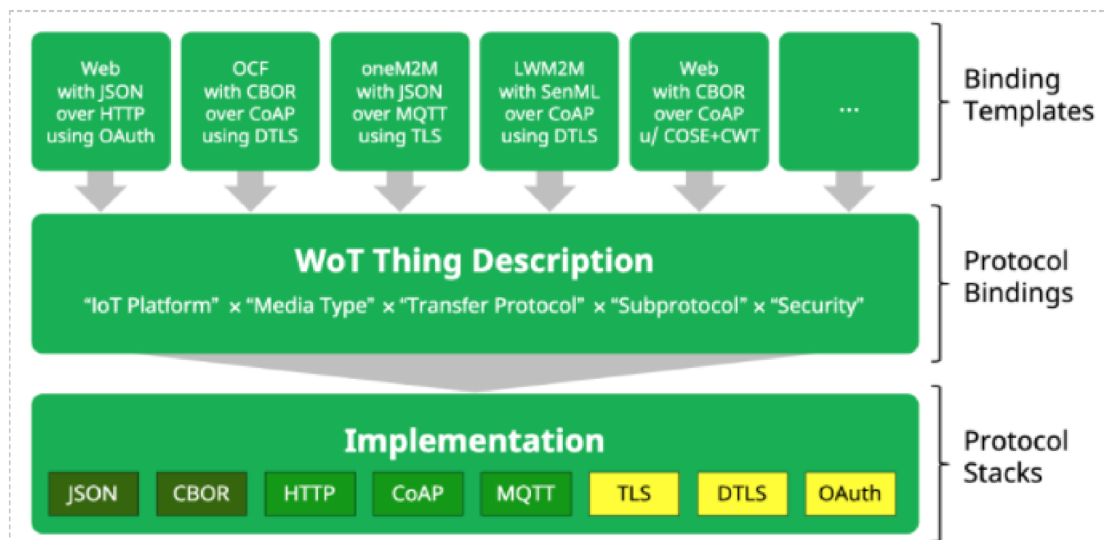


Figure 2.10: Protocol bindings [10].

```
1  {
2      "forms": [
3          {
4              "href": "http://thing.example.it/test",
5              "contentType": "application/json",
6              "op": [
7                  "readproperty"
8              ],
9              "htv:methodName": "GET"
10         }
11     ]
12 }
```

Listing 2.2: An extracted property of a Thing Description. The forms property indicates protocol binding configurations to access a specific affordance

2.3.3 Servient

Starting from the lower level we find the protocol implementation and the system APIs. The system API is the platform-dependent interface from where local hardware/software resources can be accessed and processed (e.g., GPIO software interface a Raspberry PI board). Above this layer resides the WoT Runtime; a middleware where the WoT concepts are exposed as a coherent set of data and functionalities. This block is safely isolated from other local processes, similarly as a browser script context is separated from localhost programs and operating system. Moreover, it conceals security-critical data to the upper levels, and it implements the logic to expose, consume, and interact with Web Things. Practically, it is able to read and create Thing Descriptions and configure the concrete protocol implementation to communicate with a remote Thing. Finally, it might provide a commodity interface for business logic code to easily use the below layers: the Scripting APIs (see Section 2.3.4).

Behavioral scripts implement the business logic of a Web Thing exploiting Scripting APIs to expose/consume services. For instance, the logic to read an I2C sensor could be coded as a WoT script. Then the same code could serve the acquired sensor data as a property affordance with the apposite Scripting API function. Other examples of behavioral scripts are:

- Actuators control and piloting
- System orchestration
- Proxying one or more Web Things

Even if the most natural form of WoT scripts is ECMAScript[20] portable code, the specification does not force any particular coding language. On the contrary, it defines possible alternative implementations of WoT runtimes that support native languages like C/C++.

To summarize, the servients are software agents that play a central role in a WoT architecture. It can be implemented in any language and can offer an ECMAScript runtime environment for high-level bussing logic code. For further details please head over [10] official documentation.

[20] `https://www.ecma-international.org/ecma-262/`

2.3.4 Scripting API

Scripting APIs is an optional block introduced in the WoT runtime servient implementation. It defines a set of opportune functions and data models to define behavior for Consumers, Web Things, and Intermediaries (software agents that proxies one or more Web Things). Usually, this layer is implemented on high computational capable nodes of a WoT deployment as it requires an ECMAScript interpreter.

The APIs can be grouped into two categories: Discovery and Thing management. The former, as the name hints, provides functions to discover other Web Things using the WoT discovery process described in a dedicated specification document [28]. The main entry point of the discovery is the discovery method that accepts as input a filter object. This filter defines how the runtime should perform the discovery and indicates the Thing Description of interest. Currently, the specification is still undergoing, and this API is still experimental. However, it already allows searching for different query types and filters like SPARQL[21] and JSONPath[22]. On the other hand, Thing management API defines functions for consuming and exposing Web Things. In particular, these APIs are split further into twos: exposing and consuming. Exposing has functions to command the generation of a TD description together with the initialization of the runtime protocol stack (e.g., starting an HTTP server or boot up an HTTP connection to a broker). Additionally, this set of function s covers also callbacks setters for the main WoT operations types (i.e., readproperty , invokeaction). For instance, it provides a function to inject business logic behavior when the network request for reading a particular property arrived: the setPropertyReadHandler function. Consuming API represents the other side of a WoT interaction pair. It consists of methods for processing a Thing Description and invokes its affordances. An example is a readProperty function which returns the value of the requested property affordance. This method (and also the other Consuming functionalities) hides the complexities of insuring a communication channel with the remote object. Among other aspects, it correctly authenticates the script, deserializes and

[21] https://www.w3.org/TR/sparql11-query/

[22] https://goessner.net/articles/JsonPath/

validates data, and manages the protocol-specific inner workings. Examples of a WoT script that uses Scripting APIs can be found in the appendix (see Listing A.2).

As this text wants to remain more informative than technical, further specifications about the single function can be found in the WoT public note of Scripting API [31].

This page intentionally left blank.

Chapter 3

Open WoT Monitoring platform

In the IT industry monitoring platforms are mostly focused on software services. This work wants to explore how the same principles can be adapted to physical objects and real-world features of interest. Specifically, it took a bottom-up approach that started from the implementation of two concrete monitoring solutions, and from there a set of generic requirements were factorized and extracted. The monitoring applications were selected within the two macro fields of Structural Health Monitoring and Environmental monitoring; an industrial prognostic application and a smart farming platform. Section 3.4 will provide more details about the two.

This chapter will follow the opposite process (i.e., top-down) starting from the abstract requirements and architecture outline that later will be concretized in two specific use cases. In particular, the following will define the platform requirements and prototype software architecture. It will assess the core contribution related to previous works in Section 3.1 and describe the platform solution in Section 3.3 . Finally, Section 3.4 will provide a description of concrete deployments and implementations.

3.1 Related works

Physical-world monitoring is one of the most natural applications of IoT technology. It is so deep-rooted that we can say that every IoT application has at least one component dedicated to long term observation of a particular physical property. According to a recent survey [32] the taxonomy of IoT applications describes defines as six out of seventeen applications directly involve monitoring procedures, whereas the remaining involve at least some kind of remote observation of resources state. For example, Smart Farming applications have dedicated monitoring services for crop growth and quality. Even the smart home domain has some possible surveillance applications like remote cameras or presence sensors. Therefore, during the years different solutions were proposed in the literature. This section has the goal to guide the reader to related software solutions that were developed to address IoT monitoring in different use-cases with the focus on SHM domain. At the same time, it provides further motivations about how the proposed solution differs from the state of the art.

> **Disclaimer**. The following is accepted to published in IEEE Consumer Communications & Networking Conference see [33]

According to the recent survey in [34], all SHM systems must rely on the integration of three major subsystems, which are, in order of abstraction, the data sensing, the data management and the data analytics layer. The first component includes a wide plethora of sensor devices and sensor networks based on different machine-to-machine communication technologies [35]. Similarly, the data analytics subsystem consists of data processing techniques and analytical models aimed at detecting, localizing and eventually quantifying damages on the monitored structure [36]. The focus of this paper is the data management subsystem or, more generically, the middleware software platforms spanning between the sensing and the data analytics components. To this purpose, the advantages of IoT-based data management approaches versus traditional SHM solutions have been highlighted in [37]. At the same time, [38] discussed the novel issues posed by the IoT paradigm, mainly in terms of scalability, security and interoperability. The literature about IoT–based SHM platforms is quite scarce,

and mainly composed of solutions where the data management layer has been deployed ad–hoc in order to match the characteristics of the sensing subsystem; issues of scalability and extensibility of the framework (e.g. capability to support other sensors/devices beyond the ones used in the proposed experimentation) are barely addressed. The few exceptions are provided in [39] and [40]. In detail, the work in [39] proposes a Wireless Sensor Network (WSN) platform suitable for both short–term (i.e. high sampling frequencies) and long–term (i.e. reduced data rates compliant with severe energy saving constraints) SHM systems; although the focus of the authors is on the sensing and communication technologies, a Service Oriented Architecture (SOA) is designed, including storage, sampling and status monitoring components. Furthermore, a four–layered SHM architecture is proposed in [40], where the cyber part is further divided into signal processing, event detection and monitoring agent; the latter enabling special actuators to notify alarm messages at the occurrence. Beside the two contributions discussed above, we can classify the existing SHM platforms either on the basis of the interface towards the sensing subsystem (i.e. HTTP vs RESTful based approaches) or in terms of involved computational nodes (i.e. cloud vs edge–cloud approaches). Focusing on the HTTP solutions, we cite the sensor-to-platform architecture presented in [41]; here, the sensing unit is constituted by several accelerometers that periodically transmit their measurements through an HTTP channel to a remote software platform that is in charge of displaying the received data on a Web GUI. In [42], a long-term SHM system for the monitoring of historical buildings is proposed and installed on a physical structure (the San Frediano's tower in Lucca). The system includes a set of accelerometers, the HTTP protocol for data acquisition, and a combination of relational/non-relational database management systems for the data storage. A RESTful, open architecture (named SnowFort) is then described in [43]: the platform is fed by sensor nodes communicating with a Zigbee module, and implements a data-pipeline including data storage (for both raw data and processed features), data cleaning and data visualization through a Web GUI. Additionally, the solutions in [41] and [42] envisage a direct sensor–to–platform connection. Vice versa, some studies investigates the use of edge processing units between the sensing and the data management subsystems. This is the case of [44], where

the edge component is constituted by a single–board computer (Raspberry PI) running the algorithms for noise filtering and damage identification, and a remote cloud for data storage. Similarly, authors in [45] describe a low-cost distributed SHM system where most of the functionalities related to vibration acquisition and processing are performed on devices.

3.1.1 Service migration

As Section 3.3.1 will introduce a migration mechanism developed within the open monitoring platform framework, this subsection will serve as a panoramic of the current state of the IoT service migration technologies.

> **Disclaimer**. The following was previously published by IEEE Access [46]

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) [47], although it presents several overlaps with other state-of-art architectures, such as Cloudlet [48], Fog Computing [49], and Follow Me Cloud (FMC) [50]. A detailed illustration of service migration techniques and strategies can be found in [47]; here, the unique challenges of MEC compared to live migration for data centers and to handover management in cellular networks are highlighted. Similarly, in [51], 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 considers delay as the principal indicator of performance [52] 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. [53]). 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 [54], 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 in [55], 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 [56]; 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 [57]. From a performance perspective, the container-based implementation is often considered more suitable for the virtualization at the network edge than the VM-based [58]. This is confirmed by several experimental studies, including [59] 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 [60] enables the migration of active `Javascript` processes between different machines by employing injection mechanisms to track the local state of each function.

3.2 Requirements

The deployment of monitoring systems is a natively interdisciplinary task involving joint research contributions from sensing technologies, data science, software engineers, and domain experts. Therefore, an open monitoring platform has to satisfy the requested features from different stack holders. Not to mention the recent trend of crowdsensing where final users contribute to the monitoring application as sensing devices, in a crowdfunded way. Consequently, every monitoring platform configures itself as a cyber-physical system where humans are highly involved in the feedback loop of different applications. In those systems, we can define two roles: services consumers and services producers. In this context, the ”service” term has to be intended in the most generic sense. In fact, in the software solution in the subject, a service could simply a new sensor added in the general picture. The provider, in this case, takes the role of service producer and should be able to register the new service in the platform so that interested consumers can exploit it. On the other hand, service consumers might not only be software services that want to analyze monitored data but also technicians who want to locate a particular sensor in the field. An open monitoring platform should take in mind the necessities of these two different stakeholder roles without leaning towards a specific IoT application. The challenge is to stay open to changes but closed to specializations, which limits the adoption of the platform in other domains losing its advantages. In principle the platform should satisfy these simple functional requirements:

1. Actors can publish sensing data into the platform
2. The published data should be associated with a specific physical object or one of its properties
3. Actors can describe themselves with metadata
4. The platform should support the publishing of metadata information
5. Actors can restrict the access of the published data
6. If requested the platform should be able to conserve real-time data to be later exposed historical records

7. Actors can consume information available on the platform

Although simplistic these requirements capture all the business logic of a typical monitoring application. They are a superset of the specifications collected from [61] and [62] projects, selected for their horizontal scope and validity across of any monitoring application. If satisfied, we could develop, for example, a dashboard application that shows the latest information about the status of a power grid (described with metadata in the platform), as well as use this status information to optimize resource usage. Moreover, technicians could access historical records about the power grid to assess possible damages or system failures. However, the IoT scenario where we are setting the platform requires more precaution, if we want to fulfill the goal to be open and as general-purpose as possible. Therefore, the following describes a list of the non-functional requirements that the platform should satisfy in order to be deployable in different IoT contexts:

8. The platform should be used also by devices with limited resources (i.e. constraint devices)

9. metadata should always be available in a machine-understandable format. Other human-readable formats might be used but always supported with their machine-understandable counterpart

10. The platform should support more IoT protocols as possible, preferably all IP based protocols

11. The platform should not be OS dependent

12. It should be distributed and support consumers and receivers at different levels of the IoT stack (e.g. edge level interaction)

13. It should allow real-time interaction for close or local nodes but near real-time interaction is accepted when it happens within remote nodes.

14. It must support applications across the sensor-to-cloud spectrum

15. The services provided should scale at different workloads and provide the best effort quality of service.

16. The platform component should be interoperable with market ready sensors, platforms, and future solutions.

If the platform will cover also those requirements, it could be used from different IoT nodes, and actors can access its functionalities from different settings. Furthermore, since it does not requires a specific IoT protocol (on the contrary it requires that most know IoT protocols should be adopted) it can break IoT silos and allows actors from a different domain to exploit non-conventional data sources. For instance, use weather historical data together with accelerometer data to assess the aging of a bridge. Related to protocol support is requirement 16: interoperability. Interoperability is the main motive of this dissertation and represents the novelty of prosed architecture. The goal is to create a platform that can operate with different protocols, devices, software components, deployment technologies, and users. Monitoring data is precious for various domains concealing it inside a single vertical application would limit its value. Sheth [63] identifies four levels of requirements that a software system should satisfy to be fully interoperable:

17. System Interoperability: the software/platform should not be tightened to a specific operating system.

18. Structural interoperability: application data models should agree on the data modeling technique. For example, a relational data model might be incompatible with an object-oriented design.

19. Syntactic interoperability: data should be exchanged with the same formatting or serialization process. At this level, the system could have some degree of freedom if a specific data format has a clear transformation function that could convert it into the desired formatting.

20. Semantic interoperability: this level allows systems to understand the meaning of the exchanged data without manual intervention.

Application support is critical to expanding platform functionalities beyond the physical world. As sensors extend the monitoring framework sensing abilities, applications add brainpower to compute more complex information and act proactively to avoid dangerous outcomes for humans

and the environment. Moreover, the scaling requirement assure that those applications perform sufficiently even in critical scenarios when one or more actors request more data than usual. In summary, the software framework should provide functionalities for publishing and reading monitoring data. Moreover, data sources and monitored objects should themself be described as supporting information. It should be also open to different IoT solutions and devices without excluding any particular IP based protocol. Finally, stakeholders should freely enjoy platform capabilities assuming that they have the access rights to use the resources. A complete list of the targeted platform's users follows:

- Engineers
- Data scientists
- Developers
- Technicians
- Municipalities
- Private companies
- Citizens
- Public agencies
- Researchers

3.3 Architecture outline

We consider the abstract architecture depicted in Figure 3.1. Starting from the bottom we found the sensing layer. This layer has two main responsibilities: acquiring data and expose with it a standard interface. The twos satisfy the first requirement of our monitoring platform that is being able to publish monitoring data in the system. This is possible thanks to the fact that the acquired data is exposed by a standard interface that allows any of

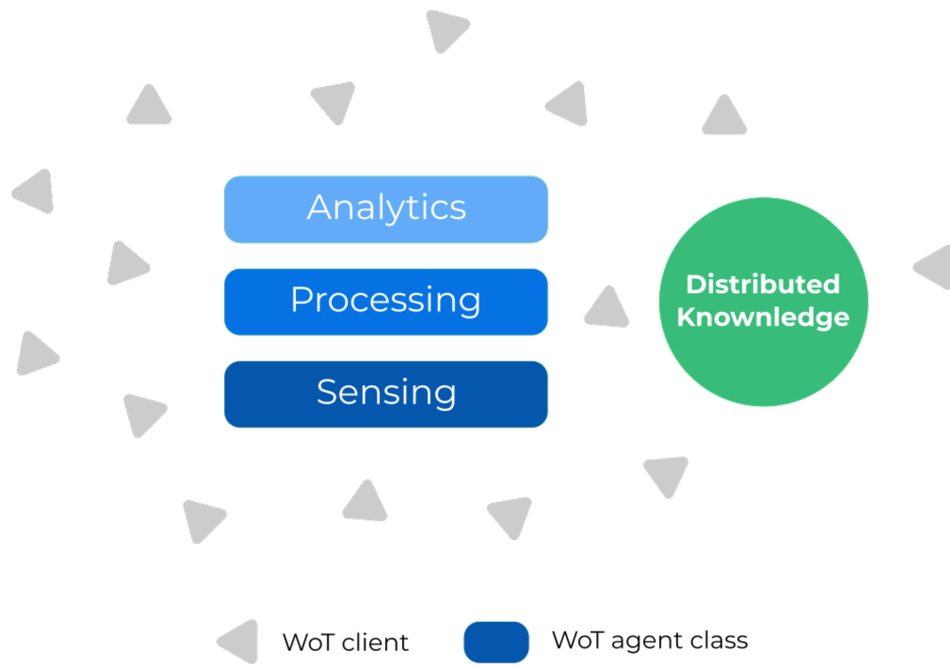


Figure 3.1: The open monitoring platform abstract architecture. Rectangles represent the different agent categories whereas the grey triangles are possible clients interacting at each level.

the consumer actors to fulfill their knowledge needs. The sensing layer plays a similar role to the Connectivity layer of the 2011's WoT architecture. The difference here is that this layer is using the latest WoT specification which is now becoming a standardized interface for device web-based interaction. Furthermore, the current standard is less restrictive regarding the communication protocol used by the different agents. Therefore, sensing data are not forced to be published following a particular pattern or protocol stack. On the contrary, implementers can choose their preferred technology that suited their application-specific needs.For these reasons this layer is implemented following the WoT paradigm, integrating sensors with Thing Descriptions and Servient software agents. The integration can be carried accordingly to one of the possible patterns present in Section 2.3:

- Legacy sensor using one of the support WoT protocol -> integrated with a TD
- Legacy sensor that does not support an IP based protocol -> integrated with System APIs or a WoT intermediary
- Brand new sensor -> implemented with WoT standard in mind (i.e., Servient stack support)

It is important that at this level data is published as raw as possible or if any processing has been performed that the procedure used is well described as metadata of the monitoring information. The raw data allows the upper layer (usually more capable in terms of computing power) to tap into a greater knowledge pool and consequently offer more complex services interoperably. For example, consider a temperature sensor represented in the sensing layer. Its virtual representation publishes a new temperature measurement every hour as the mean of 3600 temperature reading happened in that hour. Since the mean operation loses some level of information (e.g., rare high-frequency changes), the upper layers have no access to this more accurate representation of the physical property. Consequently, this limits the possible applications that could have been developed if the missing information was available. Besides, since the mean is a straightforward operation and the upper layers have more computational power, they

could calculate this information by themselves. Of course, there might be some circumstances where accurate raw data measuring cannot be exposed to other actors. This is the case when the protocol high bandwidth constraints that ceiling the amount of data transferable or when the device has particularly strict energy preservation needs. However, these are physical limits and not software designing choices, as such, they should be clearly stated in the sensor metadata.

Speaking of which, Thing Descriptions are responsible to expose measurement and sensor metadata. There, publisher actors can state contextual information about the sensor and the physical property that it is measuring. As we will see later, contextual information allows us to state the location of the sensor or its association to a particular set of peers. The fact that this is information is directly accessible in the description permits any consumer that has access to it to exploit the data for its needs without prior knowledge. Consequently, also the requirements number 2 and part of the 3 are satisfied.

Climbing the software architecture, we encounter the next layer: processing. While the sensing layer is almost every time confined in the extreme edge (i.e., sensors or small gateways) the processing layer spans across different IoT levels; although it is mostly concentrated at the edge rather than the cloud. At this layer, raw sensor data is processed and aggregated to provide more value to possible consumers. In the proposed architecture, we provide a useful design pattern to export the processed data in an interoperable way. As web things can also represent virtual entities [10], raw data transformation processes can be expressed as special virtual Web Things, called Virtual Sensors. For instance, the raw video feed of a camera can contain a variety of unprocessed information that can be extracted. A virtual sensor, in this case, would connect with the remote camera and extract, for example, the position of an object in the view; we can call this virtual position sensor. It is not associated with a particular hardware board but it still feeds the platform with vital information about the physical world. Another example is an agent that provides short term storage for sensor data. As we have discussed, the sensing layer is assumed to have low computing capabilities and consequently, it is often incapable to store the sensed data for a certain amount of time. On the other hand, in

the processing layer, we could define a virtual sensor that store the data acquired for the below layer and exposed it to a temporal series of samples. it could even provide means to query these data and select only a portion. Still, storage capabilities are limited in the processing layer, therefore, long term storage services are considered more advanced a moved up in the upper layer.

These virtual sensors could be implemented with any technology but forcing it to be a Web Thing promote the creation of computational chains that leverage on WoT standard. Consequently, it encourages the creation of multi-agent applications which has a protocol-agnostic interface (i.e., WoT network interface). Simple consumers can still interact with this set of virtual things as typical WoT clients assuring a common interface within the platform. Although encouraged this design pattern proposal does not exclude the presence of other microservices or processes that exploited the sensing layer. Since the architecture is indirectly actor based they can coexist in the same environment, even if they lose some level of interoperability with other agents in the platform.

Finally, the upper layer, the analytical. At this level, services have a more global view of the monitoring deployment and can have access to collaborative knowledge created by the other layers. Therefore, their typical goals are to extract complex models from the acquired data, take a decision, and store monitoring data in long term databases (requirement 5). The analytical layer is located at the highest level in the IoT deployment spectrum since it requires more computation and storage capabilities. The typical role of analytical services is the consumer, however, they can become sources of monitoring knowledge. In these circumstances, we can exploit again the patter of Virtual Things by taking it to a full representation of system or object: Virtual System. A virtual system is a digital twin of the monitored object, and it differentiates from a Virtual Sensor for its scope. While a virtual sensor aims to model a specif data source, a virtual system represents an interconnection of different virtual and physical sensors, together with their metadata and analytical mathematical models. An example could be a bridge represented as the collection of its sensors; together with the model that extracts its aging level and the metadata that describes its location and physical components. As with virtual sensor,

they could be implemented with any technology, however, exposing them as Virtual Web Things brings, again, interoperability advantages as requested in Section 3.2.

Web Things are not the only actors in the system; simple applications like dashboards, Maintainance expert systems, design aid, control software, etc, represent other possible consumers of the data produced by the platform in the subject. Thanks to the distributed nature of the system these agents might operate across different levels. For instance, a sensor calibration application might operate directly with the sensing layer while a plant monitoring dashboard could interact with the upper layers (e.g., take some information from the analytical layer). Since the software actors interact heavily with the WoT ecosystem an implementation suggestion is to code them as WoT scripts. The advantage would be greater portability and more natural interaction with the other actors in the system.

The layered architecture represents different categories of actors divided by their role in knowledge production. The more complex is the information that they produce, the higher they are in the logical structure of the platform. As we climb up in the hierarchy the knowledge of the above layer is consumed and transformed, creating a logical interconnection between the producer and the processor. Therefore, logically related agents create a network of computation nodes similarly to actor-based software architectures [64]. Picture 3.2 shows schematically this web of nodes distributed in the IoT deployment spectrum. As mentioned, sensing nodes are the basic sources of information and they live close to sensor devices, but rarely they might be in other levels like when sensors are proxied with gateways or are existing devices that have a sensor-to-cloud infrastructure. On the other hand, processing actors are located primarily on the fog/edge nodes and on cloud infrastructures for convenience or computational requirements. These actors are luckily not bound to particular hardware or host resources, hence they are good candidates to be moved across the different nodes to fulfill dynamic system loadings. As explained in Section 3.3.1 virtual things can migrate closer to the data source of interest or offload to less crowded computational devices.

Coming back to Figure 3.1, it depicts how the system creates a shared knowledge distributed across different levels. The knowledge is represented

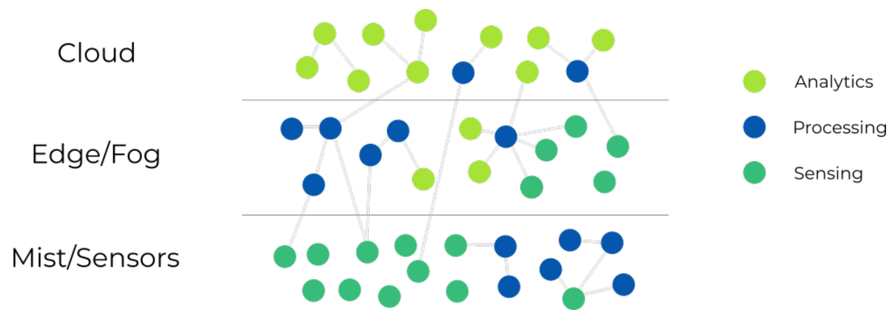


Figure 3.2: A zoomed view of the open monitoring architecture. Notice how the different layers are distributed across multiple IoT nodes.

as the sum of all the contextual metadata published by the actors in the system. An example is the location of a sensor, its capabilities (i.e., its TD), the properties that it is measuring. But also other more complex descriptions like what is the monitored object, is it a bridge? how many sensors are installed on it? where is it located? etc. Most of these pieces of information are scattered across the different thing description, consequently how it is possible to access them straightforwardly? As Section 3.3.2 will describe the discovery subsystem provides a mechanism to find interesting TDs in the monitoring software space. Together with the link support, metadata can be explored as a web of different facts and notions. Moreover, the employment of linked data technologies will increase the expressibility of metadata information given the ability to express facts that do not concern only the actors in the systems. For instance, the time of growth of a particular crop is shared information that does not fit any single Thing Description. Even if we employ a virtual thing this information would belong to every crop instance creating duplicated metadata and fragmentation. A good design pattern would be to describe it using a set of RDF terms publically shared on the web platform. If necessary, these web resources could be linked in TDs so that consumers might follow the link to know more about a particular aspect of the represented physical resources. Taking again the example of the crop growth, we could model our particular crop with a Virtual System and, in its TD, link the correct development stage descriptions published on the web.

Shared knowledge without a shared vocabulary would influence negatively the overall interoperability of the platform. Picture 3.3 shows how the proposed solution uses three vocabulary levels: Web of Things, Monitoring, and Domain. The first one expresses how to interact with the software agents and coincides with the core TD vocabulary presented in Section 2.3.1. For example, the term property means a device affordance which provides information about device status or physical measurements. This level is critical to assure common accessibility of the resources provided in the platform. Without it, means that machine-to-machine interaction should be planned ahead of time and in the worst-case ad-hoc protocols would be used, resulting in worsening the interoperability and the extensibility of the system. The next vocabulary consists of the terms used to describe monitoring entities; those include sensors, measurements, accuracy, precision, timestamp, unit of measure, samples, procedures, collection of sensors, sensor deployment, their location, time series, and relation to the measured object or property. This other level is distinct from basic interaction descriptions given in the WoT layer and serves as a common ground to express particular types of Things and Affordances. Thanks to this layer an implementer could state that a particular web thing is a sensor and its properties represent measurements. Moreover, he/she can describe how sensors are connected, where they are placed, and which unit of measure they are using. Finally, the last layer express domain-specific knowledge that is mostly application dependent. An SHM application will have a specific feature of interest that is completely unnecessary in a Smart agriculture context like material stiffness versus plant stress. However, some concepts might still overlap; consequently this layer is divided in two: common scientific and specif knowledge vocabularies. The scientific vocabulary contains common scientific concepts and properties like temperature and humidity. On the other hand, specific knowledge, as the name entitles, identifies key vocabs that are not expressed in the lower vocabularies; simple examples are the term bridge, or plant, or crop. Therefore, with this final block actors can state that the object being monitored is a crop of soy, and sensor 1235 is measuring the soil water content of the field where it is installed.

One final aspect is the Accessibility/Sharability of the data published inside the monitoring framework. We consider this feature resolved by the

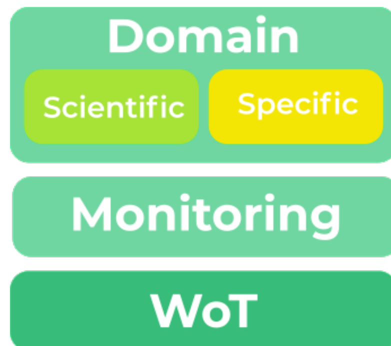


Figure 3.3: A possible knowledge layered organization for WoT based monitoring applications.

underlying Web of Thing paradigm and technology. Furthermore, when knowledge is stored in specific non- WoT based services, technologies like oAuth 2.0 might be employed to cover the requirement 5. In conclusion, the proposed architecture solve the presented requirements as following:

- The sensing layer solves requirements 1, 3, and 7; sensor data is published thanks to a Web of Things interface and consumers can read data assuming they know how to interpret a Thing Description.
- Linked Data technologies and the distributed set of TDs cover requirements 4 and 2
- WoT security technology fulfill requirement 5
- Analytical and processing layers satisfy requirement 6 since a virtual sensor may store short term data and a Virtual System could store long term monitoring information
- The layered vocabulary choice should limit interoperability issues and together with the TDs will provide machine-understandable metadata (requirement 9)
- Migration supports requirements 15 and 14

- Constraint devices are integrated thanks to the sensing layer (requirement 8)
- Other non-functional requirements are covered thanks to WoT paradigm (e.g., WoT support a number of different IP based protocols, requirement 10, and it allows agent interaction across all IoT levels, requirement 12)

3.3.1 Migration

> **Disclaimer**. This section is an adaptation of what was previously published by IEEE Access [46]

The remarkable growth of connected devices produced by the Internet of Things (IoT) can be explained by the flexibility of its model, which applies to a wide range of different applications, from digital manufacturing to smart cities and environmental monitoring [65]. Service mobility has gained significant importance for various purposes in these domains. On the one hand numerous large-scale IoT applications operate in dynamic environments: software solutions are therefore needed to adjust to rapid changes in bandwidth/computational resources, the number of connected devices, and service requirements. Several IoT platforms like [66] [54] provide such a layer of adaptation by supporting the seamless software mobility among the nodes of an edge-cloud continuum.

On the other hand, mobile IoT devices that produce space/time-variant data streams are further pushing research into scalable computational architectures that can self-configure themselves to meet the quality of service (QoS) for user applications [67]. This is the case of Mobile Edge Computing (MEC) [47] architecture (and closely related concepts such as Cloudlet [48], Fog Computing [49], and Follow Me Cloud [50]) that aim at running processing tasks in the proximity of the data sources.The ability to unload computing resources on the edge/fog servers nearest to the current user location is a key component of MEC architectures [47], mostly by means of container/virtual machines (VMs) mobility techniques [56] [57], and relocation policies driven by the physical mobility of IoT devices [53].

Service migration is not the only open challenge in the IoT landscape, as we widley discussed in Section 2.2. Most of the IoT environments are characterized by the heterogeneity of hardware and software components, as well as by the dynamicity of their interactions. Interoperability issues are estimated to reduce up to of 40% the potential revenues [6]. At the same time, novel business opportunities can swell by enabling different IoT systems to communicate together [6].

In this section, the two previously stated IoT issues (i.e. service migration and service interoperability) are discussed from a WoT perspective: more precisely, we strive to expand WoT abilties to dynamic IoT environments by fostering dynamic WT orchestration and mobility among the available computational resources of the full IoT spectrum (edge/fog/cloud nodes). The WT migration provides novel opportunities compared to current software mobility approaches in the MEC literature e.g. [47], [68]. Indeed, since the WTs interactions are defined by uniform software interfaces (i.e. the TDs), it is possible to engineer fine-grained and adaptive allocation policies; by taking into account the real-world network and computational load conditions and with far lower implementation complexity for service monitoring than other ad-hoc solutions, such policies will move groups of WTs to satisfy system-wide QoS requirements. At the same time, a WT's mobility from one node to another may affect the operations of other WTs using it. Therefore, to control the WT handoff and to ensure device continuity, innovative solutions must be implemented. The work presented in [46] answers research questions relating to the processes for WT migration and WT migration policies, i.e:

- How to enable the seamless migration of a WT between two nodes?
- How to optimize the performance of a WoT deployment by orchestrating the WT allocations on a cloud-edge continuum?

In that paper, we propose the Migratable Web of Things (M-WoT), a novel architectural framework supporting the dynamic allocation of W3C WTs to the available computational nodes. Specifically, by managing the handoff process for WT consumers, we investigate how to allow *stateful* migration of WTs. At the same time, it defines a particular agent called

the Orchestrator. An Orchestration in M-WoT is responsible for tracking WT interactions and calculating the optimal allocation of WTs to nodes on the basis of high-level policies(e.g. data locality maximization, latency minimization, etc). More in detail, three main contributions are provided by [46]:

- On two chosen IoT use-cases, we examine the benefits of WT migration mechanisms, and then we describe the components of the M-WoT software architecture.
- We formulate the WT allocation as a multi-objective optimization problem. After, the paper proposes a centralized heuristic that aims to balance the load of inter-host communication (generated by the interactions between WTs) and each host's computational load.
- M-WoT operations are validated by two test beds. First, in edge computing scenarios, we test the efficiency of various allocation policies where we vary the number of WTs and their interactions. Second, we explore the efficacy of the M-WoT system on a generic IoT monitoring scenario in which real-time diagnostic services are dynamically transferred from cloud to edge nodes based on background conditions

The evaluation analysis shows that when compared to greedy policies, the suggested heuristic can efficiently balance the inter-host communication and the computational load. In addition, the M-WoT solution is able to efficiently reduce the diagnostic latency compared to a state-of-the-art, no-migrate approach in the IoT monitoring use case. This thesis reports a summary of the results presented in [46]. In particular, in the following we describe the M-WoT archietecture together with a concrete example of the migration process. Finally, we introduce the policies defined and in Section 4.2 we will discuss the experimental results obtained during the evaluation of the migration mechanism.

The M-WoT software architecture is depicted in Figure 3.4. We assume a set of W3C WoT Servients, deployed on different nodes; each Servient hosts exactly one WT. Differently from a legacy static W3C WoT deployment, the M-WoT enables WT mobility between different nodes. To achive this goal,

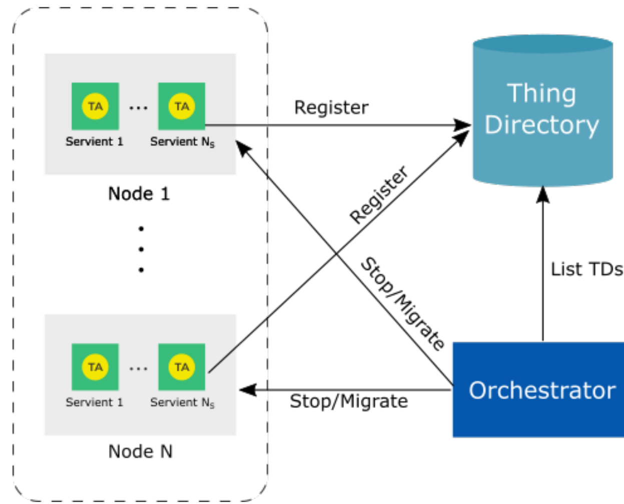


Figure 3.4: Main system components of a Migratable WoT deployment.

the M-WoT features two novel components: respectively the Orchestrator and the Thing Directory; these modules are statically deployed and can be installed either on the edge (if the computational requirements are met) or on cloud servers. In addition, the Servient software stack is augmented with a Monitoring Layer (see Section 3.3.1.3). In the following, we detail the internal structure of the three software components, while in Section 3.3.1.4 we clarify the modules' operations when a WT migration process occurs.

3.3.1.1 Thing Directory

The Thing Directory (TDir) serves as registry of the M-WoT resources, i.e. of the active Thing Descriptors (TDs) [1]. In more depth, two types of TDs are possible in a M-WoT system, one associated with WTs and one associated with Servients; the latter defines the runtime environment capabilities and is used to allow the monitoring layer functionality listed in Section 3.3.1.3. Each Servient records its TD and the TD of the hosted WT on the TDir

[1] When we worked on [46] the Thing Description Directory was not yet standardized by the W3C. The reader can think the TDir as an early concept of a TDD.

once activated. Then, two major roles are performed by TDir. First, it acts as a discovery service, i.e., when queried by clients it returns a list of TDs that follow the demand criteria; thus, the Orchestrator module will be aware of the list of Servient currently available in the WoT deployement. Second, it supports generic push notifications towards WTs/Servients once specific system-wide events are detected, for instance a WT handoff completion. It is critical in thing migration scenarios that this module supports push notification mechanisms. This is due to the dynamism of the environment; moreover it is practical to quickly restore a consistent state of the system and, at the same time, minimizing the network usage. For example, 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 is broken unless T_2 is prompted about the mobility event and the new service location. The notification process is depicted in the sequence diagram of Figure 3.7, discussed later in Section 3.3.1.4. 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.

3.3.1.2 WT Orchestrator

The critical component of the M-WoT design is the Orchestrator. It uses the TDir to retrieve the list of active servients(i.e., the list of their TDs), as explained before. In order to gather live data, such as the use of the CPUs and the network traffic produced by the WT connections, it then regularly queries each servient exploiting servient WoT interface. Consequently, the Orchestrator decides the appropriate allocation of WTs/Servients to nodes based on the obtained metric values and on the optimization policy in place. The allocation plan is then transferred to an underlying module (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. During the system lifespan, the above steps are constantly executed by the Orchestrator; as a result, the dynamicity of the IoT/ WoT environment is completely supported in terms of WT

creation/disposal, network bandwidth variation, policy update at run-time. Moreover, in order to favour the platform extensibility, the structure of the Orchestrator has been modularized into the three main sublayers of Figure 3.5, 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 3.3.1.5 and the other greedy policies evaluated in Section 4.2; 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 rely on any specific software mobility technology. Instead, we have introduced an abstraction layer, called the Migration Substrate, which can employ any state-of-art solution (via proper migration connectors), such as Docker containers, VMs, or Javascript processes [47] [60]. Those connectors will perform the Optimizer output plan received as input. Concretely, the current implementation relies on Docker Swarm as a default migration connector.

3.3.1.3 M-WoT Servient

Finally, the M-WoT framework introduces light modifications to the Servient runtime [69] in order to feed the Optimizer with real-time data about system performance. More precisely, a Monitoring API layer has been introduced

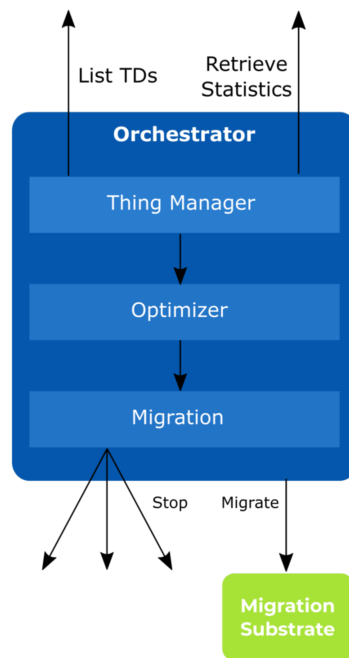


Figure 3.5: The main modules of a Orchestrator agent. Three modules cooperate with the goal to find the optimal allocation plan and actuate it thanks to the migration substrate.

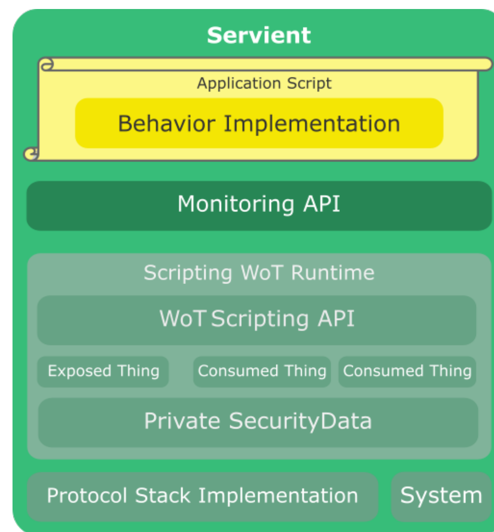


Figure 3.6: The internal software stack of a migratable servient. See the new added module dedicated to the extraction of monitoring parameters about application behaviour

between the WT business logic and the Scripting WoT runtime, as seen in Figure 3.6. The layer is responsible for the interception of Scripting API invocations and for the generation of periodic Thing Reports (`TR`s). The latter can be considered a snapshot of the current Servient/WT execution state, and it contains the metrics' values (both for the Servient and WT) required by the Optimizer; in the Appendix A we report a fragment of the `TR` structure in use. The Monitoring layer provides all the data collected trough a proper Action Affordance, which is listed in the Servient TD; by invoking it, the Orchestrator or another consumer can issue request for `TR` generation to the Servient.

3.3.1.4 Migration example

To summarize the operations of the three components presented so far, we provide an example of WT migration process. We evaluate two WTs/Servients, respectively T_A/S_A and T_B/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 `TR`s; this is implemented by consuming the TDs of the Servients and issuing a `retrieveReport` command (details in [46]). 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 are shown in Figure 3.7. 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 register 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

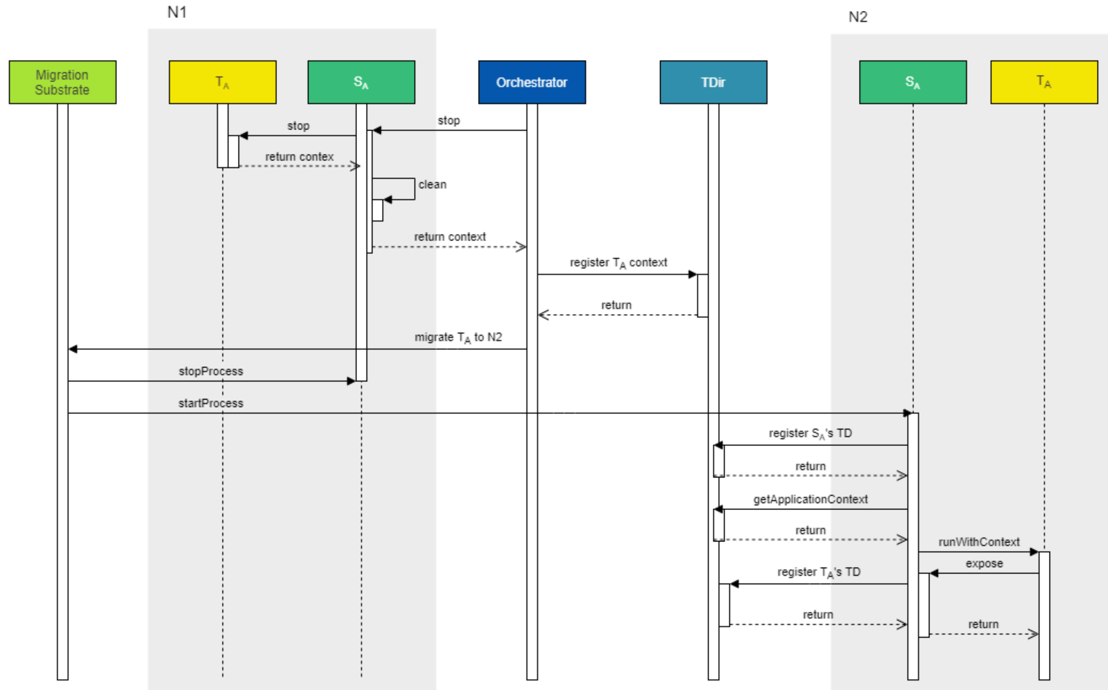


Figure 3.7: Sequence diagram of a WT migration event.

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.

3.3.1.5 Policies

In the following, we introduce the M-WoT migration problem informally together with a set of possible policies evaluated in Section 4.2. For an in-depth mathematical illustration, please refer to [46]. In this study, we considered a twofold optimization process that takes into account the load-

balancing issue (i.e., how much each host[2] is loaded) and the network communication overhead (i.e., how much data is exchanged among hosts). Simply speaking, giving a number of different nodes where a set of Web Things can be deployed, M-WoT tries to find the optimal distribution of those WTs such as the Host Fairness metric is less than a user-defined parameter (Δ) and Network Overload metric is minimized. Specifically, the Host Fairness measures the difference in terms of computational load between the most loaded and most unloaded host of the cluster (i.e., the set of nodes managed by M-WoT). On the other hand, Network Overload takes into account the total inter-host communication load (in bytes) occurring due to interactions among WTs hosted by different nodes. Notice that in M-WoT inter-host communication happens always as a call to a remote Web Thing interaction affordance. Consequently, the number of bytes transmitted is a linear combination of the amount of remote interaction affordance calls.

The two distance functions are titled coupled: minimizing the network load can be achieved by allocating all the WTs to the same host, whereas it is the worst case for the Host Fairness. This is why the Δ hyperparameter is employed as a tuning knob to select how much the system should strive in the balancing of the computational load. More in detail two extreme cases are possible:

1. The system goal is to minimize the data transmitted over the network, regardless of the service latency. This might the case of an edge-cloud IoT scenario, where the manager is interested in minimizing the amount of data exchanged toward a remote infrastructure for privacy reasons. In this case, $\Delta = \infty$

2. 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, $\Delta = 1$

In this context, a strategy that tries to solve the migration problem

[2]The terms hosts and nodes are used interchangeably

given a delta as input is called policy. In [46], we evaluated 4 different policies:

- NoMigrate: do not perform any migration.
- Greedy NetLoad: this is a greedy policy that aims at minimizing Network Overload. At specific intervals, it selects the WT producing the highest network traffic and it migrates it towards the same node of the consumer WT.
- Greedy CPULoad: another greedy policy but that minimizes the host load metric. It focuses on 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: We defined an exact policy that leverages on the interaction graph created by the managed set of WTs. It is able to obtain the optimal configuration in N defined steps outperforming the other policies in most of the simulated test cases. For this policy, we define three different sub configurations:
  - $\Delta = \infty$: the policy aims exclusively at minimizing the Network Overload metric, while no load-balancing action is executed.
  - $\Delta = 5$: The balance parameter is in a mid configuration. The policy computes a minimal Network Overload solution ensuring that the host load cannot exceed the Δ threshold equal to 5.
  - $\Delta = 1$: this is similar to the previous policy, however, we set the system to provide an even distribution of the WTs allocations over the nodes of the cluster.

In conclusion, the M-WoT framework aims to solve a mathematical allocation problem, optimizing different metrics. In [46] we defined a set of policies (i.e., strategies) to find exact and approximated solutions of the optimal deployment. Finally, Section 4.2 describes the experimental results obtained in the evaluation of those policies in a virtual set up.

3.3.2 Discovery

As discussed in Section 3.3, the actors of the platform create a distributed web of knowledge. The discovery process is the action of navigating this web looking for particular information or fact. An example is a consumer that wants to know the nearby sensors or which monitored structures are in the desired area. Since the knowledge has a distributed nature, it does not have a single point of access which requires the definition of discovery techniques. Unfortunately, at the time of this document, the W3C standardization process for discovering is still ongoing. Therefore, the following will go through possible discovery ideas explored during those three years of research and development.

In the platform in the subject, the central source of information is the Thing Descriptions of the web things deployed on a particular setup; they are the starting point to obtain raw or processed sensor data. TDs are service descriptors of physical/virtual devices, and as such, its discovery is a particular type of service discovery. In [70], the authors review the discovery mechanisms used in different frameworks and provides a taxonomy to categorize them. Although similar to other frameworks (e.g., UPnP[3]), WoT deployments have unique challenges when it comes to discovery. The major WoT strength, protocol agnostic, hinders the definition of a network discovery protocol specific for WoT deployments. To overcome this issue, the current practice is to split the discovery process into two phases. In the first step, application-specific discovery protocols retrieve a list of TDs URLs, while in the second phase, consumers can access those URLs to access the desired information. Employing this method allows defining a boundary where only authorized actors can read and use TD knowledge and device capabilities. After phase one, URL access can be closed using one authentication method suitable for the specific protocol. For instance, an HTTP URL might force the requester to authenticate using OAuth 2.0[4] whereas an HTTP URL might require an HTTP username and password credentials. Remembering the requirement 5 is critical to assure confiden-

[3] https://docs.osgi.org/specification/osgi.cmpn/7.0.0/service.upnp.html

[4] https://tools.ietf.org/html/rfc6749

tiality in the system. Phase one protocols may differ from one application to another, but WoT runtimes and Scripting API may provide an abstraction layer so that scripts could discover protocol agnostically Thing Descriptions. Considering again [70], the authors provide a list of possible mechanisms for service discovery; most of those methods can as well be employed in a WoT scenario. For example, the mDNS protocol might be used to collect network-specific servient locations. When the application requests a search in the local network, the runtime would go through each address and ask for the list of the exposed Thing Descriptions.

Usually, phase one protocols do not support complex queries, due to the challenges on query propagation and processing in P2P networks [70]. However, phase two can provide richer filtering methods thanks to the support of super-nodes called Thing Description Directories. The TDDs are particular WoT actors that serve as collection points of TDs and are as well described with a particular TD type. In its simplest form, a TDD is a web resource containing a list of descriptions, but more sophisticated services allow TD publishing and filtering using different query methods. The working specification draft suggests two possible functions for TD filtering: JSONPath and SPARQL. JSONPath provides a syntactic filter to choose the structure of the desired Thing Description. For instance, retrieve all TDs document that has a property affordance the name status. On the other hand, SPARQL may be employed to issue semantic base queries like: find the TDs which represents a switch device. Although richer, the full capabilities of SPARQL queries are yet to be explored. However, they provide a hint about how to discover other non-TD knowledge published in our monitoring platform.

Non-TD knowledge may be published as a web resource (i.e., a document retrievable using HTTP) and linked using Thing Descriptions. However, linked resources are not comfortably queryable; for instance, JSONPath cannot use link contents in its filters. Neither SPARQL, even if it has some mechanism to fetch remote documents (i.e., FROM and SERVICE clause), cannot use the linked content to decide if the TD is application side compelling or not. Moreover, some monitoring relevant data are not always linked in TDs because it is generic contextual information like weather forecasts or how many citizens lives in a determined area(analytical services

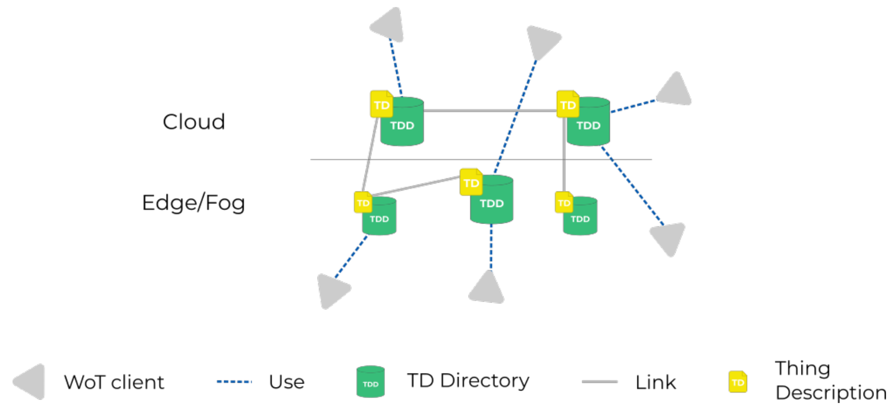


Figure 3.8: The distribution of Thing Description Directories across different IoT layers. It can noticed how different clients can access the knowledge graph from distributed phisical locations.

might use this knowledge to predict possible risks). Therefore, it is useful to publish these pieces of information as RDF liked data in a SPARQL endpoint. A SPARQL endpoint is an HTTP service that supports the SPARQL 1.1 protocol which allows users to update and query a set of RDF resources. This has two benefits: first, every piece of information contained inside a SPARQL endpoint is machine-understandable (requirement 9), second, it is queryable with a query language. As also Thing Description Directories might support SPARQL, we could use SPARQL endpoints to store also TDs so that everything is presented in a coherent interface. Moreover, SPARQL TDDs can be connected in a complex net thanks to the Federated Query support of the protocol. Basically, with SPARQL Federated Queries, users can define a list of SPARQL endpoints that should process a particular query, then the first SPARQL endpoint takes charge to forward it and join the obtained results. This gives the opportunity to evenly distribute monitoring knowledge across IoT layers. An example is shown in Figure 3.8, where local TDDs connect with other TDDs thanks to links in their Thing Descriptors and different users access the same knowledge base from different entry points.

Storing TDs and other contextual information in SPARQL endpoints

solve the need to have a common access point for discovery and knowledge consumption. Although less than sensor data, contextual information is still subject to changes during the time; information like the number of the installed sensors, predictions about the weather, stock market values represents all examples of dynamic contextual data. At the same time, SPARQL 1.1 protocol does not support query subscription, that is the ability to obtain notifications about changes in a query result. In the following, we will describe a possible software architecture to solve this problem and propose a Thing Description Directory that leverage this solution.

3.3.2.1 SPARQL Event Processing Architecture

> **Disclaimer**. This section was previously published by MDPI [71]

Nowadays, the Web of Data is becoming a Web of Dynamic Data, where detecting, communicating and tracking the evolution of data changes play crucial roles and open new research questions [72, 73]. Moreover, detecting data changes is functional to enable the development of distributed Web of Data applications where software agents may interact and synchronize through the knowledge base [74,75]. The need for solutions on detecting and communicating data changes over the Web of Data has been emphasized in the past few years by research focused on enabling interoperability in the Internet of Things through the use of Semantic Web technologies (like the ones shown in [76–79] just to cite a few). Last but not least, an attempt made by W3C is represented by the Web of Things working group and by the Linked Data Notifications (Linked Data Notifications, W3C Recommendation 2 May 2017, [5]) released in 2017 that provide the recommendations to enable notifications over Linked Data.

In this section, we propose a decentralized Web-based software architecture, named SEPA (SPARQL Event Processing Architecture) built on top of the authors' experience acquired developing an open interoperability platform for smart space applications [80–90]. SEPA derives and extends the architecture presented in [91] through the use of standard Linked Data

[5] `https://www.w3.org/TR/ldn/`

technologies and protocols. It enables the detection and communication of changes over the Web of Data by means of a content-based publish-subscribe mechanism where the W3C SPARQL 1.1 Update and Query languages are fully supported respectively by publishers and subscribers. SEPA is built on top of the SPARQL 1.1 Protocol and introduces the SPARQL 1.1 Secure Event protocol and the SPARQL 1.1 Subscribe Language as a means for conveying and expressing subscription requests and notifications.

In particular, assuming an event as “any change in an RDF store”, SEPA has been mainly designed to enable event detection and distribution. The core element of SEPA is its broker (see Figure 3.9): it implements a content-based publish-subscribe mechanism where publishers and subscribers use respectively SPARQL 1.1 Updates (i.e., to generate events) and SPARQL 1.1 Queries (i.e., to subscribe to events). In particular, at subscription time, subscribers receive the SPARQL query results. Subsequent notifications about events (i.e., changes in the RDF knowledge base) are expressed in terms of added and removed query results since the previous notification. With this approach, subscribes can easily track the evolution of the query results (i.e., the context), with the lowest impact on the network bandwidth (i.e., the entire results set is not sent every time, but just the delta of the results). The SEPA broker design is detailed in Section 3.3.2.2.

In [71], we propose the SPARQL 1.1 Secure Event (SE) Protocol and the SPARQL 1.1 Subscribe Language presented, along with the mechanisms to support client and server authentication, data encryption and message integrity. The SPARQL 1.1 SE Protocol allows agents to interact with the broker like with a standard SPARQL Protocol service (also know as the SPARQL endpoint), but at the same time, it allows one to convey subscriptions and notifications expressed according to the SPARQL 1.1 Subscribe Language. SEPA provides developers with a design pattern where an application is constituted by a collection of agents. As shown in Figure 3.9, each agent plays a specific role within an application (i.e., producer, aggregator or consumer) and can be shared among different applications. While a producer publishes events by means of a SPARQL 1.1 Update, a consumer is subscribed to specific events through a SPARQL 1.1 Query. An aggregator plays both roles: it is subscribed to events and generates new events based on the received notifications. The application

design pattern introduced by SEPA, along with the application domains that may benefit from the adoption of this model.

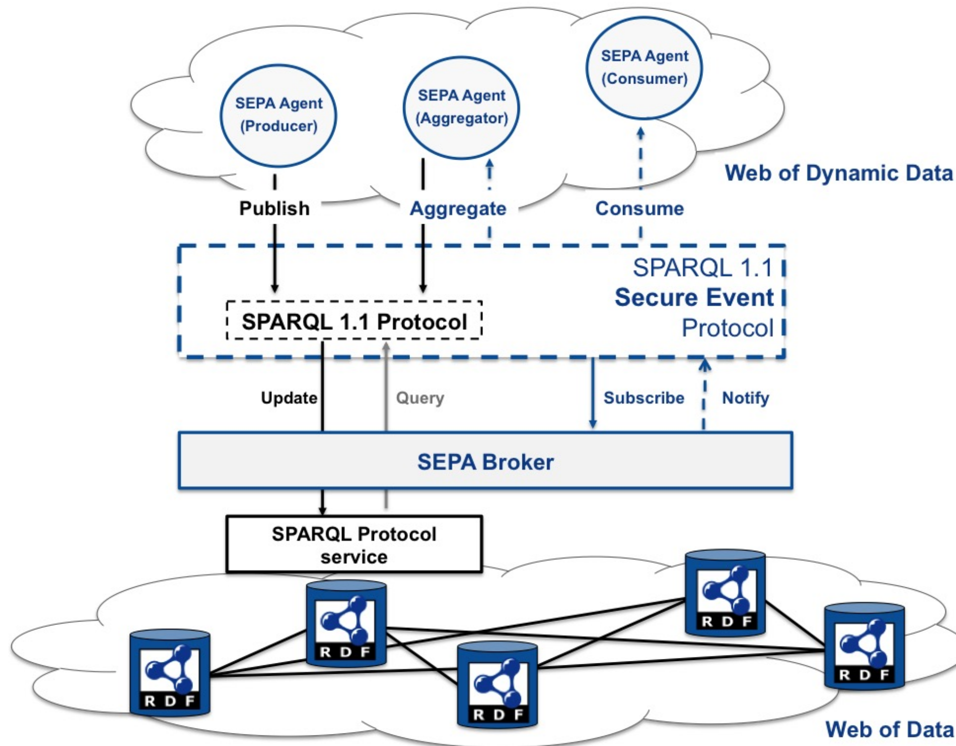


Figure 3.9: From the Web of Data to the Web of Dynamic Data. SEPA, SPARQL Event Processing Architecture.

3.3.2.2 Software architecture

> **Disclaimer**. This section was previously published by MDPI [71]

The modular design of the SEPA broker allows one to support new protocols, mechanisms and algorithms for enabling subscriptions over the Web of Data. A reference implementation of the broker is available on GitHub[6]. As shown

[6]`https://github.com/arces-wot/\acrshort{sepa}`

in Figure 3.10, the broker architecture is layered in three parts: gates, scheduler and core. The following sections provide details on the different parts of the broker.

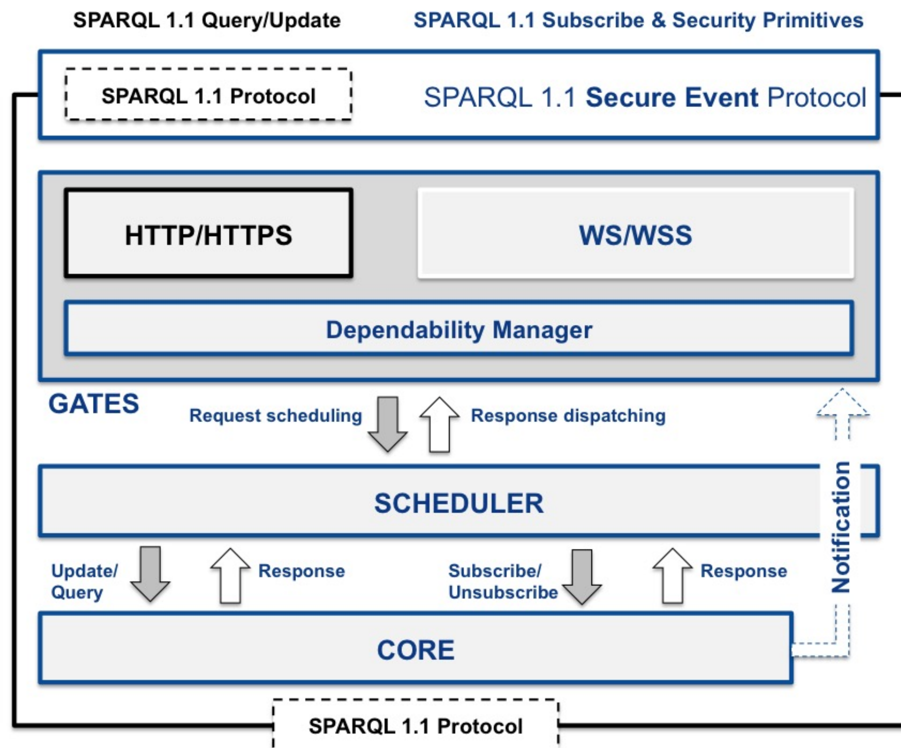


Figure 3.10: Broker reference architecture.

3.3.2.3 Protocols And Dependability

> **Disclaimer**. This section was previously published by MDPI [71]

The gates layer implements the SPARQL 1.1 SE Protocol (see [71]): it create requests (i.e., update, query, subscribe and unsubscribe) for the

scheduler and delivers responses and notifications. As shown in Figure 3.10, the SEPA broker reference implementation provides two gates: HTTP(S) and WS(S), both supporting also the Transport Layer Security (TLS) Protocol. The former processes updates and queries according to the SPARQL 1.1 Protocol, while the latter uses the WebSocket protocol for conveying subscription requests and notifications. The HTTP gate is based on the non-blocking, event-driven I/O model based on Java NIO, provided by the Apache HTTP Components Apache HTTP Components, [7], while the WebSocket gate is based on the Java WebSockets library(Java WebSockets library by TooTallNate, [8]) Other protocols like HTTP, CoAP or Linked Data Notification can be supported by implementing the corresponding gates.

SEPA aims to provide a minimum level of dependability [92] through the Dependability Manager. On the one hand, it implements the security policies and mechanisms presented in [71]. In a real-world scenario, the OAuth 2.0 Authorization Server would be different form the Resource Server (i.e., the SEPA broker). Clients who need secure access to the SEPA broker would register and get a valid token from such an external service (e.g., https://auth0.com). However, at the same time, to provide an off-the-shelf solution for testing SEPA security, the reference implementation of the Dependability Manager implements the client credentials grant type and uses JSON Web Tokens (JWT) (the reference implementation uses the APIs provided by Connect2Id, [9]). On the other hand, reliability is achieved with simple, but effective methods of failure detection in the communication between the broker and subscribed agents. This property is important to grant the general dependability of the connection, but also, it is functional, on the broker side, for cleaning unused resources. In this sense, the WebSocket protocol used by the reference implementation embeds a failure detection mechanism. In fact, thanks to the ping-pong controls frames, the broker and the agents can recognize a failure (i.e., a broken connection) and react accordingly to some fault-tolerant policies. Furthermore, the broker can exploit this information to free unused resources created by agents that

[7] `https://hc.apache.org/`

[8] `https://github.com/TooTallNate/Java-WebSocket`

[9] `https://connect2id.com/products/nimbus-jose-jwt`

have been disconnected in an unexpected way, avoiding a negative impact on the performance. Overall, the protocol and the broker architecture must support the development of distributed applications with some degree of availability and resilience. Therefore, future implementations of new gates should at least implement a basic fault detection mechanism to recognize disconnections or node failures.

3.3.2.4 Requests Scheduling and Responses Dispatching

> **Disclaimer**. This section was previously published by MDPI [71]

The scheduler layer implements the scheduling mechanisms and policies. In the reference implementation, requests are scheduled as FIFO, and the scheduler can be configured with a maximum number of pending requests (i.e., the size of the FIFO queue). The scheduler implements also the dispatching of responses coming from the core layer by forwarding the correct response to the correct gate, which will then send back the response to its client. The same applies to notifications. The requests coming from the gates layer may also be scheduled according to load balancing policies (e.g., processing may performed also on a different machine), and the scheduler may deny requests due to a high number of pending requests (e.g., for quality of service purposes). Requests may also have different priorities, or some requests may be avoided in some application contexts (e.g., the use of time-consuming queries may be avoided in highly synchronized and reactive environments).

3.3.2.5 Processing

> **Disclaimer**. This section was previously published by MDPI [71]

The core of the broker processes the requests coming from the scheduler (i.e., update, query, subscribe and unsubscribe). As shown in Figure 3.11, the main building blocks of the core are: the Query processor, the Update

processor the Subscription Processing Unit (SPU) manager and a main thread holding a FIFO queue of update requests.

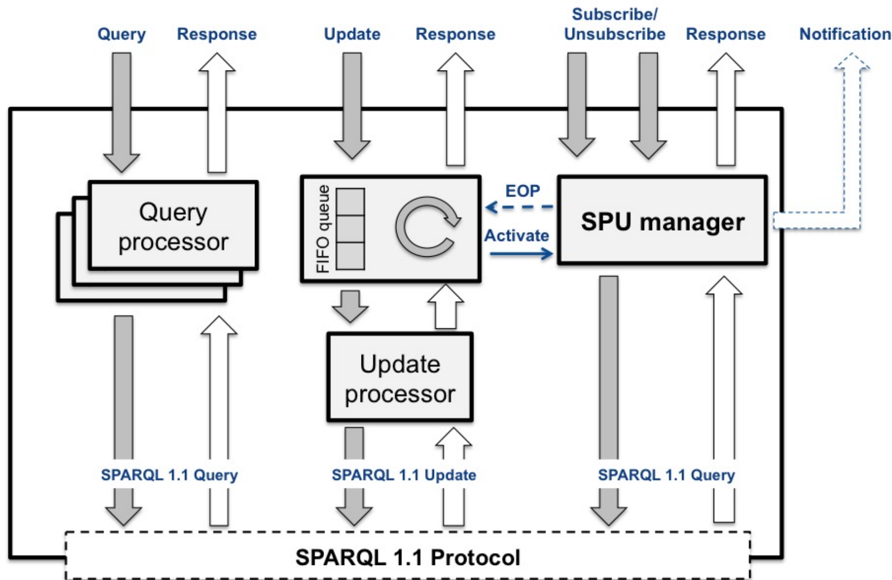


Figure 3.11: Core of the broker architecture. EOP, end-of-processing; SPU, Subscription Processing Unit.

While queries can be processed in parallel (i.e., multiple Query processor instances can run concurrently), updates are sequentially processed through a FIFO queue (i.e., only one instance of the Update processor can be active). As soon as a query arrives, it is sent to the underpinning SPARQL endpoint, and the decision on when to process such a query is made there. As most of the SPARQL endpoints are supposed to be able to process multiple requests in parallel, queuing together queries and updates could result in a substantial decrease of the performance. On the one hand, this means

that the coherence of query processing (with respect to updates) is not granted, but on the other hand, this allows one to take advantage of all the processing power of the underpinning SPARQL endpoint.

Instead, the sequential processing of updates is a fundamental requirement to grant coherence on subscriptions' processing. In fact, as updates change the content of the RDF store, all the active subscriptions must be checked on the same RDF store snapshot. Because of that, a new update can only be processed after the processing of all subscriptions ended. More in detail, the Update processor and SPU manager are synchronized as follows. The core thread sends an update request to the Update processor and waits to receive a response. Once received, the response is forwarded to the publisher (it should be noted that, in this way, the publisher receives a response on the effective status of its update (i.e., the response to the publisher is not provided as soon as the request has been inserted into the FIFO queue, but once the response from the SPARQL endpoint has been received). This allows one to implement, at the application level, synchronization mechanisms that are fundamental for the development of distributed applications. In the case of a successful response, the core thread activates the SPU manager and waits to receive an end-of-processing (EOP) indication. The EOP indicates that all the active subscriptions have been processed. The core thread can so extract from the FIFO queue the next update request (if present) and send it to the Update processor.

With this approach, the update processing will never overlap with the subscription processing. This can be avoided only if the SPU manager does not need to perform queries on the SPARQL endpoint during subscription processing. There are two possibilities to implement this: (i) the SPU manager holds a local RDF store (i.e., cache) for each subscriptions (i.e., this is referred to as the Context Triple Store in [91]); (ii) the SPU manager implements a subscription algorithm that does not require access to the knowledge base, like the one presented in [93] (based on the Rete algorithm [94]). In both cases, the Update processor must return the triples that have been added or removed so that the SPU manager can track the evolution of the RDF store caches or the Rete network. In this scenario, the SPU manager is expected to indicate the EOP as soon as it receives an activate request (i.e., the request can be added to a synchronized queue), and the

core can immediately start processing the next update request.

3.3.2.6 Subscription Processing Unit Manager

> **Disclaimer**. This section was previously published by MDPI [71]

This section describes the internal structure of the Subscription Processing Unit (SPU) manager that processes the subscriptions. The SPU manager architecture is shown in Figure 3.12.

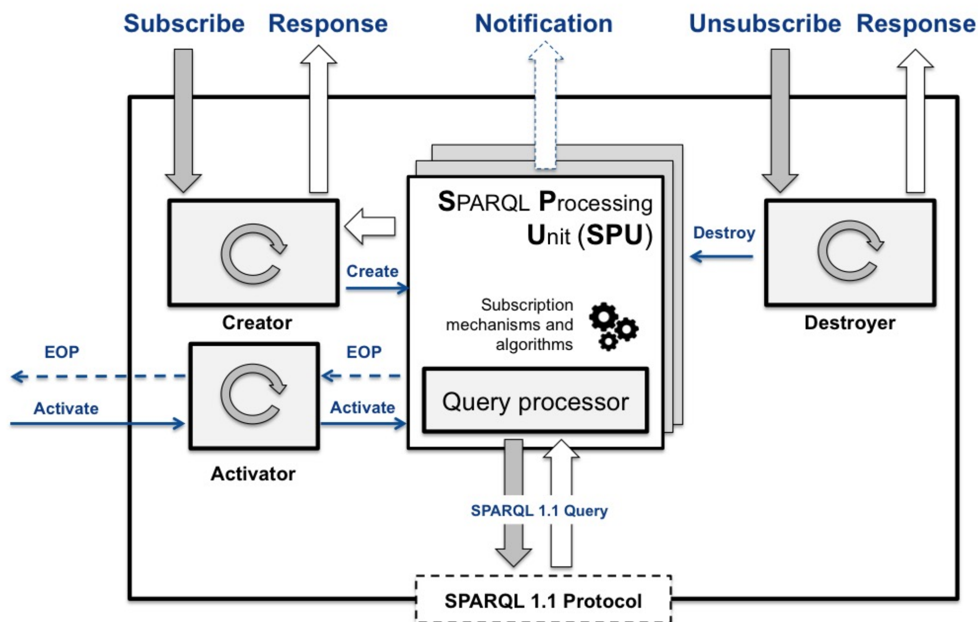


Figure 3.12: SPU manager architecture.

Each subscription is processed by an SPU that is instantiated by the Creator module when a subscribe request is received. The Creator module,

by analyzing the SPARQL query, may instantiate a different kind of SPU (i.e., implementing a different algorithm) or link the subscription with an existing SPU (i.e., the SPARQL query is the same [95]). An SPU is deallocated by the Destroyer module when an unsubscribe request is received. This may be issued directly by a client if a client connection has been lost [71].

On each update received by the broker, the SPU manager is activated (see Activate in Figure 3.12). The Activator module activates all the SPUs and waits for all of them to complete processing. Then, it signals to the main core thread (see Figure 3.11) the effective end-of-processing (EOP) so that the next update request can be processed. SPUs run in parallel, and each SPU may also run on a different machine in a distributed computing environment.

An SPU implements the subscription algorithm and notifies its subscriber (or subscribers if the SPU is shared by multiple clients) of changes due to the last update (if any). As each subscription must be processed at any update, subscription processing shapes the scalability level. Here, multi-resolution approaches where a fast coarse-grained step filters out most unaffected subscriptions leaving the burden of detecting the need for notification to a few candidates, turn out to be particularly effective, as shown by the performance evaluation sections in [80, 91]. In particular, in [91], an algorithm is presented that speeds up the query processing and the results matching by (i) binding variables before sending the query to the SPARQL endpoint and (ii) performing a fast filtering stage based on look-up tables to reduce the amount of subscriptions that are candidates to produce notifications. Another option to optimize the subscription processing could be to implement a Rete network as described in [93]. Discussion on subscriptions processing optimization is out of the scope of this paper, but the reader can refer to [96] for a discussion on how performance can be evaluated.

3.3.2.7 A Dynamic SPARQL Directory

A SEPA instance (i.e., a software implementation of the SPARQL Event Processing Architecture) represents a convenient way to obtain notifications

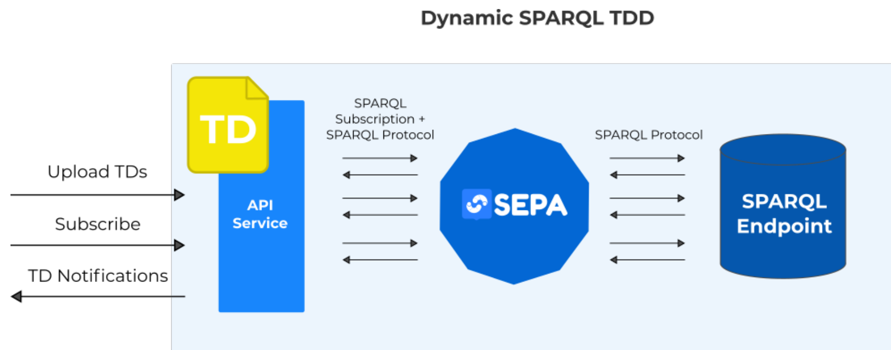


Figure 3.13: A Thing Description Directory powered by a SEPA microservice

about changes in an RDF dataset. On the other hand, considering a WoT deployment, software agents expect also alerts about updates of the TDs present in the system. Even if this information could be extracted from SEPA subscription messages, it is convenient to provide a set of API that manipulates TDs rather than RDF triples.

Figure 3.13 depicts an extension of the SEPA architecture to morph it into a Thing Description Directory. However, it differentiates from other SPARQL based directories because of its ability to capture changes in TD collections, hence its name: Dynamic SPARQL TDD. The architecture adds a new microservice to the standard configuration that is dedicated to transforming TDs in RDF triples and the other way around. As Figure 3.13 shows the TDD API microservice serves as an intermediary between consumers and the SEPA endpoint. In practice, it exposes the same API described in Section 3.3.2 but it enriches the event subscription methods with a SPARQL filter. Before going into detail, it is important to understand the mapping function that transforms valid Thing Description documents into RDF triples. The JSON-LD specification describes a formal transformation to obtain an RDF dataset from a JSON-LD file. Remembering that every TD is serialized with JSON-LD format, we could employ this process to store TD documents inside a SPARQL endpoint or a SEPA instance. Basically, the transformation function consists of expanding JSON values and serializing them as a node map. The expansion uses the context

file as a guide to change simple properties (e.g., label) to full URIs (e.g., `http://www.w3.org/ns/td#label`) so that they can directly be used as well-formed RDF terms. On the other hand, the node map serialization creates a list of nodes that represents a graph, which is easily translated to RDF. Unfortunately, the mapping transformation between JSON-LD and RDF is lossy. During the morphing, some pieces of information about the original document structure are lost; for example, which was the root element of the document or how the URIs were shortened (i.e., the context). The problem is the conversion of the tree document structure to a graph. In a tree the hierarchy is clear, it has a root and a set of children and parents. On the other side, in a graph the relationships between nodes are egalitarian and it has not a definitive access order (Figure 3.14). Consequently, the JSON-LD specification employs a dedicated file type to reconstruct the original document: JSON Frames. A JSON frame is a JSON document that guides the conversion algorithm to shape data in the desired format. It reestablishes the hierarchical structure lost in the conversion to RDF, and it shows an example of the old arrangement. As consequence, it is possible to craft a properly designed JSON frame to reobtain the original TD from the RDF set completing the loop. Hence, the process illustrated allows going from one representation to the other and to store TDs in SPARQL endpoints as RDF data sets. Coming back to the TDD API microservice, it does not only implements the process above but also manages Thind Description Directory data models and internals. In fact, one relevant aspect covered by the TDD API microservice is the meta-model (i.e. how to store TDs in a SPARQL endpoint) and SPARQL results translation. In the solution of the matter, the microservice, when a TD is published, translates it and stores it in an RDF Graph with the same URI of the TD id. As we'll see, this particular data model (shown in Figure 3.16) is preparatory for the other role of TDD API that is the conversion of SPARQL query results into well-formed Thing descriptions. Briefly, SPARQL queries yield specific solution formats that do not resemble a TD; on the contrary, they are designed to covey data in a table format (similarly to rich CSV files[10]). Taking the Listing A.3 as a reference, it is possible to notice that the file

[10] `https://en.wikipedia.org/wiki/Comma-separated_values`

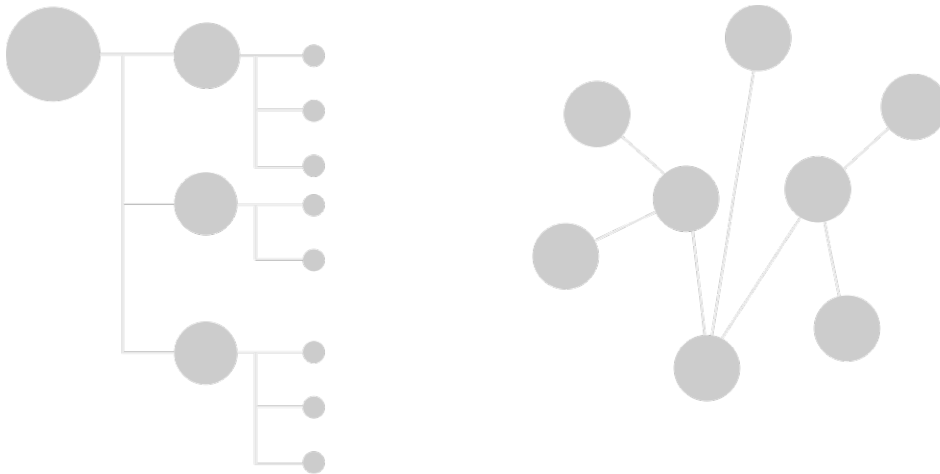


Figure 3.14: On the left a tree like data structure whereas on the right a graph based data model.

is composed by the definition of the table header (i.e., vars property) and the list of the rows of this table (see bindings field). This same format, called SPARQL JSON results, is used as the returned data type for every SPARQL query type (i.e., SELECT, DESCRIBE, CONSTRUCT, ASK). Even if WoT agents might be able to process SPARQL JSON results and extract meaningful information to interact with the described Web Thing, it is more convenient to provide a homogenous API. The advantage is not only on the design side, but also it allows even the least capable nodes to exploit SPARQL queries for discovery. For these reasons, the TDD microservice for SEPA performs the following steps on SPARQL search:

1. The query must be a DESCRIBE SPARQL query
2. The object of the DESCRIBE must point to an RDF resource that is rdf:Type of Thing
3. The Describe query is transformed into a CONSTRUCT query as shown in Figure 3.15
4. The new query will have:

   (a) The same where clauses of the original
   (b) Additional clauses to retrieve TDs triples from the container graph

5. The new query is issued to the SPARQL endpoint

6. The resulting triple set is grouped in a list of TD in RDF format

7. Each element of the list is processed and framed to be a valid TD document

8. Each element of the list is processed and framed to be a valid TD document

Starting from the beginning, the algorithm verifies that the query is a particularly formed SPARQL query type (see code A.4). Other SPARQL query types do not fit in the purpose of filtering TD documents. For example, SELECT queries work on RDF term granularity so that users can select a particular list of objects, subjects, or predicates. On the contrary, DESCRIBE specifically ask for a full description of the selected subject (in our case a Thing Description). Unfortunately, DESCRIBE does not return the full graph connected with a particular subject; instead, it stops at the first direct connection. Consequently, the query is transformed into a CONSTRUCT that can return the content of an RDF Graph. If a TD was scattered across multiple graphs or if one graph contained more than one document, this query transformation would have been much more challenging. That is why we chose the model shown in Figure 3.16.

Thanks to SEPA microservice, we do not have only queries at our disposal but also subscriptions. In this case, the interface microservice receives results as added and removed Quads (i.e., triples grouped in graphs). Similarly to queries, the TDD agent acts as an intermediary to translate user requests and responses. To understand the process is convenient to think about the possible update operations that users can perform: add, remove, and update. When adding or removing a TD, the TDD microservice adds or deletes an entire graph, hence the conversion process can detect an added event by simply test if the removed triple set is empty and vice versa. On

the other side, when updating, the notification contains both added and removed triples. Here, we use a dedicated internal graph to store the id of the updated TD, the timestamp, and the information about the updated paths. The TDD inserts this information with the same update operation that modifies a TD. Then it receives the updated triples together with the metadata stored previously. Finally, it exploits this data to frame the updated triples and to send the resulting notification to the client. To obtain this information, the query processing algorithm is slightly different from the search operation. Specifically, from one single DESCRIBE it create two CONSTRUCT queries. The former is the same query described in the algorithm above and it detects added and remove TD events. The other queries the dedicated graph to be informed about update events.
In conclusion, the Dynamic SPARQL Directory consists of a multi-agent solution that allows WoT users to obtain notifications about SPARQL selected TDs. Even if it acts also as a SPARQL endpoint it features a dedicated API which to fulfill WoT consumer's needs. It can be used as an advance discovery service for both constraint devices and high-end nodes.

3.4 Implementation

The abstract software components detailed so far were implemented and evaluated in two different use cases. The first one is a structural health monitoring platform called MODRON [33]. It was developed together with INAL inside the Mac4Pro[11] project. The latter is a smart agriculture framework called the Smart Water Management Platform (SWAMP)[12] designed within the homonymous Horizon 2020 European Union project. The two solutions were developed independently during the three years of the Ph.D. course with the efforts of my research group and other institutions. In both cases, the strong requirements on interoperability lead to the adoption of semantic-based technologies. However, only Mac4Pro applied the WoT paradigm as the main enabling touchingly. The decision-makers of the

[11]https://site.unibo.it/mac4pro/it
[12]http://swamp-project.org/

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX td: <https://www.w3.org/2019/wot/td#>
DESCRIBE ?td {
  ?td rdf:type td:Thing .
  ....
}
```

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX td: <https://www.w3.org/2019/wot/td#>
CONSTRUCT {
  ?x ?y ?z
}WHERE {
  GRAPH ?td {
        ?x ?y ?z
  }
  ?td rdf:type td:Thing.
  ...
}
```

Figure 3.15: Query translation in a Dynamic TDD

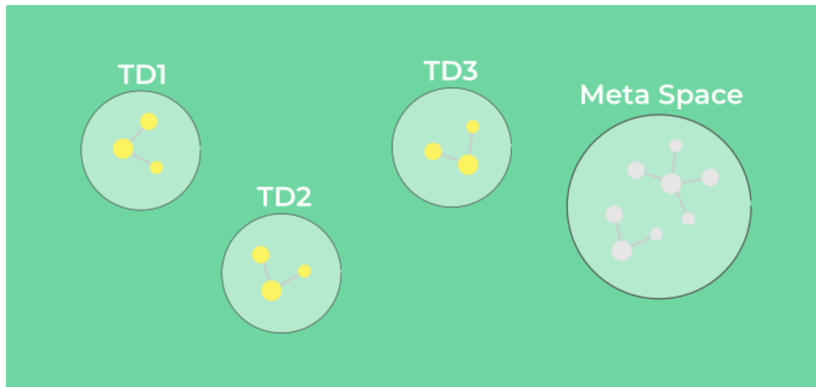


Figure 3.16: A simplified view of how the Dynamic TDD store Thing Descriptions and other directory relevant metadata (i.e.,modification records and ownership)

SWAMP opted for FIWARE[13] a European Union founded IoT platform. In this work, we will describe an alternative solution for SWAMP completely base on the WoT specifications.

The following will present the two projects and how WoT technologies were employed to achieve the monitoring functionalities required. Finally, it will focus on how the implemented software agents fit inside the general architecture described in 3, with an eye on the applied data model and vocabularies.

3.4.1 Structural Health Monitoring platform

> **Disclaimer**. The following is an adapted content of what is accepted to be published in IEEE Consumer Communications & Networking Conference [33]

The Structural Health Monitoring (SHM) defines a method for evaluating the state of aging constructions by testing the status of their present integrity. Additionally, SHM data is useful for building prognostic schemes to predict

[13] `https://www.fiware.org/`

the remaining life of structures/infrastructures [97]. SHM systems will play a crucial role in modern smart cities thanks to their ability to reduce the vulnerability of strategic structures, increase the protection of architectural heritage, and to some degree, to save human lives [42].

Several research studies have recently shown the possibility of enhancing the performance and reliability of SHM systems by employing IoT inspired sensor data management and analytics technologies [34]. In fact, even in the presence of high sampling rates and long-term measurements, big-data techniques have been shown to help distributed storage and real-time processing of SHM deployments [98]. Moreover, Machine Learning (ML) and Artificial Intelligence (AI) techniques offer valuable methods for condition assessment and/or structural damage detection, considering the enormous number of data sets collected. For instance, in the presence of image-based inspection, such ML/AI-driven approaches have proven to be especially successful [36]. Thus a modern SHM framework must jointly optimize all the various software architectural layers to ensure real-time and over-time functionalities. Although several ad hoc software implementations have been published in the SHM systems literature (e.g. [42] [41] [43]), only a few propose complete and scalable IoT platforms for collecting and analyzing sensor-to-cloud data [40]. The software solution presented in this subsection addresses two main requirements of IoT based SHM deployments, namely the system scalability and interoperability.

The first problem not only relates to the need for massive data volumes to be handled but also implies the optimum balance of computing between the network resources available in a cloud-to-edge continuum. The need for interoperability, on the other hand, is also dictated by the heterogeneity of the sensor devices that can be mounted on the structure [44], a solution that is generally favored to increase the robustness of the SHM systems effectively [99]. Besides, sensor devices may belong to different manufacturers, speak different network protocols (e.g. HTTP, MODBUS, etc), or even provide different sensing features (e.g, accelerometers, hygrometers, strain–gauges). As introduced in Chapter 2, such fragmentation is one of the primary reasons for high computational and installation costs, which may unavoidably hamper the full–scale applicability of present IoT implementations. Issues that the WoT paradigm aims to address; consequently,

we propose MODRON platform, a WoT based software framework for sensor–to–cloud data acquisition and management in SHM scenarios [33]. Specifically, the platform addresses the acquisition of sensor data from the monitoring layer, the sensors installation logistics, the distributed data storage, visualization of data, and analytics. In order to accommodate heterogeneous sensor environments, MODRON relies heavily on the WoT W3C standard: this is done following the layered architecture presented in [33], with an edge layer composed of WTs exposing the sensor data, and a remote cloud layer consuming and accessing the WTs services. Moreover, the platform is implemented with extensibility in mind. Thanks to the modular design, it allows the installation of new classes of sensor devices by registering their TD descriptions with both human and machine interfaces. Finally, another characteristic of MODRON solution is its high adaptability since the cloud services can update their behaviors (e.g., UI) accordingly to the set of active WTs and hence, it abstracts from the sensing technologies in use. The remainder of this section will present the implemented software solution in relation to the concepts presented in [33], together with a vocabulary proposal for SHM data.

3.4.1.1 Architecture

Figure 3.17 illustrates the MODRON layered implementation. It is possible to notice that the MODRON architecture is more deployment bounded than the generic monitoring architecture. Every layer in MODRON is collocated at a specific IoT sensor-to-cloud chain level, whereas the abstract solution would not confine one single layer to a specific node type. With the reference to the same figure, starting from the bottom, we have the monitoring layer. In this solution, the monitoring layer corresponds with the hardware components of the platform (i.e., devices and sensors) and their vendor dependant protocols. It is the main role is similar to the Sensing layer, it acquires real-time sensing data and provides an interface to extract it. At the time no WoT technologies were thought to be deployable in such constraint devices. Therefore, at this level sensor would expose non WoT complaint interfaces limiting sensor-to-sensor interoperability. However, as [REF Victor system] demonstrated, it is possible to design a constraint

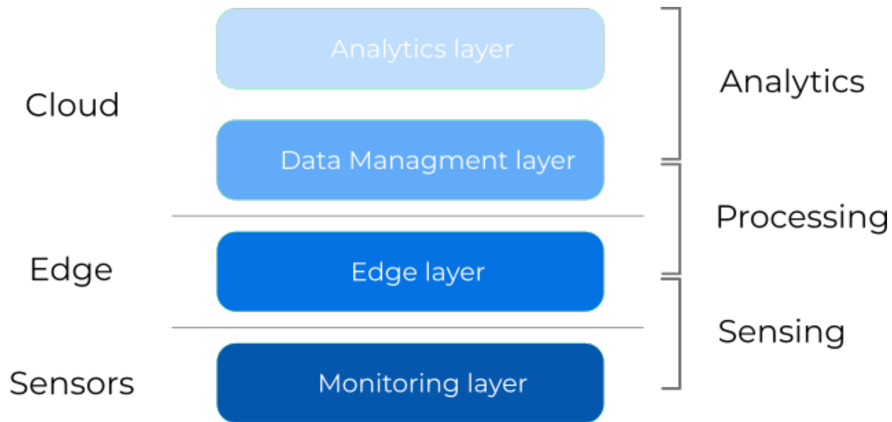


Figure 3.17: Summary of the MODRON architecture. The layers prosed in Chapter 3 are on the right.

system with embedded TD descriptions and WoT protocols. Consequently, we could incorporate this monitoring layer in the more generic sensing layer proposed above.

Next, the Edge layer 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, HTTP, etc); (ii) making sensor values and devices' status information available to the data management layer, and, at the same time, hiding the heterogeneity of acquisition protocols/hardware. By resorting to a W3C WoT approach, the above requirements were tackle, thus creating a set of W3C WTs on the edge nodes. More precisely, MODRON edge WTs divide into two types: Sensor WTs and Digital Twin WTs. The first category defines WTs that corresponds exactly to one sensor unit (e.g. an accelerometer) and it allows remote interaction with the device based on WoT protocols. The latter, instead, models a virtual entity derived from the aggregation of multiple Sensor WTs. For instance, a group of sensors installed on the same structure might be represented as a virtual group for convenience or organization purposes. The Modron edge layer can be mapped in two of the generic layers defined above because of its intrinsic dualism. First Sensors WTs represent basic sensing layer agents, thus they are mapped at the lowest layer of the generic architecture. Second, the

concept of Digital Twin WTs in MODRON closely relates to Virtual Sensor WTs. Consequently, this type of agent can be thought to belong to the Processing layer.

Finally, the Data Management and Analytics Layers. In MODRON these two layers contain a set of business logic and infrastructural services, like a persistence dedicated agent and a Thing Description Directory. Business logic modules consume data observed from the layers below, to provide application-specific information to the end-users. For instance, the Persistence mash up application (i.e., a WoT based application) is in charge of gathering data from the sensing layer; to this purpose, they consume the TDs registered in the TDD and query them at fixed intervals (for request-response interactions) or register to the events produced by the remote WTs (for publish-subscribe interactions). The monitoring data are then stored in a distributed database, entailing data consistency, and replication capabilities. Remembering the definition of the last layer in the reference generic architecture, services like the persistence application can be mapped to the analytical layer. Other high-level services examples in the MODRON platform are:

- Visualizer. The Visualizer 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

- Analytics app. Analytics process the stored data and provide the signal processing techniques for structural integrity evaluation.

- Data Aggregator, allowing to select, merge or extract features from the stored time-series

- Data Plotter supporting the visualization of the output of the Data Aggregator, and/or their exportation on files

In summary, every MODRON data management and analytics service falls under the analytical layer of the monitoring architecture.

3.4.1.2 Implementation insights

The first implementation detail of interest is how MODRON handles the control access process for the WTs in the edge layer (i.e., sensing layer). 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). Consequently, each sensor device is described by three different TDs that represent three different access roles. The first type defines read-only access to observed data and it describes the so-called Observable Sensor WT. An Observable Sensor WT exposes only properties and actions allowing to read sensors' measurements but not to modify the sensor configuration. On the other hand, Controllable Sensor WT (described with another TD type) also supports the remote device configuration, including the possibility to upload at run-time a new Behaviour of the WT. This TD is visible only to agents that have administration-level permission since it enables to change the behavior of the system. Finally, Debug WT includes basic diagnostic and self-testing functionality that may be useful during the installation and calibration of the SHM system.). Then, effective access control policies can be devised through the mapping of the user profiles with the instances of the Sensor WTs. Moreover, each TD links the other one so that consumers can navigate the access chain straightforwardly. For example, if a consumer has access to an Observable Sensor WT it can follow the link with relational type "modron:control" providing the correct credentials to gain control of the same sensor instance.

The supporting WoT runtime for MODRON WT is node-wot, the official implementation of the Scripting API specification. This runtime was installed and deployed in the different architectural layers. At the edge, it exposes the Sensor WTs described above for accelerometers and piezoelectric sensors. Runtime-to-sensor communication happens over the custom made Systems APIs that translate and forward WoT messages on a USB serial port. Table I reports the main Affordances of a MEMS accelerometer device. At the cloud, MODRON uses a wide number of JS libraries/tools and database systems. The back– end functionalities are implemented through Node.js and related libraries, including, among

the others: LoopBack, SocketIO, and Nest.js. The Visualizer exposes APIs for WT registering, searching, and filtering based on GraphQL, an open-source data query and manipulation language. Also, the framework includes a combination of database technologies, such as (i) Blazegraph, a triplestore used to save the application metadata and the TD; (ii) Apache Cassandra , a NOSQL database used to store the sensing data gathered by the Persistence MA; (ii) Redis, an in-memory data structure store used as 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 application 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), and then periodically transferred to the Cassandra database management system. The latter has been configured for distributed operations: the cluster is composed of three instances and employs a distributed data balancer and a basic replication strategy with a factor equal to the number of available instances.

3.4.1.3 Vocabulary

As outlined in Section 3.3 another important aspect of the monitoring platform is the data model for the exchanged monitoring information. With the reference to the layered structure for platform vocabularies, MODRON utilizes state of art ontologies depicted in Figure 3.18. Starting from the Monitoring layer, we found Semantic Sensor Network[14], MODRON Access, and Schema[15] vocabularies. The SSN ontology contains concepts like Sensor, System, Deployment, and their relation with measured physical properties (e.g., ObservedProperty RDF property). In MODRON SSN describes the types of WTs to know if a particular instance is a Sensor or another entity (i.e., a platform where the sensor is installed). Moreover, it explains how differently related to each other; for instance how multiple sensors belong to the same subsystem. The location of these entities is described thanks to the concepts defined in Schemas like place and geoCordinates. Finally,

[14] `https://www.w3.org/TR/vocab-ssn/`
[15] `https://schema.org/`

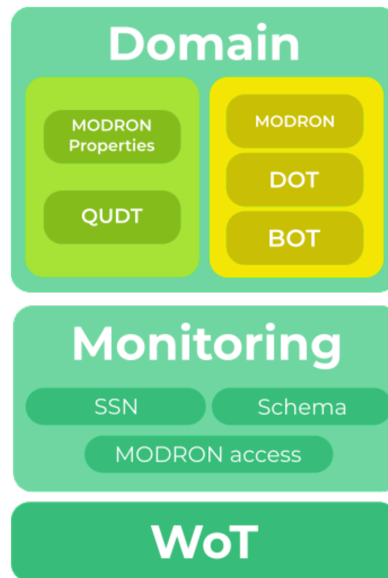


Figure 3.18: Layered knowledge model of the MODRON platform.

MODRON Access vocabulary defines the two relational types to express the association between an Observable, Control, Debug Sensor: controls and debug. On the domain level MODRON employs:

- BOT[16]. BOT serves as common ground to express part of a complex construction or a building. In MODRON structures are subclasses of BOT elements.
- DOT[17]. DOT augments the BOT ontologies with terms to express damages to elements and how they evolve during time. It also contains a proposed taxonomy to categorize the different damages and their causes
- MODRON. DOT and BOT do not define concrete structure types or structural elements. Therefore, MODRON vocabulary has the goal

[16] https://w3c-lbd-cg.github.io/bot/
[17] https://alhakam.github.io/dot/

to indicate all the construction types and industrial components of structural health monitoring interest. It is still in its early stages but it will be extended in future interactions. One example of one term defined at this level is a concrete bridge.

- QUDT[18]. it is a well-known vocabulary that contains a set of scientific quantities and units of measurements. In the platform of the matter, it is employed to describe the subject of a measurement and its unit. For example, qudt provides Temperature as a quantity and Celsius and Kelvin for units.

- MODRON properties. This vocabulary defines other physical properties specific to SHM use cases like Acoustic Emission and Dynamic Vibrational Response.

In conclusion, the MODRON data model follows the same structure defined in the abstract monitoring architecture. In the appendix, readers can find an example of this data model applied to a TD and a structure described as a JSON-LD document.

3.4.2 Smart Agriculture

The agricultural sector is undergoing a transformation driven by new technologies, which seems very exciting, as this primary sector will be able to shift to the next stage of agricultural productivity and profitability. We are experiencing the third wave of revolution in the farming and agriculture sectors that consists of optimizing the crop inputs (i.e., water, soil, and Sun) to obtain the best crop yield possible. This process takes the of Precision Agriculture and since heavily influenced by the usage of digital technology is widely know as Smart Agriculture (the word smart is used also in other sectors like Smart Buildings, Smart Home, Smart Factory, etc., to entail the employment of silicon-based devices and information technology). As agriculture is the main consumer of fresh water in the world, amounting to up to 70% of the total use [100] reducing crop water needs

[18] `http://www.qudt.org/`

has not only economic impacts but also environmental benefits. Farmers spray more water than required (over-irrigation) in an effort to prevent loss of productivity caused by water stress (under-irrigation), and as a result, not only productivity is threatened, but also water and resources are wasted. In contrast, in smart agriculture precision irrigation methods can use water more effectively and efficiently, eliminating both under-irrigation and over-irrigation problems. IoT based solutions are an obvious option for smart water management applications, but the integration of various technologies necessary to make it function smoothly in practice is still a challenge. The Smart Water Management Platform project had the goal to develop a high-precision smart irrigation system concept for agriculture. The key concept is to make it possible to optimize irrigation, water delivery, and usage on the basis of a holistic study that gathers data from all aspects of the system, including the natural water cycle and accumulated knowledge of specific plant growth. This results in savings for both parties, as all leakages and losses are detected and water availability is best guaranteed in circumstances where water supply is limited. The SWAMP solution was developed with a hands-on approach based on four pilots in Brazil, Italy, and Spain[19]. In order to adapt to various SWAMP pilots, the platform can be specifically configured and implemented, tailored to meet the requirements and limitations of different settings, countries, climate, soils, and crops.

The official SWAMP platform contained the core modules of FIWARE and semantic features provided by a SPARQL-based context engine [71]. It may be installed in a range of different configurations for component placement in the cloud or in the fog, IoT protocols, and smart algorithms and analytics (cloud or fog based). This is aimed to have the necessary flexibility to experiment with the different pilot deployment scenarios of the SWAMP Platform and providing additional features in terms of the replicability and adaptability of its components to different settings. As scalability is a major concern for IoT applications, a performance review of FIWARE key software components was carried out, tailored for each pilot scenario. The early results showed that the platform in the subject met the pilots' requirements but at the price of scalability. FIWARE components

[19] `http://swamp-project.org/post-422/`

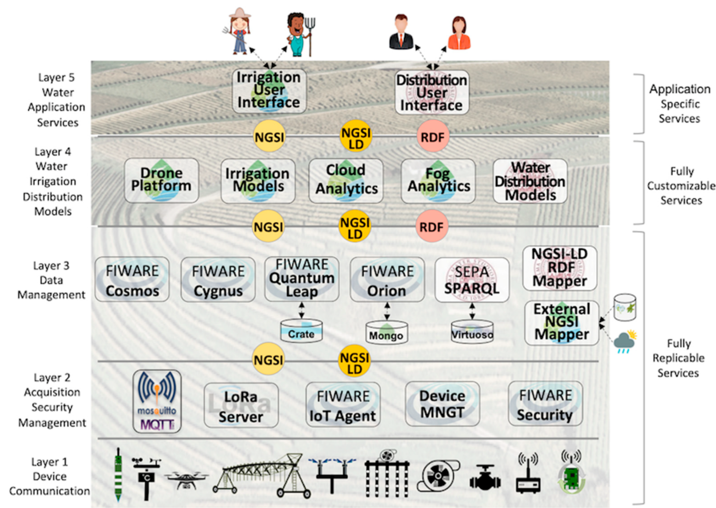


Figure 3.19: The orginal SWAMP platform as proposed in [61]

had to be fine-tuned to provide improved performance and others had to be completely re-engineered to provide higher scalability using less computational resources. Also, MongoDB was identified as the bottleneck of the FIWARE tested installation that may cause system crashes. For this reason this document will propose an alternative archiectural design with the goal to increse scalability and replicability of the system.

The rest of this section will follow the same structure of Section 3.4.1, starting from the description of the software architecture in relation to the monitoring WoT platform. Finally, it will describe the designed ontology specifically made to address open issues in Smart Agriculture vocabularies.

3.4.2.1 Architecture

The original SWAMP platform is described in Figure 3.19. It is possible to notice that also SWAMP has a layered architecture, but it is heavily centralized due to the FIWARE interaction model (it has a star topology with an information broker as the center). Figure 3.20 proposes the same functionalities but harmonizing them with the WoT monitoring platform presented in this work. At the bottom layer, there are several WTs in

charge of mapping Smart Agriculture specif IoT protocols. Notice that since it is common to employ LoRaWAN technology in those contexts, these WTs may also be installed as cloud or fog components (the hierarchy does not reflect the location inside the IoT stack). In fact, LoRaWAN employs a sensor-to-cloud architecture with encrypted communication. Intermediaries cannot interact with LoRaWAN sensors without passing by the central node that is in charge of authentication data consumers and configurators.

On the processing layer, multiple Virtual sensors aggregate and store short-term historical data acquired from sensing WTs emulating the Quantum leap role in the original solution. The different Virtual WTs works together to provide a set of distributed data analysis and storage services so that the upper business logic applications do not rely on a single point of failure. Those applications are classified, indeed, as analytics actors in the WoT monitoring platform. Some of them emulate entire physical systems like Water Distribution model application (i.e., a Virtual System) others analyze data to convey more complex data to platform clients like the Analytics service.

As in the SHM use case, knowledge is stored in a human machine-understandable format on a SEPA based Thing Description Directory. It is notable how in the proposed architecture clients use SEPA as the main source of contextual information but depend on the single WTs and WoT agents for monitored data. Thus the interaction has not a single point of failure but it is distributed on the different subsystem that composes the software solution, enhancing the scalability of the system.

To conclude, SWAMP architecture can be revisited to be conformant to the WoT paradigm. The feature set is maintained but the platform scalability is improved.

3.4.2.2 Implementation insights

In open-field agriculture, the IoT solutions leverage on different radio protocols and devices. Usually, radio protocols should cover long distances (even kilometers) and be energy efficient. Devices too need to be energy saving as they are deployed for months and sometimes even years in harsh environments. A sleeping-cycle is one mechanism they use to save energy

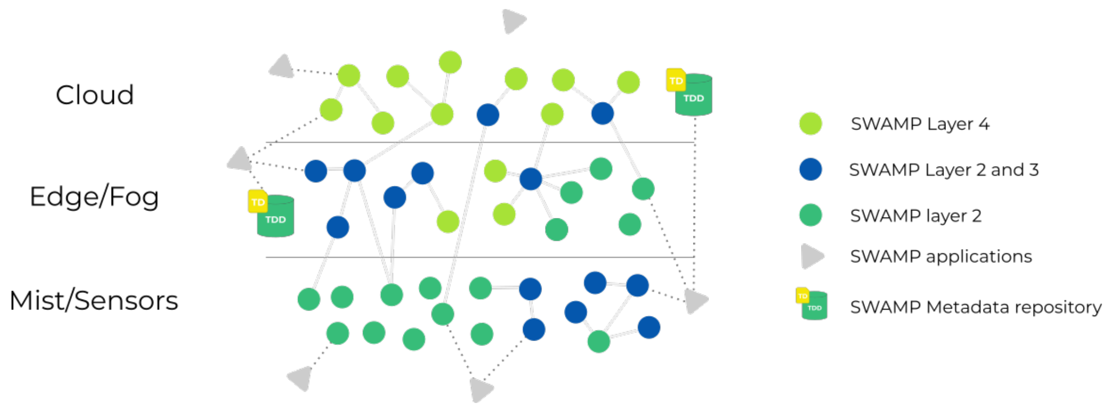


Figure 3.20: The SWAMP architecture revised with the Web of Things technologies. SWAMP layers 2,3 and 4 are respectively mapped into the Sensing, Processing, and Analytical layers.

usually coordinated by loggers/gateways or preprogrammed. Loggers are deployed closed to sensor devices and have more storage space. They serve as buffers between sensors and higher services. Often loggers and sensors are embedded in the same board, otherwise, they are connected using cables or close-ranged radio protocols. On the other hand, gateways serve as a collection point for data of an entire field or farm. They are much more capable devices and usually are more energy-consuming. In some deployment scenarios, they host a full operating system with multiple software facilities installed. Otherwise, gateways only serve as relays of data sent from the loggers and sensors to cloud services and vice-versa. The cloud services may be partially hosted in edge servers to preserve data privacy and responsiveness of the whole IoT solution.

Sensor data plays a central role in Smart Agriculture. In particular, it is critical that the information sensed is associated with a timestamp. Common algorithms use time series to calculate the water needs of a crop. Furthermore, soil sensors usually are calibrated over a specific soil type (which may differ even in the same geographic region). For example, the calibration data for a soil moisture sensor is represented by a function that maps sensor output to soil water content. In literature, this function is

knowns as a calibration curve . Commercial sensors are precalibrated with a "standard" curve but on most occasions, it fails to accurately measure the water content. Therefore, it can be configured during the installation phase (which may happen every time the soil is plowed). Finally, a crucial aspect is forecasting. Farmers use this information to actively change their management procedures. Services exploit it to suggest irrigation schedule or change device settings to behave accordingly to environmental changes.

3.4.2.3 Vocabulary

Since the state of the art data models were found insufficient to describe SWAMP entities and concepts, the project required from-zero engineering of the main vocabularies. Taking again the vocabulary layered architecture (Figure 3.3), right above the WoT level we find SSN together with SWAMP Physical IoT. Physical IoT vocabulary categorizes all the possible agriculture specific sensors and actuators that are not usually employed in other fields. An example is a soil moisture sensor which has known to be employed as a reference measuring device about crop nutriments. Moreover, this vocabulary defines a relation to express that another entity has a set of devices installed. It is semantically similar to ssn hosts relation, although it might identify a loose logical association between that particular object (e.g., the device is installed somewhere near the object but it monitors its status).

On the domain-specific level, SWAMP ontology defines three vocabularies: Agriculture, Water Management, and Crop Needs and Recommendations. The first set of terms specifies concepts related to agriculture techniques and entities. It is aligned with the most known agriculture specific ontology, Agrovoc, but it adds ideas from a management point of view. For example, it introduces the concept of Management zone, which is the smallest region of a field with unique soil characteristics or it describes how fields can contain one or more crop types. Then the Water Management vocabulary describes watering systems and watering parts like sprinklers, pipes, canals, water reservoirs, etc. Finally, there is the Crop Needs and Recommendations; this vocabulary serves to define shared application knowledge about irrigation scheduling and crop nutrition needs.

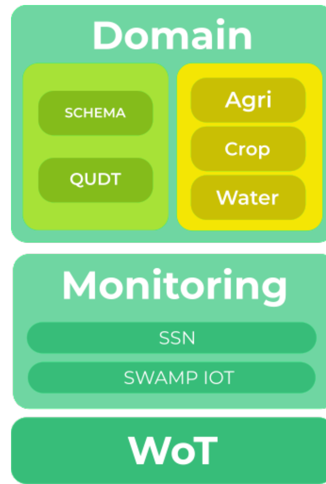


Figure 3.21: SWAMP vocabulary structure

For instance, the term water need indicates the amount of water that a plant requires in a specific moment. This information can be exploited to create irrigation plants; a concept described in the same vocabulary.

Additionally, SWAMP vocabulary structure comprehend also:

- QUDT. It describes measurements and units
- Schema. Schema at the domain level defines user management concepts like Farmer Name, Surname, or address

3.4.3 Tools

During these years of the development with WoT technologies, the author of this thesis dedicated some of his time to contribute to the WoT software ecosystem. In particular, he focused on dissemination and easy to use aspect of the WoT standard. Therefore he developed three Typescript based tools to help newcomers in understanding this new technology stack. The first one is called WoT Application Manager (WAM). WAM is a simple Command Line Interface to guide new users in the creation of a WoT

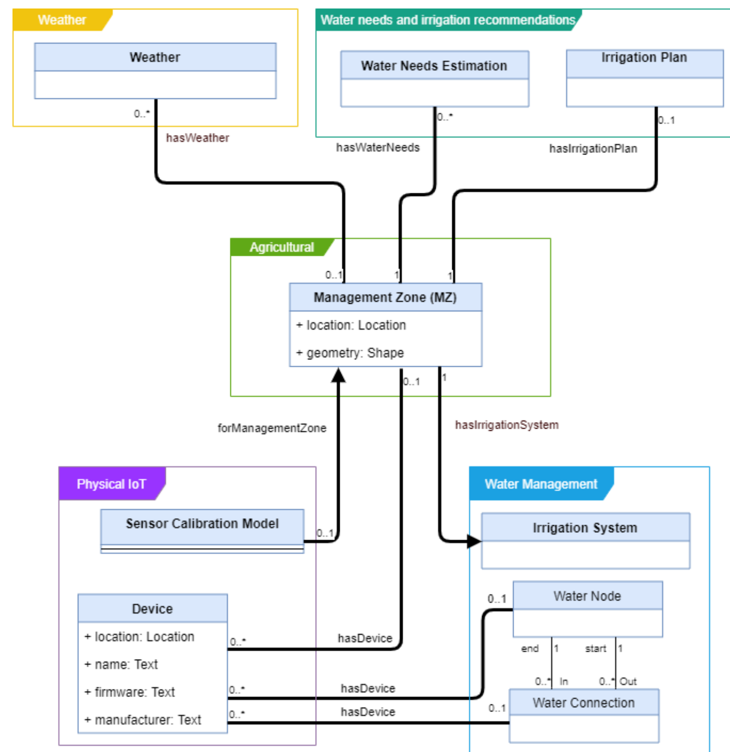


Figure 3.22: SWAMP original ontology summarized data model

```
$ wam init

? language: (Use arrow keys)
  TypeScript
> JavaScript
```

Figure 3.23: Initialization of a WoT application using WAM

application in JavaScript or TypeSript. As shown in Figure 3.23, it provides a guided setup process that installs and compile a basic project, ready to be developed. Moreover, WAM helps in the deployment thanks to its bundling capabilities. From a project with multiple modules and files, it creates on single minified JS file that can later be deployed in any node-wot enable IoT node. Alongside WAM TD-Code is a helper visual studio code extension that eases the design of Thing Description files. The extension provides a set of auto-filling templates that are useful when manually editing Web Things descriptors in a complex WoT application. In addition, it also provides a validation process that informs the user if his/her TDs are compliant with the latest W3C specifications. Finally, the third tool is a didactic device created to allow students to have hands-on sessions on WoT technologies without acquiring any particular hardware. It is a simple simulation of a Smart Farm environment inspired by the SWAMP platform: WoT Farm. It consists of two modules: the backend and the web frontend. The backend runs on a remote server hosted in the university department and it is in charge of exposing simulated Web Things, the Web frontend and implement the simulation logic. Currently, the simulation creates a virtual soil that can be irrigated with virtual sprinkles (exposed as WTs). Then the water content can be sensed through a set of virtual soil moisture sensors. In the future, the backend will simulate more complex scenarios

Figure 3.24: WoT Farm main screen. On the left there is a code editor with a sample WoT script (see Listing A.2). On the right a 3D view of the simulated farm.

like a full water cycle and other sensor devices (e.g., wind speed sensor, weather stations, crop growth sensor, etc.). On the other hand, the Web frontend renders the current simulation station (see Figure 3.24) and a text area where users can test WoT scripts and APIs. The idea is to ask students to implement gradually complex application logic so that they can evaluate their understanding of the WoT architecture and inner workings. For example, irrigate crop one when the water content sensed by sensor http://wot.farm.com/1234 is below 20%. Currently, the set of possible applications is limited by the set of the simulated physical properties and sensor, but, as indicated above, the platform will have more sensors in the near future. Finally, another future enhancement will cover the possibility to have multiple users and simulation sharing trough custom created links.

Chapter 4

Evaluation and Discussion

This last chapter covers the evaluation of the open monitoring platform through a scenario-based analysis method. This method was chosen among the others for its ability to assess qualitative properties like extensibility and interoperability that are the core non-functional requirements of the software product. Moreover, it will summarize the early results obtained when testing the Web Thing migration feature. Finally, it will discuss possible challenges in the WoT standard that were encountered during the development of the MODRON and SWAMP WoT based platforms.

4.1 Software Architecture Analysis

The Software Architecture Analysis Method (SAAM) [101] was born in the early 90s to improve the software development lifecycle and assess the quality of a descriptive architecture. The method specifies five simple steps:

1. Requirement definition
2. Architecture Description
3. Scenario development
4. Scenario prioritization

5. Evaluate the architecture with the respect to scenarios
6. Overall evaluation

The peculiarity of this method is its ability to give a good estimation of architectural qualities even at an early development stage. Since the open monitoring platform, architecture is still in its infancy and is a high-level design, SAAM was found suitable for the assessment of its characteristics. We already saw how the architecture is generic enough to be able to describe two diametrical use cases coming from heterogeneous IoT contexts. This section will focus on three other fundamental features that we want the platform to have: Interoperability, Extensibility, and Openness. After describing the requirements and the architecture (See Chapter 3), the SAAM method foresees the definition of at least one relevant scenario for each evaluated feature. The scenarios serve as a concretization of the abstract quality and a future assessment tool. Consequently, in the following, each subsection relates to a specific evaluated feature, and it contains the scenario description and the scenario discussion. The different scenarios are considered equally important, and they are listed in random order. Later an overall comment is drawn, and a final discussion on the scalable properties of the platform is reported.

4.1.1 Interoperability

In a broader sense, interoperability is the ability of a computer system or software to exchange and use information from other platforms. Assessing this feature is challenging due to the countless software/hardware solutions which exist on the market. Consequently, we define three scenarios describing how the architecture would react with previously unknown systems.

In the first scenario, a new device type needs to be introduced in an online platform deployment. In this context, the architecture naturally handles the new addition, and it integrates the devices with no substantial structural changes. Precisely, if the device already implements the WoT stack and it is associated with its TD, the platform assimilates it as any other Sensing Layer agent. On the other hand, if the device does not have

a WoT runtime, minor modifications should be performed in the platform deployment such that the device is correctly recognized as part of the sensing layer. More in detail, the device might either support a WoT complaint protocol (i.e., HTTP, HTTP, etc.) or use a proprietary communication language. In the former case, developers or maintainers need to design and deploy a Thing Description document capable of describing the new device type. In the latter, WTs in the processing layers take the burden of proxy the proprietary protocol and expose the new device as a WoT Virtual Sensor. Of course, developers might even define general-purpose software in charge of deploying such proxy services on demand, decreasing the overall integration effort.

In the second scenario, stakeholders want to employ a new data model for a previously unknown physical property. The newly added model might overlap with earlier context definitions and in this case, ontology alignment countermeasures should be taken to continuously assess backward compatibility with other platform services. Future developments of the architecture could define dedicated agents that process and translate metadata information to one representation and vice versa. Currently, Virtual Sensors might perform this transformation exposing both translated and not translated metadata in their TDs. On the other side, if the new data model is completely orthogonal with the old vocabulary it can be published with no further modification. Although, old services might need to be extended or updated to exploit this new information.

In the third and last scenario, consider that a new device publishes sensor data encoded in a new data format, how will the monitoring platform cope with this new addition? If the new encoding format has an associated MIME type[1] and it is among the WoT supported file types no changes are required. On the other hand, if the encoding is a property solution the approach is similar to the addition of a new device and a custom communication protocol. Virtual Sensors can take the role of format translator exposing the same sensor data with a known format. Even if covered, a more reusable solution would be to employ dedicated services that are able to translate a data format onto another on-demand.

[1] https://tools.ietf.org/html/rfc2046

Overall the platform can sustain considerably the challenges introduced by the addition of previously unknown devices or services. Even if it requires runtime additions the structural changes are minimal, consequently, we can conclude that it satisfies the interoperability requirement.

4.1.2 Extensibility

Extensibility is the ability of a software system design to withstand future additions and enhancements. In our analysis, we consider three scenarios: adding a new data processing chain, create a new digital twin of a monitored object, and add unplanned functionality like actuation. By design, the first two settings are satisfied by the architecture components; new processing chains or digital twins can be added developing and deploying new Virtual Sensors or Virtual Systems. On the contrary, introducing the concept of actuation in the platform might involve some changes in the design. For example, actuation needs the concept of ownership; since two software agents might compete in the usage of a particular actuator coordinated access to the device is a need in order to avoid resource conflicts.

In summary, the architecture can supports at least two of the defined scenarios. However, the third scenario might result in a profound modification of the presented software structure. On the other hand, adding the actuation functionality might be considered out of scope for a monitoring oriented platform.

4.1.3 Openness

Openness is often used to reference the degree of transparency and collaboration of a given organization. In IT, openness is a compound capability of a software system that is interoperable, portable, and extensible. We already demonstrated how the platform is both interoperable and extensible but is it portable? Consider the following scenario: move one platform component (e.g., a Virtual Sensor) to another operating system or hardware. Thanks to the definition of the WoT runtime and the Scripting API, the software migration operation requires only the installation of a Servient runtime in the target operating system. Therefore, a straightforward porting is

limited to the supported platforms by different runtimes. Although runtime specifications are public, consequently, if needed, it is possible to develop an ad hoc solution to cover an unsupported system or board. Besides these three qualities, an open platform should allow anyone to contribute or implement a part of its services. Take for example the World Wide Web which is a collection of royalty-free technologies that everyone has the right to exploit for the development of Web components without requiring approval or license fees. Founding on the Web of Things standard and W3C specification, the platform is able to support new stakeholders that want to contribute to the production of monitored data and services. In fact, the publishing of Web Things Thing Description is similar to the publication of an HTML page, although it has to take into account the discovery process. To be easier discoverable TDs may be published inside Thing Directories. Therefore, considering this scenario, we can imagine public repositories (i.e., TD directories) like a WoT store where those documents can be freely uploaded by authenticated users.

4.1.4 Final comments

The previous Subsections outlined different scenarios that assessed platform qualities following the SAAM procedure. In summary, the platform has been found to satisfy the requirements for interoperability, extensibility, and openness. No substantial changes are needed to cover the proposed settings, although it might be required to perform some online configurations or software deployments.

One final aspect that is important in the evaluation of an IoT solution is scalability. Given the number of nodes that an IoT application needs to support, scalability measures how much software can maintain its quality of service with an increasing number of users or producers; hence its relevance in an IoT context. In the proposed solution, it is possible to pinpoint two criticalities in regard to scalability: 1 knowledge consumption and production 2 monitoring data consumption. Thing Description Directories and SPARQL endpoints constitute a possible bottleneck for platform users. In fact, they represent the main entry point for Thing Descriptions and therefore, they are critical to access sensor monitoring data and services. To

mitigate this issue implementers can leverage a distributed solution splitting the load into different TDDs. SPARQL based TDD could exploit SPARQL federated query to transparently issue a query to the net of different TDD distributed across the IoT spectrum. Finally, query caching might improve the ability of the system to respond to high demands, as well. On the other hand, too many wot consumers could still impact the performance of the system when it comes to sensing data reading. Consider that most of the constraint devices cannot afford to sustain a long-standing connection with a considerable number of users. Moreover, even with a request-response communication, the possible number of requests per second could exceed the capacity of this kind of node. Consequently, Virtual Sensors, deployed in more capable nodes, might serve as frontend caching sensor responses and handling a greater number of connections or inquiries. Then consumers can read the TD of the constraint devices to find suitable Virtual Sensors and consume their TDs in lieu of the desired device. Unfortunately, these two approches are not yet tested in a real scenario. Therefore, a real measure of the number of concurrent users that a WoT monitoring platform can withstand is yet to find.

4.2 Migration

> **Disclaimer**. This section contain material previously published by IEEE Access [46]

In this Section, we test the performance of the M-WoT framework via a twofold experimental evaluation. First, in Section 4.2.1 we compare different migration policies, including multiple variants of the graph-based heuristic presented in Section 3.3.1.5, on edge scenarios. Then, in Section 4.2.2 we investigate the effectiveness of WT migration mechanisms on the edge-cloud continuum. More in detail we exploit the Structural Health Monitoring use case of the monitoring platform as a playground to test the performance of the proposed heuristic. The characteristics and parameters of each scenario are discussed separately in Sections 4.2.1 and 4.2.2.

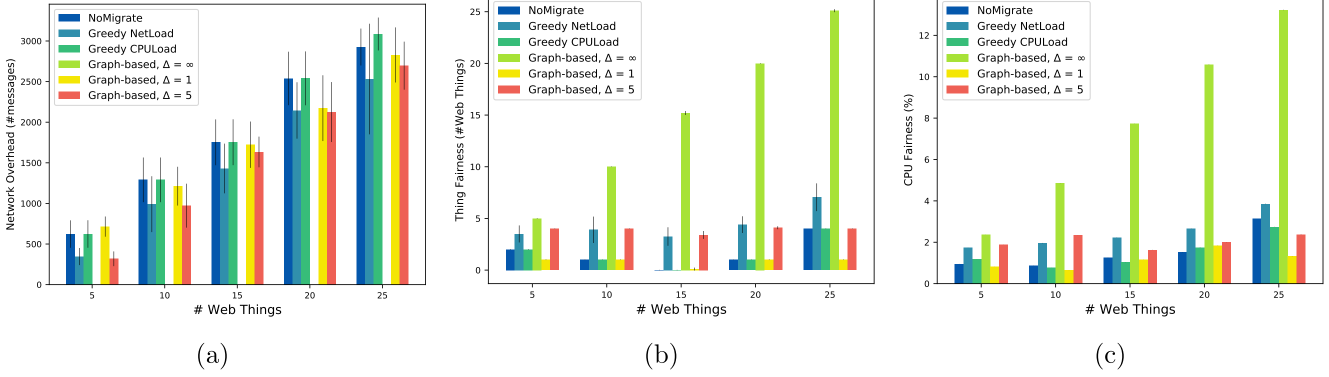


Figure 4.1: The NO, TF and CF metrics for the six policies when varying the number of active WTs are shown respectively in Figures 4.1(a), 4.1(b) and 4.1(c).

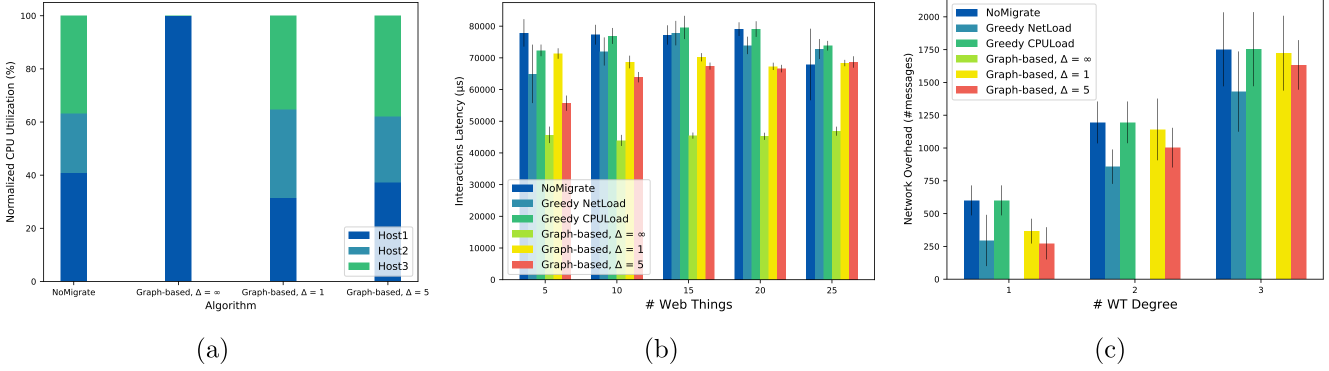


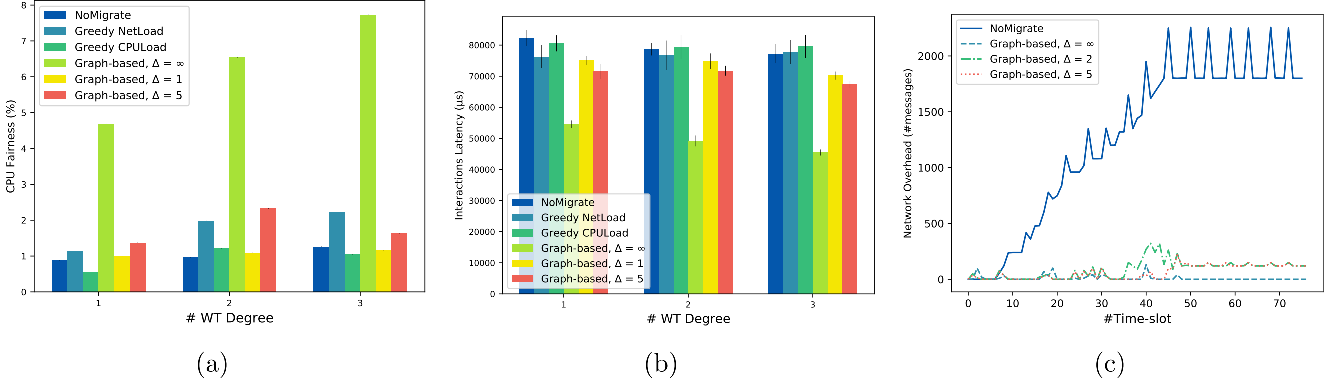
Figure 4.2: The average utilization of each computational node is shown in Figure 4.2(a). The IL metric when when varying the number of active WTs is shown in Figure 4.2(b). The NO metric as a function of the WT degree is reported in Figure 4.2(c).

4.2.1 Policy Analysis

We consider a distributed setup composed of three edge servers (i.e., $N_H =$ 3), physically located at the DISI/ARCES data centers 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 with an Intel Xeon E5440 processor with 32 Gb of RAM. Moreover, the Orchestrator and the TDD 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 N_H 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 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 and of passing them the proper setup (e.g., the list of WTs to consume). Every t_f=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} which represents the discrete intervals used by the Orchestrator to compute a migration plan). 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 metric presented in Section 3.3.1.5, plus other custom metrics introduced to evaluate other factors:

- *Network Overhead* (NO): this is the performance index defined in Section 3.3.1.5 and quantifying the amount of inter-host network



(a) (b) (c)

Figure 4.3: The CF and IL metrics when varying the WT degree are shown respectively in Figures 4.3(a) and 4.3(b). The NO over time-slots in a dynamic WoT deployment where the number of WTs is varied over time is reported in Figure 4.3(c).

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).

- *CPU Fairness* (CF): this is the performance index mathematically defined in [46] and quantifies the fairness unbalance in terms of max-min difference of the average CPU occupation loads among the N_H nodes of the cluster.

- *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.

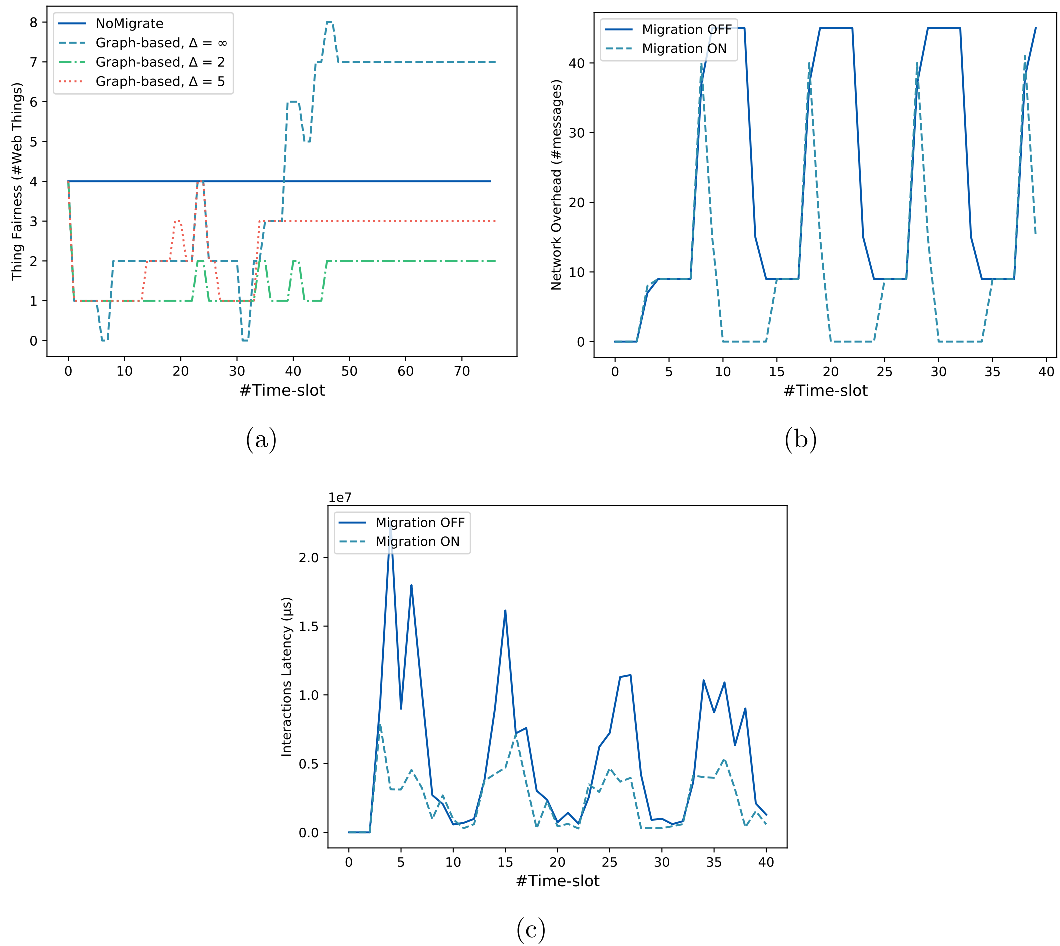


Figure 4.4: The TF over time-slots in a dynamic WoT deployment where the number of WTs is varied over time is reported in Figure 4.4(a). The NO over time in the IoT monitoring use-case is shown in Figure 4.4(b); the processing latency for the same scenario is reported in Figure 4.4(c).

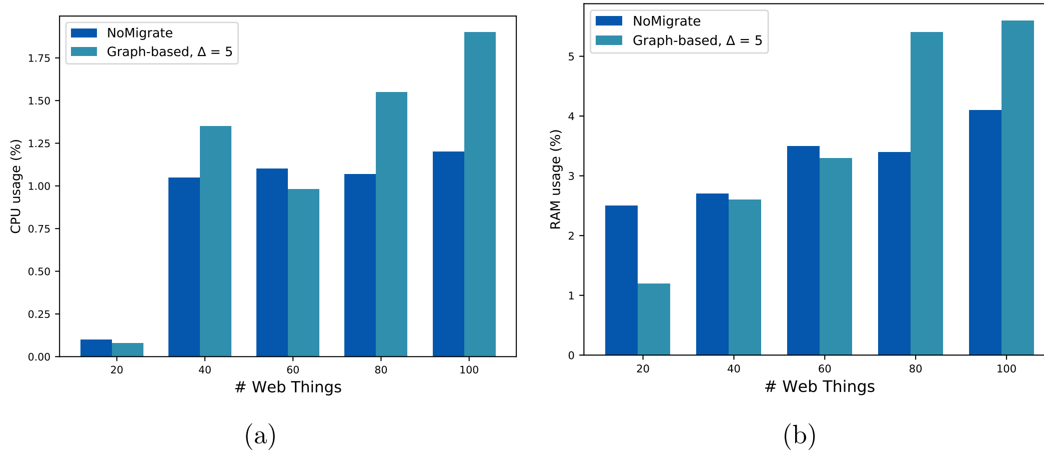
For each configuration, we ran 10 repetitions, and then averaged the

metric values; on each repetition, a random initial allocation of WTs to nodes, and random dependencies among the WTs are considered.
Figure 4.1(a), 4.1(b), 4.1(c) and 4.2(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 4.1(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 *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 4.1(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 4.1(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 our approach. By comparing Figures 4.1(a) and 4.1(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 4.2(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 4.2(b) shows the IL metric for the six policies. Moreover, 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 4.3(a), 4.3(b), 4.3(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 4.3(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 4.3(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 4.2(c) and 4.3(a), we can appreciate again 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 4.3(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 4.3(c) and 4.4(a) we analyze the performance of M-WoT on 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 setup 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 4.3(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 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 4.4(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)

Figure 4.5: CPU load (Figure 4.5(a)) and RAM consumption (Figure 4.5(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 4.5(a) and 4.5(b) show our findings. The results were obtained by sampling the container metrics every second, and then averaging the results for different number 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 3.3.1.4. 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 we plan to investigate as future works.

4.2.2 Use-case Analysis

We consider an IoT monitoring application, which mimics the operations of the SHM deployment of the open monitoring platform presented in Section 3.4. Figure 4.6 depicts a Structural Health Monitoring (SHM) application based on IoT/ WoT technologies [36][37] and the implementation proposed in [46]. 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, sensors (e.g., accelerometers) monitor the building's dynamic response over time. The sensor data is made available through the Sensor Web Things (SWT), exposing functionalities such as data querying and device status updating. The sensor data processing pipeline is handled by migratable Virtual Sensors T1, T2, T3, and T4. Respectively they implement the functionalities of data fusion, data cleaning, data alerting, data forecasting. In Normal mode, T1, T2 run on a shelter/fog node in the proximity of the monitored structure, while a remote cloud server hosts T3 and T4; this introduces some network latency in detecting anomalous/dangerous situations (computed byT3), but at the same time

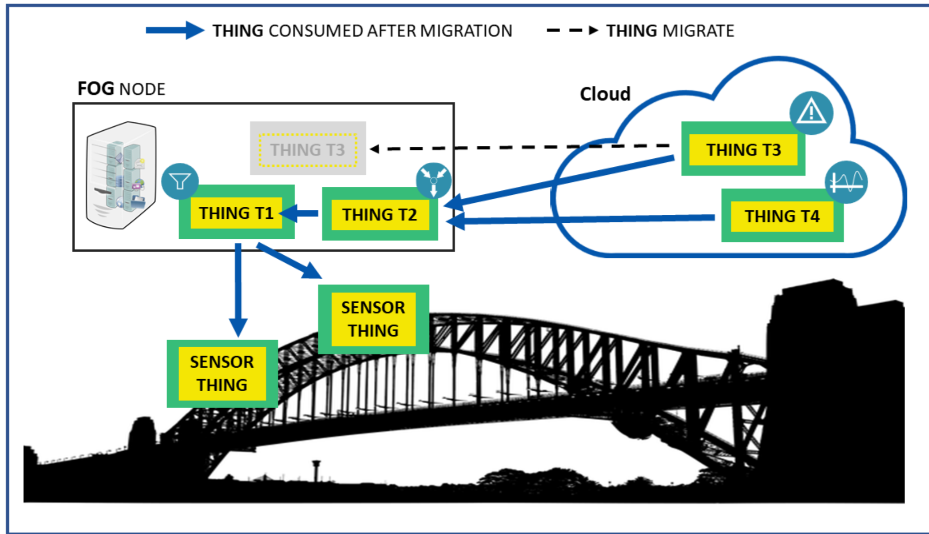


Figure 4.6: A SHM monitoring application.

it minimizes the load on fog nodes. At one point of the system execution, we assume that the system detects consecutive data anomalies on the row data (T2), hence the monitoring system switches its mode from Normal to Critic; this action might also request a higher degree of responsiveness for the diagnostic platform. In the M-WoT environment, the mode change can be automatically managed by migrating the T3 service from the cloud to fog nodes (or vice-versa when the mode switches back to Normal), without any manual need of configuration, and without introducing any explicit signaling mechanism among the involved WTs (i.e., T2 and T3).

In the evaluation process, we considered a simplified configuration that involves a W3C WoT system designed to acquire data from a smart building. The WoT monitoring 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 (using the monitoring platform notation this WT is considered an analytical 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 specially in *Warning* mode, i.e. the time from when the data is acquired to when the forecast value is produced in 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-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 which 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, than it switches to *Warning* mode for 1 second, then again it repeats the same sequence for other two times. Figure 4.4(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 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 4.4(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.

4.3 WoT open points

The W3C Web of Thing standardization is a promising framework that will help the development of real cross silos IoT applications. Still, the Working Group is yet to finish its second charter of recommendation document publications. As such, the standard is continuously evolving, covering new blind spots at every release. This thesis will exploit the concrete use cases presented so far as an evaluation platform for the standard, reporting in this section open issues that might need to be addressed in future specification documents.

One common scenario encountered during the development of the MODRON and SWAMP platform is the sleepy device. In IoT applications,

small devices are bounded by the number of available energy sources in the environment. To obtain a longer operating lifetime engineers usually resort to the employment of processor sleeps intervals that hibernate the board until a specified timeout. During this time the board is deaf to any stimulus and it cannot be programmed or configured. Although some early experiments explore the ability to use specific radio impulses to wake up the board as requested, those sleep interruptions may afflict the overall system performance if not correctly coordinated by device users. Currently, WoT runtimes need to be always fully operational to take part in a WoT application or system hence they cannot run on those constraint sleepy devices. However, those types of sensors might be integrated through the deployment of Virtual Sensors on a mist hub or an edge getaway. Virtual Sensors may completely hide the sleeping lifecycle of the remote board using smart caching and behavior virtualization techniques. Still, it is challenging to define a proxy behavior that works for every possible use case of the system. For example, a system might want to sacrifice a little of stored energy to obtain more sensor data since it knows that it will be a sunny day tomorrow, and sensor batteries could be fully recharged. On the other side, in particularly harsh periods, it might choose to avoid polling that sensor data since it can approximate it using nearby devices. For this reason, a possible future evolution of the W3C WoT standard might also cover sleeping device behaviors promoting them as the first citizen of the WoT architecture. Possible strategies are :

- Provide a vocabulary to express properties affordances and TD metadata related to the current status of the device (i.e, is it a sleepy device? is it currently on?)

- Think about standardized warm-up affordances that can be used to wake up the device at will or schedule it next awaking.

- A support vocabulary to express battery voltage and the remaining operational time might help implementers in the development of an out-of-the-box interoperable sleeping system.

- Pinpoint WoT level error responses which indicate that the current

device is in sleep mode and optionally provide a virtual sensor that might carry the latest known sensed value.

Another issue is Control or, in other words, resource ownership. When it comes to reading data streams from a sensor, the control problem does not arise: ideally, multiple readers can read the stream without interfering with each other. On the contrary, how can different users control the same robotic arm or vacuum cleaner? Most of the time, a single physical resource requires one single control user that configure its behavior and settings. At the moment, the WoT standard does not have the expressiveness to describe such situations or constraints. It is left to implementers to define protocols or algorithms aimed at WT usage management. However, in the future, WoT systems might exploit interoperable protocols to acquire, rightfully, an exclusive or shared exploitation of a physical resource. An example could be an authorization framework based on OAuth 2.0, that leverage on information about the current owner in the TD of the targeted WT (e.g., a link which points to the current user TD). Another idea is to create private only TDs which are issued only to the current owners of the device control and consequently masked to other candidates. A similar discussion is already taking place inside the group regarding long-standing Action resources[2]. In summary, long stating physical process might be modeled as private web or WoT resources (described by a TD or a TD fragment) such as the issuer of the action can control its status of completeness or even cancel the running operation.

A more practical missing feature of the current WoT ecosystem is the lack of constraint runtime implementations. The state of art solutions target small or high-end computing devices leaving embedded developers behind. Since the WoT framework is an IoT framework more attention should be paid to the embedded world enabling makers and companies to exploit WoT building blocks as a catalyst for their device adoption and ease development. Although some early experiments are starting to appear [102] the WoT community needs a full fledge tool-chain that covers both the development, deployment, and runtime of embedded applications. Possible starting points are small constraint enabled operating systems that could serve as ground

[2] https://github.com/w3c/wot-thing-description/issues/899

base for the introduction of the basic servient modules such as protocol bindings and an embedded runtime. In this context, we cite a promising technology that could help in the definition of a standard compilation target and execution environment such as developers can port different libraries to different boards: Web Assembly[3]. Web Assembly runtimes could be employed as wot script or application interpreters on top of the constrained operating system components mentioned above.

Related to the lack of development tools for embedded developers, WoT architecture misses a defined Servient Thing Description specification that provides implementation hints about affordances for remote script management and debugging. Node-wot runtime features a basic action affordance able to execute a simple string in the sandbox javascript running context. However, a more fine-grained solution is needed to address the following requirements needed to offer a full application management layer:

- Define an application bundle format and description
- Define action affordances to stop a particular application (if a multi-application running context is supported) or the only running application.
- Define a description or affordances aimed at describing the process to remotely debug the hosted application. Node-based runtimes could leverage the inspector protocol but a more generic interface is needed to support other runtimes.
- Define different sandbox clearance levels, so that highly trusted scripts can access restricted resources, at the same time constraining the application code to use only the declared resource (similarly to the android Intent model[4])

Introducing this concept at the specification level rather than leaving it as an implementation detail, is crucial to assure the portability of WoT applications. Finally, it creates a common development process that could sprout the creation of shared tools among different runtimes.

[3]`https://webassembly.org/`

[4]`https://developer.android.com/reference/android/content/Intent`

Finally, one last aspect surveyed during these years is the repeating pattern of a typical deployment of an IoT system. We observed that every device is subjected to different life phases that start from its production and end with its decommission. Those phases are well described in the latest editor draft of the W3C Web of Things architecture specification. Figure 4.7 reports the lifecycle specification; as an example consider one SWAMP soil sensor device and its deployment phases. After their creation, each SWAMP soil sensor was bootstrapped with public and private keys for encryption and installed in the soil. Once the remote connection was established it was configurated and calibrated using soil calibration curves obtained from a remote database. Finally, when the crop yielded the sensor was punt in a Maintenance mode to verify that it was not damaged and then re-bootstrapped for the next season. If this recently added sequence flow diagram covers SWAMP requirements it still lacks the correct description of the WoT application lifecycle. As previously mention WoT applications are still underspecified and with no surprise, a full explanation of their possible runtime phases is still missing. An application lifecycle could cover the sleeping phases of the device or temporary context switches carried out by the operating system to prioritize vital processes or functionalities. Taking again the Android metaphor, WoT needs to specify the application runtime phases similar to the Activity lifecycle, so that transient state shifting events are also modeled in the deployment diagram depicted in Figure 4.7. Figure 4.8 introduces a possible state flow for WoT applications.

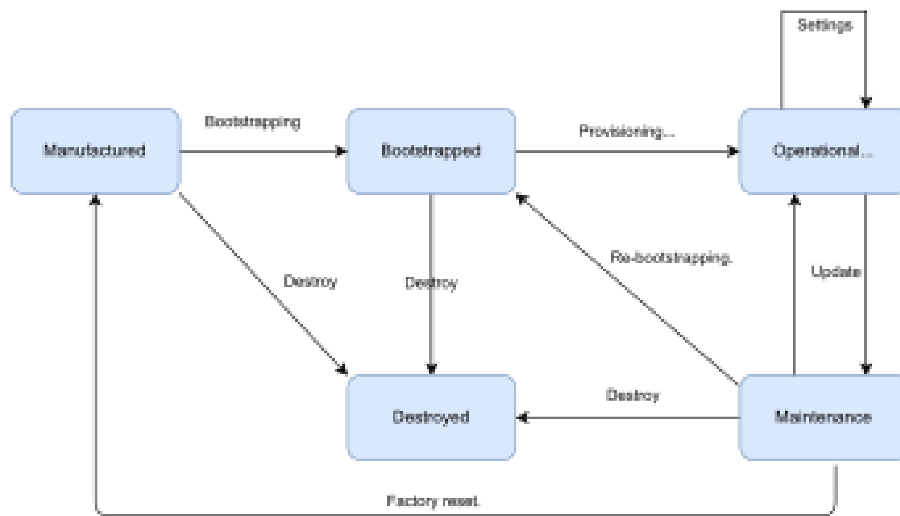


Figure 4.7: The system lifecycle of a WoT enabled device

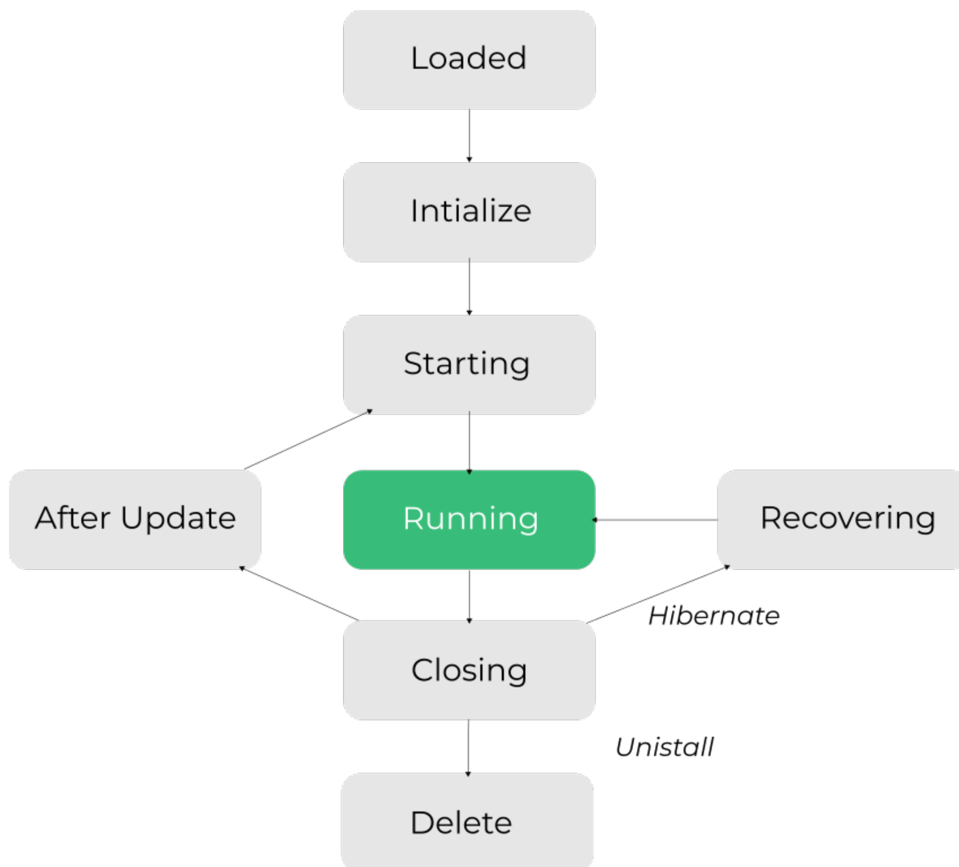


Figure 4.8: A proposed lifecyle for WoT applications.

This page intentionally left blank.

Chapter 5

Future work and conclusions

5.1 Future work

This thesis presented an abstract agent-based architecture for monitoring the environment and civil structures with the Web of Things. Although the platform is able to successfully describe both of the chosen use cases, future works may assess its expressivity in other IoT contexts, for example in an Industry 4.0 setting. The more the platform will be tested, the more it could evolve and cover more complex scenarios fulfilling user needs. The platform evolution is critical to have a common ground for monitoring applications cutting the cost to digitalize a physical asset. Another open point in the platform design is knowledge management and distribution. If in principle, Thing Description Directories could play the role of full function application domain knowledge keepers, more research should be performed on how the knowledge should be distributed across those services. Furthermore, knowledge management services might be heterogeneous regarding protocols used and data format. A full interoperable solution should be designed such that also non-SPARQL endpoints may participate in the metadata description of the platform data sources and objects. One example of a possible knowledge container could be BIM systems to express the geometry and components of a monitored system. Regarding TDDs, the Dynamic SPARQL Directory, presented in Section 3.3.2.7, is planned to be evaluated in the MODRON platform as the core directory service. In particular, we

are interested in its scalability and query expressiveness with respect to standard JSONPath. Moreover, since directories may represent a network bottleneck other discovery methods should be analyzed and tested, such as local multicast or peer-to-peer discovery services.

The document presented a proposal for WoT application migration as an innovative solution to fulfill the Quality of Services requirements in a dynamic context. As an early proposal, the migration framework proposed could be improved along with different aspects. Here we mention the scalability of the platform. Since the migration is completely managed by a central authority (the Orchestrator), it is possible that with high volumes of migratable WTs the overall process is subject to performance degradation. More studies need to be conducted regarding possible solutions however, as a first step, a large-scale experimental setup could provide insights about the concrete performance of the system in a real-world IoT application. For further future works about the Migration- WoT framework, please refer to [46].

Concerning the two use cases provided in Chapter 3 as future work, we foresee the full evaluation of the MODRON architecture in year-long monitoring of an experimental pipeline and a concrete bridge. As the platform will grow the ontologies presented will be extended, covering new terms and properties in order to describe the new setup. In addition, since the SWAMP WoT platform is still under an early development phase it will be polished and studied in comparison to the current online platform based on the FIWARE architecture.

The monitoring platform heavily relies on WoT applications as the main extension points and service producers. However, as briefly mentioned in Section 4.3, WoT ecosystem lacks mature toolset and frameworks able to create and install massive and complex software services. As future work, it is planned the extension of the tools presented in Section 3.4.3 along with the production of different WoT applications for sensors and virtual sensors.

Finally, a complete study of the level of security provided by the WoT layer should be carried out with the goal to assess its feasibility also in a confidential environment.

5.2 Conclusion

Monitoring has become a key feature of every Internet of Things based system. However, the IoT landscape is fragmented in a considerable amount of different monitoring solutions not really interoperable with each other. Even if it has been proved that most of the IoT potential resides in the unexpected interaction between systems, a competitive open platform for monitoring is yet to emerge.

This thesis explored a monitoring architecture based on a novel promising paradigm in the IoT landscape: the Web of Things. Instead of defining yet another protocol, the proposed work focused more on software modules categorization and interactions. It defined a multi-tier structure where different WoT actors coexist to fulfill the monitoring process and different protocols are abstracted thanks to the WoT network interface. Moreover, the architecture was designed with interoperability and openness as core non-functional requirements. Thanks again to the WoT paradigm, it is possible to publish monitoring data in enabled platform instances with a similar process to publishing an HTML page on the web. Finally, the thesis proposed a layered data model categorization to guide implementers in the definition and usage of ontologies in their applications.

The platform requirements were abstracted from two concrete monitoring use cases. The first one was about a Structural Health Monitoring application, able to assess the structural status of industrial instruments and civil constructions. The SHM field is an IoT low adoption field, even if it has been proved that IoT technologies can enhance the overall process. However, due to the heterogeneousness of devices, protocols, and data models, SHM is still carried out with ad hoc solutions and silos applications. Consequently, this document reviewed a previous work where a WoT based SHM platform was proposed as a solution to the aforementioned issues. Specifically, this thesis discussed the platform architecture, implementation details, and an outline of the employed vocabularies.

At the same time, in the second use case, the work in the subject presented a Smart Agriculture application aimed at monitoring water usage: Smart Water Management Platform. Smart Agriculture has similar issues to the SHM field when it comes to the monitoring infrastructure. Sensors

and protocol technologies are usually not homogenous between different deployments. Moreover, monitored data is, again, encapsulated in application silos to fulfill one particular goal (e.g., predict crop yield). Therefore, the SWAMP project was used as a foundation to describe a possible WoT based platform to accomplish the same monitoring capabilities. In the dedicated section, the SWAMP solution was described together with its WoT founded revision and an IoT oriented agriculture/water management ontology was introduced. Finally, the two concrete implementations were compared with the novel open WoT monitoring architecture demonstrating its flexibility and generality.

On Chapter 4, the thesis presented a set of open issues with the novel WoT paradigm. Among the others, it was discussed how Web Things can manage multiple users for actuation or configuration and how to model sleeping devices that use duty cycle techniques to conserve battery energy. Finally, some yearly results of the architecture capabilities were presented and discussed.

In summary, the thesis covered:

- A detailed list of functional and non-functional architectural requirements for a monitoring platform
- A WoT centric software architecture that satisfies those requirements
- An implementation of the architecture in two concrete use cases.
- Discussed a possible non-functional feature that might enhance the runtime scalability of the platform: Migratable Web of Things.
- Propose a dynamic SPARQL Thing Description Directory based on the SPARQL Event Processing Architecture
- Provide a list of open issues related to the WoT standard closely inspired by the platform use cases

Chapter 6

Acknowledgements

In conclusion of this work, I would like to thank all the people that helped me to get through those toughs years of research and study. First of all, I want to mention my supervisor Prof. Tullio Salmon Cinotti, who inspired me to choose this path and pass down his everlasting passion for research. Not only is he a great supervisor, professionally speaking, but he gave me a lot of good life pieces of advice, which is something rare and valuable. I will be forever thankful to him for his kindness and positivity.

Furthermore, I want to thank Dr. Luca Roffia for his research insights and points of view. It all his fault if I end up researching and study the Web of Things. I still remember the meeting where he presented the work that W3C was doing about this odd new paradigm of using web technologies for IoT projects. He made my research project more exciting and worth exploring, I would probably lose interest in the research topics of the group pretty soon without his support.

Another person that believed in this exciting research topic is Prof. Marco Di Felice. As Luca he is was always supportive and insightful. From him, I learned a lot about writing a good research paper and developing the right framework for tests and data analysis.

Thanks to Marco I was able to join my second research group family at DISI Primslab, where I met a bunch of very skillful and interesting people. First, Dr. Luca Sciullo shared with me research topics and goals. Together we designed and foresee much of the results presented in this work. In

particular, I remember that time when we went to the second workshop of the W3C Web of Things where we presented our first results about WoT applications deployments and management. I would like to thank him for the amazing time and interesting discussions. Second, I want to mention Dr. Lorenzo Gigli, an amazing coder and software engineer. We worked closely on the design of the migratable Web of Things where we challenge our expertise to our limits. It was always fun and interesting working with him, which is something rare to find. Finally, I'd like to thank Dr. Angelo Trotta and Dr. Leonardo Montecchiari for their insights and expertise in the IoT world.

Another group that I would like to thank is the whole W3C Web of Things working group. After joining the group I learned so many things both humanly and professionally. The group gave me the opportunity to express my ideas as a young researcher; I felt approached and respected there. I feel so lucky to be part of the standardization process, and I thanks the chairs to let me join this amazing group.

Furthermore, I cannot list in the acknowledgments the SWAMP working group. They were the first international group that I joined. There I learn a lot about the challenges in designing a software system remotely and deploying it in the physical world. Thanks to Prof. Carlos Kamienski and Dr. Jeferson Rodrigues Cotrim.

Thanks also to my first research group inside the ARCES department. Even if at the end of my research period we lost the contacts (also caused by this pandemic), they made my days lighter and joyful. How to not remember our lunch walks to Conad and the funny stories that we told during lunchtime? It was really interesting to learn about their research topic from other areas; to mention a few: Dr. Michelangelo Maria Malatesta, Prof. Nicola Testoni, Dr. Denis Bogomolov, Dr. Luca Perilli, Dr. Alessia Maria Elgani, Dr. Federica Zonzini, Dr. Simone Sindaco, Dr. Filippo Piva, Dr. Francesco Renzini and, all the others.

Professional support is nothing without the love of your family and friends. I would like to thanks my parents for their continuous care and day-to-day help. They were always there for me, and I could have been luckier than this. Moreover, I have an amazing brother who shares with me a passion for computer science and engineering. Thank you, Gianlu for the

good talks and coding challenges together; you are a skillful coder and a great friend; I wish you all the best and a fantastic carrier.

To the love of my life, she knows how much I owe to her. I want to thank her for her love and the happy days together. Thank you, Irene.

Finally, I want to thank all my friends from my home town, from Bologna, and around the world. You did not help me directly with this work, but your smiles and laughs made every blue day easier. Thank you.

This page intentionally left blank.

Appendices

Appendix A

Code listings

```
1      export interface Report{
2      id:string;
3      hostID:string;
4      serviceID:string;
5      nodeStats: NodeStats;
6      interactions: SerializableMap<string,
     InteractionReport>;
7  }
8
9  export interface NodeStats{
10     cpu:Number,
11     memory:BigInt
12 }
13
14 export interface InteractionReport{
15     id:string;
16     reconsumeCounts:number;
17     url?:Url;
18     propertyReports: SerializableMap<string,
     AffordanceReport>;
19     actionReports: SerializableMap<string,
     AffordanceReport>;
```

```
20
21     eventReports: SerializableMap<string,
   AffordanceReport>;
22     summaryReport: AffordanceReport;
23 }
24
25 export interface AffordanceReport{
26     calledTimes: number;
27     computationalCost: bigint;
28 }
29
```

Listing A.1: The definition of the Thing Report object using Typescript.

```
1      WoTHelpers.fetch("https://farm.com/soilstation7331").
    then(async (td) => {
2      WoT.consume(td).then((thing) => {
3          // read and log humidity and temperature sensors
4          setInterval(() => {
5              thing.readProperty("humidity").then((h) => {
6                  ui.humidityGraph.log("Humidity", h);
7              });
8              thing.readProperty("temperature").then((t) => {
9                  ui.temperatureGraph.log("Temperature", t);
10             });
11         }, 10 * 1000); // 10 seconds
12         // if soil dry, sprinkle for 5m
13         thing.subscribeEvent("tooDry", () => {
14             thing.invokeAction("startSprinkler", { "timeout"
    : 5 * 60 });
15         })
16     });
17 }).catch((err) => { console.error("Fetch error:", err); });
```

Listing A.2: An example of a script that uses W3C Scripting API

```
1 {
2   "head": { "vars": [ "book" , "title" ]
3   } ,
```

```
  "results": {
    "bindings": [
      {
        "book": { "type": "uri" , "value": "http://example.org/book/book6" } ,
        "title": { "type": "literal" , "value": "Harry Potter and the Half-Blood Prince" }
      } ,
      {
        "book": { "type": "uri" , "value": "http://example.org/book/book7" } ,
        "title": { "type": "literal" , "value": "Harry Potter and the Deathly Hallows" }
      } ,
      {
        "book": { "type": "uri" , "value": "http://example.org/book/book5" } ,
        "title": { "type": "literal" , "value": "Harry Potter and the Order of the Phoenix" }
      }
    ]
  }
}
```

Listing A.3: A SPARQL query result

```
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX wot: <http://www.w3.org/ns/td>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
DESCRIBE ?td
WHERE {
  ?td rdf:type wot:Thing;
      rdf:type sosa:Sensor;
      wot:title "Test"
}
```

Listing A.4: A SPARQL describe query

This page intentionally left blank.

Bibliography

[1] P. Harry A. Kinnison and T. T. Siddiqui, *Aviation Maintenance Management, Second Edition*, 2nd ed. New York: McGraw-Hill Education, 2013. [Online]. Available: https://www.accessengineeringlibrary.com/content/book/9780071805025

[2] G. S. Brager and R. J. de Dear, "Thermal adaptation in the built environment: a literature review," *Energy and Buildings*, vol. 27, no. 1, pp. 83 – 96, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378778897000534

[3] A. Kumar, H. Kim, and G. P. Hancke, "Environmental monitoring systems: A review," *IEEE Sensors Journal*, vol. 13, no. 4, p. 1329–1339, Apr 2013. [Online]. Available: https://ieeexplore.ieee.org/document/6378389

[4] H. Wang, T. Tao, T. Guo, J. Li, and A. Li, "Full-scale measurements and system identification on sutong cable-stayed bridge during typhoon fung-wong," *The Scientific World Journal*, vol. 2014, p. 1–13, 2014. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/24995367/

[5] N. Mohamed, I. Jawhar, J. Al-Jaroodi, and L. Zhang, "Sensor network architectures for monitoring underwater pipelines," *Sensors*, vol. 11, no. 11, p. 10738–10764, Nov 2011. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3274311/

[6] "The internet of things: Mapping the value beyond the hype," *McKInsey Global Institute*, 2015.

[7] J. Manyika, M. Chui, P. Bisson, J. Woetzel, R. Dobbs, and J. Bughin, "The internet of things: mapping the value beyond,," *McKinsey Glob*, p. 3, 2015.

[8] "Deliverable d03.01 report on iot platform, unify-iot project," 2016.

[9] D. Guinard and V. Trifa, *Building the Web of Things.* Manning Editions, 2016.

[10] K. Matthias, M. Ryuichi, L. Michael, K. Toru, T. Kunihiko, and K. Kazuo, "Web of things (wot) architecture,," Feb 2020, online. [Online]. Available: https://www.w3.org/TR/2020/PR-wot-architecture-20200130/.

[11] M. Jahn, "Economics of extreme weather events: Terminology and regional impact models," *Weather and Climate Extremes*, vol. 10, 08 2015.

[12] D. Letson, D. Sutter, and J. Lazo, "The economic value of hurricane forecasts: An overview and research needs." *Natural Hazards Journal*, vol. 8, pp. 78–86, 08 2007.

[13] E. Lis and C. Nickel, *The ImpacT of extreme Weather events on Budget Balances and ImplIcations for fiscal policy Working paper series no 1055/may 2009.* [Online]. Available: https://www.ecb.europa.eu/pub/pdf/scpwps/ecbwp1055.pdf

[14] I. Manisalidis, E. Stavropoulou, A. Stavropoulos, and E. Bezirtzoglou, "Environmental and health impacts of air pollution: A review," *Frontiers in Public Health*, vol. 8, Feb 2020. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7044178/

[15] H. Willoughby, E. Rappaport, and F. Marks, "Hurricane forecasting: The state of the art," *Natural Hazards Review*, vol. 8, 07 2005.

[16] R. Staff, "Genoa bridge reconstruction to cost 150-200 million euros, official says," Sep 2018. [Online]. Available: https://www.reuters.com/article/us-italy-motorway-collapse-reconstructio-idUSKCN1LP0J2

[17] "Morandi bridge absence costs italy 784 mln euros a year in lost gdp," 2018. [Online]. Available: http://www.xinhuanet.com/english/2018-11/29/c_137638193.htm

[18] M. Abdo, *Structural Health Monitoring, History, Applications and Future. A Review Book*, 01 2014.

[19] F.-K. Chang, *Structural health monitoring 2000.* CRC Press, 1999.

[20] A. Rytter, "Vibrational based inspection of civil engineering structures," Ph.D. dissertation, Denmark, 1993, ph.D.-Thesis defended publicly at the University of Aalborg, April 20, 1993 PDF for print: 206 pp.

[21] B. Peeters, J. Maeck, and G. D. Roeck, "Vibration-based damage detection in civil engineering: excitation sources and temperature effects," *Smart Materials and Structures*, vol. 10, no. 3, pp. 518–527, jun 2001. [Online]. Available: https://doi.org/10.1088/0964-1726/10/3/314

[22] E. Favarelli and A. Giorgetti, "Machine learning for automatic processing of modal analysis in damage detection of bridges," *IEEE Transactions on Instrumentation and Measurement*, vol. 70, pp. 1–13, 2021.

[23] M. Weiser, "The computer for the 21 st century," *Scientific american*, vol. 265, no. 3, pp. 94–105, 1991.

[24] K. Ashton, *How to fly a horse: The secret history of creation, invention, and discovery.* Doubleday New York, NY, 2015.

[25] *Benchmark of MQTT servers*, 2015.

[26] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific american*, vol. 284, no. 5, pp. 34–43, 2001.

[27] S. Kaebish, T. Kamiya, M. McCool, V. Charpenay, and M. Kovatsch, "Web of things (wot) thing description," Apr 2020. [Online]. Available: https://www.w3.org/TR/wot-thing-description/

[28] A. Cimmino, M. McCool, F. Tavakolizadeh, and K. Toumura, "Web of things (wot) discovery," 2017. [Online]. Available: https://w3c.github.io/wot-discovery/

[29] M. Lagally, M. McCool, R. Matsukura, S. Kaebisch, and T. Mizushima, "Web of things (wot) profile," Nov 2020. [Online]. Available: https://www.w3.org/TR/wot-profile/

[30] V. Charpenay and S. Käbisch, "On modeling the physical world as a collection of things: The w3c thing description ontology," *Proc. of the European Semantic Web Conference (ESWC)*, pp. 599–615, 2020.

[31] Z. Kis, D. Peintner, C. Aguzzi, J. Hund, and K. Nimura, "Web of things (wot) scripting api," Nov 2020. [Online]. Available: https://www.w3.org/TR/wot-scripting-api/

[32] P. Asghari, A. M. Rahmani, and H. H. S. Javadi, "Internet of things applications: A systematic review," *Computer Networks*, vol. 148, pp. 241 – 261, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128618305127

[33] C. Aguzzi, L. G. L. Sciullo, A. Trotta, F. Zonzini, L. De Marchi, M. Di Felice, A. Marzani, and T. S. Cinotti, "Modron: A scalable and interoperable web of things platform for structural health monitoring," pp. 1–8, 2021.

[34] C. J. A. Tokognon, B. Gao, G. Y. Tian, and Y. Yan, "Structural Health Monitoring Framework Based on Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 4, no. 3, pp. 619–635, 2017.

[35] L. Alonso, J. Barbarán, J. Chen, M. Díaz, L. Llopis, and B. Rubio, "Middleware and communication technologies for structural health monitoring of critical infrastructures: A survey," *Computer Standards and Interfaces*, vol. 56, no. March 2017, pp. 83–100, 2018.

[36] L. S. Sun, Z. Shang, Y. Xia, S. Bhowmick, and S. Nagarajaiah, "Review of bridge structural health monitoring aided by big data and

artificial intelligence: From condition assessment to damage detection," *Journal of Structural Engineering*, vol. 146, no. 5, 2020.

[37] C. Scuro, P. F. Sciammarella, F. Lamonaca, R. S. Olivito, and D. L. Carní, "IoT for Structural Health Monitoring," *IEEE Instrumentation & Measurement Magazine*, vol. 21, no. 6, pp. 4–9, 2018.

[38] F. Lamonaca, C. Scuro, P. Sciammarella, D. Carnì, and D. Olivito, "IEEE Instrumentation and Measurement Magazine," *Structural Health Monitoring Technologies and Next-Generation Smart Composite Structures*, vol. 21(6), pp. 4–9, 2018.

[39] X. Liu, J. Cao, and P. Guo, "SenetSHM: Towards practical structural health monitoring using intelligent sensor networks," *Proceedings of the 2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud 2016), Social Computing and Networking (SocialCom 2016), and Sustainable Computing and Communications (SustainCom 2016)*, pp. 416–423, 2016.

[40] 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*, vol. 8, no. 2, pp. 45–52, 2019.

[41] 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)*, pp. 57–62, 2019.

[42] P. Barsocchi, P. Cassará, F. Mavilia, and D. Pellegrini, "Sensing a city's state of health: Structural monitoring system by internet-of-things wireless sensing devices," *IEEE Consumer Electronics Magazine*, vol. 7, no. march, pp. 22–31, 2018.

[43] Y. Liao, M. Mollineaux, R. Hsu, R. Bartlett, A. Singla, A. Raja, R. Bajwa, and R. Rajagopal, "SnowFort: An open source wireless

sensor network for data analytics in infrastructure and environmental monitoring," *IEEE Sensors Journal*, vol. 14, no. 12, pp. 4253–4263, 2014.

[44] M. A. Mahmud, K. Bates, T. Wood, A. Abdelgawad, and K. 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)*, vol. 2018-January, pp. 275–279, 2018.

[45] A. Girolami, D. Brunelli, and L. Benini, "Low-cost and distributed health monitoring system for critical buildings," *Proceedings of the 2017 IEEE Workshop on Environmental, Energy, and Structural Monitoring Systems (EESMS 2017)*, 2017.

[46] C. Aguzzi, L. Gigli, L. Sciullo, A. Trotta, and M. Felice, "From cloud to edge: Seamless software migration at the era of the web of things," *IEEE Access*, vol. PP, pp. 1–1, 12 2020.

[47] S. Wang, J. Xu, N. Zhang, and Y. Liu, "A Survey on Service Migration in Mobile Edge Computing," *IEEE Access*, vol. 6, pp. 23 511–23 528, 2018.

[48] K. Ha, Y. Abe, Z. Chen, W. Hu, B. Amos, P. Pillai, and M. Satyanarayanan, "Adaptive vm handoff across cloudlets," *Technical Report CMU-CS-15-113*, 2015.

[49] C. Puliafito, E. Mingozzi, F. Longo, A. Puliafito, and O. Rana, "Fog computing for the internet of things: A survey," *ACM Transaction on Internet Technologies*, vol. 19, no. 2, Apr. 2019. [Online]. Available: https://doi.org/10.1145/3301443

[50] T. Taleb, A. Ksentini, and P. A. Frangoudis, "Follow-me cloud: When cloud services follow mobile users," *IEEE Transactions on Cloud Computing*, vol. 7, no. 2, pp. 369–382, 2019.

[51] C. Puliafito, E. Mingozzi, C. Vallati, F. Longo, and G. 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)*, pp. 97–105, 2018.

[52] H. Abdah, J. P. Barraca, and R. L. Aguiar, "QoS-aware service continuity in the virtualized edge," *IEEE Access*, vol. 7, pp. 51 570–51 588, 2019.

[53] S. Wang, R. Urgaonkar, T. He, M. Zafer, K. Chan, and K. K. Leung, "Mobility-induced service migration in mobile micro-clouds," in *Proceedings of the 2014 IEEE Military Communications Conference*, 2014, pp. 835–840.

[54] C. Zhang and Z. Zheng, "Task migration for mobile edge computing using deep reinforcement learning," *Future Generation Computer Systems*, vol. 96, pp. 111–118, 2019. [Online]. Available: https://doi.org/10.1016/j.future.2019.01.059

[55] K. Kientopf, S. Raza, S. Lansing, and M. 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)*, vol. 2017-Octob, pp. 1–5, 2018.

[56] 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)*, 2017, pp. 957–962.

[57] C. Dupont, R. Giaffreda, and L. Capra, "Edge computing in IoT context: Horizontal and vertical Linux container migration," *Proceedings of the Global Internet of Things Summit (GIoTS 2017)*, pp. 2–5, 2017.

[58] F. Ramalho and A. Neto, "Virtualization at the network edge: A performance comparison," in *Proc. of the IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (IEEE WoWMoM 2016)*, 2016, pp. 1–6.

[59] R. Morabito and N. 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*, no. 607728, pp. 1–6, 2016.

[60] K. Jung, J. Gascon-Samson, and K. Pattabiraman, "Demo: ThingsMigrate - Platform-independent live-migration of javascript processes," *Proceedings of the 2018 3rd ACM/IEEE Symposium on Edge Computing (SEC 2018)*, pp. 356–358, 2018.

[61] C. Kamienski, J.-P. Soininen, M. Taumberger, R. Dantas, A. Toscano, T. Salmon Cinotti, R. Filev Maia, and A. Torre Neto, "Smart water management platform: Iot-based precision irrigation for agriculture," *Sensors*, vol. 19, no. 2, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/2/276

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

[63] A. Sheth, "Changing focus on interoperability in information systems: From system, syntax, structure to semantics," 07 2015.

[64] M. Kolp, P. Giorgini, and J. Mylopoulos, "Multi-agent architectures as organizational structures," *Autonomous Agents and Multi-Agent Systems*, vol. 13, no. 1, pp. 3–25, Jul 2006. [Online]. Available: https://doi.org/10.1007/s10458-006-5717-6

[65] H. Xu, W. Yu, D. Griffith, and N. Golmie, "A survey on industrial internet of things: A cyber-physical systems perspective," *IEEE Access*, vol. 6, pp. 78 238–78 259, 2018.

[66] F. Jalali, T. Lynar, O. J. Smith, R. R. Kolluri, C. V. Hardgrove, N. Waywood, and F. 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)*, pp. 77–86, 2019.

[67] X. Sun and N. Ansari, "EdgeIoT: Mobile Edge Computing for the Internet of Things," *IEEE Communications Magazine*, vol. 54, no. 12, pp. 22–29, 2016.

[68] P. Yu, X. Ma, J. Cao, and J. Lu, "Application mobility in pervasive computing: A survey," *Pervasive and Mobile Computing*, vol. 9, no. 1, pp. 2 – 17, 2013, special Section: Pervasive Sustainability. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574119212000934

[69] "Eclipse thingweb node-wot." [Online]. Available: https://github.com/eclipse/thingweb.node-wot

[70] E. Meshkova, J. Riihijärvi, M. Petrova, and P. Mähönen, "A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks," *Computer Networks*, vol. 52, no. 11, pp. 2097 – 2128, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S138912860800100X

[71] L. Roffia, P. Azzoni, C. Aguzzi, F. Viola, F. Antoniazzi, and T. Salmon Cinotti, "Dynamic linked data: A sparql event processing architecture," *Future Internet*, vol. 10, no. 4, 2018. [Online]. Available: https://www.mdpi.com/1999-5903/10/4/36

[72] J. Umbrich, B. Villazön-Terrazas, and M. Hausenblas, "Dataset dynamics compendium: A comparative study," in *Proceedings of the First International Conference on Consuming Linked Data*, vol. 665. Aachen, Germany: CEUR-WS.org, 2010, pp. 49–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=2878947.2878952

[73] R. Sanderson and H. Van de Sompel, "Cool URIs and Dynamic Data," *IEEE INTERNET COMPUTING*, vol. 16, no. 4, pp. 76–79, 2012.

[74] M. Murth and E. Kühn, "Knowledge-based interaction patterns for semantic spaces," in *Proceedings of the 4th International Conference on Complex, Intelligent and Software Intensive Systems*, 2010, pp. 1036–1043.

[75] ——, “Knowledge-based coordination with a reliable semantic subscription mechanism,” *Proceedings of the 2009 ACM symposium on Applied Computing - SAC ’09*, p. 1374, 2009.

[76] K. R. Llanes, M. A. Casanova, and N. M. Lemus, “From Sensor Data Streams to Linked Streaming Data: a survey of main approaches,” *Journal of Information and Data Management*, vol. 7, no. 2, pp. 130–140, 2016.

[77] S. Schade, F. Ostermann, L. Spinsanti, and W. Kuhn, “Semantic Observation Integration,” *Future Internet*, vol. 4, no. 4, pp. 807–829, 2012.

[78] M. N. Boulos, A. Yassine, S. Shirmohammadi, C. S. Namahoot, and M. Brückner, “Towards an ”internet of food”: Food ontologies for the internet of things,” *Future Internet*, vol. 7, no. 4, pp. 372–392, 2015.

[79] A. Alti, A. Lakehal, S. Laborie, and P. Roose, “Autonomic semantic-based context-aware platform for mobile applications in pervasive environments,” *Future Internet*, vol. 8, no. 4, pp. 1–26, 2016.

[80] A. D’Elia, F. Viola, L. Roffia, P. Azzoni, and T. Cinotti, “Enabling interoperability in the internet of things: A OSGi semantic information broker implementation,” *International Journal on Semantic Web and Information Systems*, vol. 13, no. 1, pp. 146–167, 2017.

[81] F. Viola, A. D’Elia, L. Roffia, and T. Cinotti, “A modular lightweight implementation of the Smart-M3 semantic information broker,” in *18th FRUCT Conference*, 2016, pp. 307–376.

[82] A. D’Elia, F. Viola, L. Roffia, and T. Salmon Cinotti, “A Multi-broker Platform for the Internet of Things,” in *LNCS-Internet of Things, Smart Spaces, and Next Generation Networks and Systems*. Springer, 2015, pp. 34–46.

[83] L. Bedogni, L. Bononi, M. Di Felice, A. D’Elia, R. Mock, F. Montori, F. Morandi, L. Roffia, S. Rondelli, T. S. Cinotti, Others, and F. Vergari, “An interoperable architecture for mobile smart services over

the internet of energy," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2013 IEEE 14th International Symposium and Workshops on a.* IEEE, 2013, pp. 1–6.

[84] F. Morandi, L. Roffia, A. D'Elia, F. Vergari, and T. S. Cinotti, "RedSib: a Smart-M3 semantic information broker implementation," in *12th FRUCT Conference.* SUAI, 2012, pp. 86–98.

[85] L. Roffia, S. Bartolini, D. Manzaroli, A. D. Elia, T. S. Cinotti, and G. Raffa, "Requirements on System Design to Increase Understanding and Visibility of Cultural Heritage," in *Handbook of Research on Technologies and Cultural Heritage: Applications and Environments.* IGI Global: Hershey, PA, USA, 2011, ch. 13, pp. 259–284.

[86] S. Pantsar-Syväniemi, E. Ovaska, S. Ferrari, T. S. Cinotti, G. Zamagni, L. Roffia, S. Mattarozzi, and V. Nannini, "Case study: Context-aware supervision of a smart maintenance process," in *11th IEEE/IPSJ International Symposium on Applications and the Internet, SAINT 2011*, 2011, pp. 309–314.

[87] F. Vergari, T. S. Cinotti, A. D'Elia, L. Roffia, G. Zamagni, and C. Lamberti, "An integrated framework to achieve interoperability in person-centric health management," *International journal of telemedicine and applications*, p. 10, 2011.

[88] D. Manzaroli, L. Roffia, T. S. Cinotti, E. Ovaska, P. Azzoni, V. Nannini, and S. Mattarozzi, "Smart-M3 and OSGi: The Interoperability Platform," *SISS 2010, IEEE First International Workshop on Semantic Interoperability for Smart Spaces, Symposium on Computers and Communications*, pp. 1053–1058, 2010.

[89] F. Vergari, S. Bartolini, F. Spadini, A. D'Elia, G. Zamagni, L. Roffia, and T. S. Cinotti, "A smart space application to dynamically relate medical and environmental information," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 1542–1547.

[90] A. D'Elia, L. Roffia, G. Zamagni, F. Vergari, A. Toninelli, P. Bellavista, A. D'Elia, L. Roffia, G. Zamagni, F. Vergari, A. Toninelli, and P. Bellavista, "Smart Applications for the Maintenance of Large Buildings: How to Achieve Ontology-based Interoperability at the Information Level," in *SISS 2010, IEEE First International Workshop on Semantic Interoperability for Smart Spaces, Symposium on Computers and Communications*, 2010, pp. 1072–1077.

[91] L. Roffia, F. Morandi, J. Kiljander, A. D'Elia, F. Vergari, F. Viola, L. Bononi, and T. S. Cinotti, "A Semantic Publish-Subscribe Architecture for the Internet of Things," *IEEE Internet of Things Journal*, dec 2016.

[92] A. Avižienis, J.-C. Laprie, and B. Randell, "Dependability and its threats: A taxonomy," in *Building the Information Society*, R. Jacquart, Ed. Boston, MA: Springer US, 2004, pp. 91–120.

[93] M. Rinne, E. Nuutila, and S. Törmä, "INSTANS: High-performance event processing with standard RDF and SPARQL," *CEUR Workshop Proceedings*, vol. 914, pp. 101–104, 2012.

[94] C. L. Forgy, "Rete : A Fast Algorithm for the Many PatternIMany Object Pattern Match Problem," *Artificial Intelligence*, vol. 19, no. 1982, pp. 17–37, 1982.

[95] R. Dividino and G. Gröner, "Which of the following sparql queries are similar? why?" in *Proceedings of the First International Conference on Linked Data for Information Extraction - Volume 1057*, ser. LD4IE'13. Aachen, Germany, Germany: CEUR-WS.org, 2013, pp. 2–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=2874472.2874474

[96] F. Viola, A. D'Elia, L. Roffia, and T. S. Cinotti, "Performance Evaluation Suite for Semantic Publish-Subscribe Message-oriented Middlewares," in *UBICOMM 2016, The Tenth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, 2016, pp. 190–196.

[97] C. R. Farrar and K. Worden, "An introduction to structural health monitoring," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 365, no. 1851, pp. 303–315, 2007.

[98] L. Sun, Z. Shang, Y. Xia, S. Bhowmick, and S. Nagarajaiah, "Review of bridge structural health monitoring aided by big data and artificial intelligence: From condition assessment to damage detection," *Journal of Structural Engineering*, vol. 146, no. 5, p. 04020073, 2020.

[99] R.-T. Wu and M. R. Jahanshahi, "Data fusion approaches for structural health monitoring and system identification: Past, present, and future," *Structural Health Monitoring*, vol. 19, no. 2, pp. 552–586, 2020.

[100] "Freshwater is used for agriculture 2017," Mar 2017. [Online]. Available: https://blogs.worldbank.org/opendata/chart-globally-70-freshwater-used-agriculture

[101] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-based analysis of software architecture," *IEEE Software*, vol. 13, no. 6, pp. 47–55, 1996.

[102] V. Charpenay and S. Käbisch, "On modeling the physical world as a collection of things: The w3c thing description ontology," in *European Semantic Web Conference*. Springer, 2020, pp. 599–615.

This page intentionally left blank.

Acronyms

AI Artificial Intelligence. 94

API Application Programming Interface. 4, 29–31, 33, 34, 41–43, 55, 67, 70, 75, 81, 87–89, 91, 98, 99, 110, 114

BOT Bulding Topology Ontology. 100

CoAP Constrained Application Protocol. 22, 23, 26, 81

CPU Central Processing Unit. 66, 73, 118, 119, 122–124, 169

CSS Cascade Style Sheet. 31

CSV Comma Seperated Values. 88

DOT Damage Topology Ontology. 100

GUI Graphical User Interface. 47, 97

HTML HypertText Markup Language. 27, 31, 115, 137

HTTP HyperText Transfer Protocol. 21–24, 26, 27, 29–31, 33, 39, 42, 47, 74–76, 81, 94, 96, 113

IoT Internet of Things. VI, VII, 3–6, 8, 9, 18, 20–23, 25, 27, 29, 30, 33, 36, 38, 39, 46, 48–53, 56–59, 62–64, 66, 72, 76, 93–96, 102, 104–106, 109, 112, 115, 116, 120, 124–127, 129, 131, 135–138, 167–169

IP Internet Protocol. 20, 22–24, 33, 38, 51, 53, 55, 62

IT Information Technology. 45, 114

JSON JavaScript Object Notation. 4, 36, 81, 87–89

JSON-LD JavaScript Object Notation Linked Data. 87, 88, 101

KP Knowledge processor. 29

LoRaWAN Long Range Wide Area Network. 23, 24, 104

M-WoT Migratable Web of Things. 63–67, 71–73, 116, 118, 122–127

MIME Multipurpose Internet Mail Extensions. 113

ML Machine Learning. 49, 94

MQTT Message Queuing Telemetry Transport. 22–24

QoS Quality of Service. 48, 49, 62, 63, 124

RAM Random Access Memory. 118, 123, 124, 169

RDF Resource Description Framework. 28–30, 36, 59, 76, 78, 84, 87–90, 99

RFC Request For Comments. 22

SEPA SPARQL Event Processing Architecture. 77–79, 81, 86, 87, 89, 90, 104, 168

SHM Structural Health Monitoring. 8, 9, 11, 13, 14, 16–18, 46–48, 60, 93–95, 98, 101, 104, 124, 125, 137, 169

SSE Server Sent Events. 21

SSN Semantic Sensor Network. 99, 106

SWAMP Smart Water Management Platform. 91, 93, 102–109, 111, 127, 131, 136, 138, 168

TCP Transmission Control Protocol. 22, 23, 26

TD Thing Description. 33, 34, 36, 42, 55, 59–61, 63, 65–67, 70, 71, 74–76, 87, 88, 90, 91, 95–99, 101, 109, 112, 113, 115, 116, 118, 128, 129

TDD Thing Description Directory. 65, 75, 76, 87–93, 97, 116, 118, 135, 168

TDir Thing Directory. 65–67, 70, 71

UDP User Datagram Protocol. 22

URL Uniform Resource Locator. 26, 38, 74

VM Virtual Machine. 49, 62, 67

W3C World Wide Web Consortium. VI, VII, 4, 8, 31–34, 63–65, 74, 77, 78, 95, 96, 109, 115, 125, 127, 128, 131

WAM Wot Application Manager. 9, 107, 109, 168

WoT Web Of Things. VI, VII, 4, 8, 9, 11, 30–34, 38, 39, 41–43, 55, 57, 58, 60–64, 66, 70, 74, 75, 87, 89, 91, 93–98, 103, 104, 106, 107, 109–116, 118–120, 122, 124–126, 128–133, 136–138, 167–169

WT Web Thing. 63–67, 70–73, 81, 95–99, 103, 104, 109, 113, 116–127, 129, 136, 168, 169

XML eXtensible Markup Language. 27

This page intentionally left blank.

List of Figures

2.1 Different phases of the rebound hammer test 12
2.2 A schematic view of a typical Structural health monitoring system. 15
2.3 A sample of different shm sensors. From the top right corner we found: accelerometers, piezoelectric, strain gauge, laser vibrometer, and fiber optic sensor 19
2.4 Original Web of Things architecture [9] 32
2.5 Interaction schema between WoT agents. 33
2.6 An overview of a WoT deployment. Image taken from [10] . 34
2.7 Consumed and Exposed Thing diagram. See [10] for futher details 35
2.8 The Thing Description data model [30] 36
2.9 Main WoT network interface operations grouped per affordance type 39
2.10 Protocol bindings [10]. 40

3.1 The open monitoring platform abstract architecture. Rectangles represent the different agent categories whereas the grey triangles are possible clients interacting at each level. . 54
3.2 A zoomed view of the open monitoring architecture. Notice how the different layers are distributed across multiple IoT nodes. 59
3.3 A possible knowledge layered organization for WoT based monitoring applications. 61
3.4 Main system components of a Migratable WoT deployment. 65

3.5 The main modules of a Orchestrator agent. Three modules cooperate with the goal to find the optimal allocation plan and actuate it thanks to the migration substrate. 68
3.6 The internal software stack of a migratable servient. See the new added module dedicated to the extraction of monitoring parameters about application behaviour 69
3.7 Sequence diagram of a WT migration event. 71
3.8 The distribution of Thing Description Directories across different IoT layers. It can noticed how different clients can access the knowledge graph from distributed phisical locations. 76
3.9 From the Web of Data to the Web of Dynamic Data. SEPA, SPARQL Event Processing Architecture. 79
3.10 Broker reference architecture. 80
3.11 Core of the broker architecture. EOP, end-of-processing; SPU, Subscription Processing Unit. 83
3.12 SPU manager architecture. 85
3.13 A Thing Description Directory powered by a SEPA microservice 87
3.14 On the left a tree like data structure whereas on the right a graph based data model. 89
3.15 Query translation in a Dynamic TDD 92
3.16 A simplified view of how the Dynamic TDD store Thing Descriptions and other directory relevant metadata (i.e.,modification records and ownership) 93
3.17 Summary of the MODRON architecture. The layers prosed in Chapter 3 are on the right. 96
3.18 Layered knowledge model of the MODRON platform. . . . 100
3.19 The orginal SWAMP platform as proposed in [61] 103
3.20 The SWAMP architecture revised with the Web of Things technologies. SWAMP layers 2,3 and 4 are respectively mapped into the Sensing, Processing, and Analytical layers. 105
3.21 SWAMP vocabulary structure 107
3.22 SWAMP original ontology summarized data model 108
3.23 Initialization of a WoT application using WAM 109

3.24 WoT Farm main screen. On the left there is a code editor with a sample WoT script (see Listing A.2). On the right a 3D view of the simulated farm. 110

4.1 The NO, TF and CF metrics for the six policies when varying the number of active WTs are shown respectively in Figures 4.1(a), 4.1(b) and 4.1(c). 117
4.2 The average utilization of each computational node is shown in Figure 4.2(a). The IL metric when when varying the number of active WTs is shown in Figure 4.2(b). The NO metric as a function of the WT degree is reported in Figure 4.2(c). 117
4.3 The CF and IL metrics when varying the WT degree are shown respectively in Figures 4.3(a) and 4.3(b). The NO over time-slots in a dynamic WoT deployment where the number of WTs is varied over time is reported in Figure 4.3(c). 119
4.4 The TF over time-slots in a dynamic WoT deployment where the number of WTs is varied over time is reported in Figure 4.4(a). The NO over time in the IoT monitoring use-case is shown in Figure 4.4(b); the processing latency for the same scenario is reported in Figure 4.4(c). 120
4.5 CPU load (Figure 4.5(a)) and RAM consumption (Figure 4.5(b)) of the Orchestrator for different numbers of deployed WTs. 123
4.6 A SHM monitoring application. 125
4.7 The system lifecycle of a WoT enabled device 132
4.8 A proposed lifecyle for WoT applications. 133