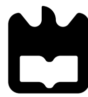
Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2013

Tiago Miguel Gameiro Lam

Interacção Máquina-a-Máquina em Computação Ubíqua

Machine-to-Machine (M2M) in Ubiquitous Computing



Universidade de Aveiro
2013

Departamento de Eletrónica,
Telecomunicações e Informática

Tiago Miguel
Gameiro Lam

Interacção Máquina-a-Máquina em Computação Ubíqua

Machine-to-Machine (M2M) in Ubiquitous Computing

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Rui Luís Andrade Aguiar, Professor associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor João Paulo Silva Barraca, Professor assistente convidado do Departamento de Electrónica. Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president	Prof. Doutor Osvaldo Manuel da Rocha Pacheco Professor auxiliar da Universidade de Aveiro
vogais / examiners committee	Prof. Doutor Paulo Alexandre Ferreira Simões Professor auxiliar do Dep. de Engenharia Informática da Fac. de Ciências e Tecnologia da Universidade de Coimbra
	Prof. Doutor João Paulo Silva Barraca Professor assistente convidado da Universidade de Aveiro (Co-Orientador)

agradecimentos / acknowledgements

Aproveito para agradecer, em primeiro lugar, ao Prof. Doutor Rui Aguiar (Orientador) e ao Prof. Doutor João Paulo Barraca (Co-Orientador) pela oportunidade e por todo o apoio dado durante a elaboração desta dissertação. Desde a fase de concepção até à fase de correcção.
Agradeço ainda ao Prof. Doutor Daniel Corujo (Colaborador) pelo material disponibilizado no início desta dissertação, que se veio a revelar extremamente útil, bem como a todo o grupo ATNoG pela disponibilidade e hospitalidade com que me acolheram.

Por último, mas não menos importante, agradeço aos meus pais, a quem não posso deixar de dedicar esta dissertação, irmã, namorada e amigos por todo o carinho, insistência, paciência e apoio.

A todos vós, um bem-haja!

Palavras Chave

Máquina-a-Máquina, Internet das Coisas, Computação Ubíqua, Redes de Sensores, Plataformas intermédias de integração

Resumo

Apesar de a área das comunicações Máquina-a-Máquina e, consequentemente, a Internet das Coisas, terem sofrido uma grande melhoria relativamente à interoperabilidade, ainda não existe nenhuma solução considerada "dominante" que permita atingir uma interoperabilidade em larga escala, até mesmo global. Desta forma, numa primeira instância este trabalho visa fornecer uma análise teórica de propostas relevantes para a área, onde se analisa maioritariamente como é que essas propostas atingem alguns requisitos essenciais para a Internet das Coisas, como a escalabilidade, heterogeneidade e gestão. Posteriormente, focando-se no standard ETSI M2M, é dado em primeiro lugar uma descrição de alto nível da sua visão, abordagem e arquitectura, e depois, finalmente, de um ponto de vista prático, é ainda apresentada e testada uma implementação funcional de uma gateway condescendente com o standard, o que fornece uma avaliação mais empírica do mesmo.

Keywords

Machine-to-Machine, Internet of Things, Ubiquitous Computing, Wireless Sensor Networks, Middleware integration platforms

Abstract

Although the area of Machine-to-Machine communications and, consequently, the Internet of Things, have undergone a great improvement regarding interoperability, there is still no 'de facto' solution proposal to achieve large scale, even global, interoperability. As a first step, this work provides a theoretical analysis of proposals relevant to the area, mainly analysing how they achieve some essential requirements for the Internet of Things, such as scalability, heterogeneity and management. Later, focusing in ETSI's M2M standard, is first given a high-level description of its vision, approach and architecture, and then, finally, from a more practical point of view, is also presented and tested a functional implementation of an ETSI M2M compliant gateway, which provides an empirical evaluation of the standard.

Contents

Contents i

List of Figures iii

List of Tables v

List of Snippets vi

Acronyms vii

1 Introduction **1**
1.1 Motivation 2
1.2 Objectives 2
1.3 Dissertation structure 3

2 State of the Art **5**
2.1 Technological evolution 5
2.2 Networks of sensors and/or actuators 6
2.2.1 Motes 7
2.3 Internet of Things 10
2.3.1 Concept and Origins 10
2.3.2 First solutions (Verticality) 12
2.3.3 Horizontality (vs Verticality) 13
2.3.4 Current proposals for Middlewares and Frameworks 14

3 ETSI: An European Standard for Machine to Machine (M2M) **35**
3.1 Technical Committee (TC) M2M 36
3.2 M2M Standard 38
3.2.1 Vision 38
3.2.2 Approach 38
3.2.3 High-level Architecture 40
3.2.4 Gateway Functional Architecture 42
3.2.5 Service Capability Layer (SCL) Resource's management 47
3.2.6 Workshops results 48

4 Implementing an European Telecommunications Standard Institute (ETSI) M2M compliant gateway **53**
4.1 Defined objectives 53
4.2 Resources representation 54
4.2.1 Database model 55

4.3 Gateway's Architecture . . . 64
4.3.1 Gateway Service Capability Layer (GSCL) component . . . 64
4.3.2 GA component . . . 73
4.3.3 Communication modules . . . 75
4.4 Procedures . . . 77
4.4.1 Create Application . . . 77
4.4.2 Retrieve ContentInstance . . . 78
4.4.3 Update Subscription . . . 79
4.4.4 Notify . . . 80

5 Evaluation and Results 83
5.1 Deployment Scenario . . . 83
5.2 Results . . . 84
5.2.1 CPU & Memory Usage . . . 84
5.2.2 Notifications measurement . . . 86
5.2.3 Requests measurement . . . 88
5.2.4 Communication analysis . . . 88
5.3 Comparison with Cocoon . . . 92

6 Conclusion 95
6.1 Future Work . . . 95

Appendix A Database selection 97
A.1 Tests . . . 97
A.1.1 Hardware and Software . . . 98
A.1.2 Results . . . 98

Appendix B Code snippets 103

Appendix C Resources's structure 109

Glossary 111

References 113

List of Figures

2.1 Graphical representation of a mote's hardware disposition 8
2.2 Global M2M connection 2011-22 by technology [18] 11
2.3 Example of a vertical solution 13
2.4 Example of a horizontal solution 14
2.5 IrisNet architecture [22] 17
2.6 Example of an Hourglass system with one realized circuit [23] 18
2.7 GSN container architecture [26] 21
2.8 Overview of WebDust's architecture [27] 22
2.9 Services from remote sensor networks [28] 24
2.10 Controller stack [29] 26
2.11 e-SENSE protocol stack architecture [31] 28
2.12 SENSEI support services [32] 31

3.1 ETSI's organization chart [33] 36
3.2 High level architecture for M2M [5] 41
3.3 GSCL functional architecture [5] 43
3.4 Mapping of reference points [5] 43
3.5 GSCL capabilities 45
3.6 Object Network Gateway (ONG) internal architecture [43] 49
3.7 OpenMTC architecture [45] 50
3.8 Embedded Web using ETSI M2M [45] 51

4.1 Representation of collection resources, in the class model 55
4.2 Representation of collection resources and respective normal attributes, in the class model 56
4.3 Representation of collection resources and respective normal and special attributes, in the class model 57
4.4 Representation of Single resources, in the class model 58
4.5 Representation of single resources and respective normal and special attributes, in the class model 59
4.6 Relation representation, in the class model 59
4.7 Final class model 61
4.8 Final physical model - implemented in the database 63
4.9 Package diagram representing the GSCL architecture 65
4.10 Internal representation and respective relations of package *dataAccess* 66
4.11 Internal representation and respective relations of package *gRAR* 67
4.12 Internal representation and respective relations of package *gAE* 68
4.13 Internal representation and respective relations of package *gGC* 69
4.14 Internal representation and respective relations of package *gSec* 70
4.15 Internal representation and respective relations of package *gCS* 71

4.16 Internal representation and respective relations of package *gREM* 72
4.17 Internal representation and respective relations of package *gIP* 73
4.18 Use case example of a Gateway Application (GA) 74
4.19 Example of a GA 74
4.20 Internal representation and respective relations of package *httpComm* 76
4.21 Internal representation and respective relations of package *coapComm* 77
4.22 Sequence diagram of a create resource. 78
4.23 Sequence diagram of a retrieve resource. 79
4.24 Sequence diagram of a update resource. 80
4.25 Sequence diagram of a notification resource. 81

5.1 Demo Illustration 83
5.2 CPU usage 85
5.3 Memory usage 85
5.4 Average time taken to address a notification (concurrently) 87
5.5 Average time taken to address a request (concurrently) 88
5.6 Average time taken to address 10, 50, 100, 500 or 1000 CoAP requests (concurrently) 89
5.7 Wireshark screenshot with a CoAP request (CoAP headers emphasized) 90
5.8 Table 'Attribute' of Cocoon's DB 93
5.9 Table 'Document' of Cocoon's DB 93

A.1 Time performance of INSERT operations 99
A.2 Memory performance of INSERT operations 99
A.3 Time performance of SELECT operations 100
A.4 Time performance of UPDATE operations 100
A.5 Memory performance of UPDATE operations 101
A.6 Time performance of DELETE operations 101

C.1 Resources Structure 109

List of Tables

2.1 Comparison between Sun SPOT, Waspmote and Zolertia Z1 motes 10
2.2 Summary for current proposals for Middlewares and Frameworks 33

5.1 Time (in seconds) per request (average) 89
5.2 Time (in seconds) per request (average, across all concurrent requests) 89

List of Snippets

1 Template of a gateway application 75
2 Payload used for the tests 91
3 XML Primitive sent during the tests 91
4 JSON Primitive 91
5 XSD representation of the application resource 103
6 POJO representation of the application resource 104

Acronyms

Notation	Description
ARIB	Association of Radio Industries and Businesses.
ATIS	Alliance for Telecommunications Industry Solutions.
CCSA	China Communications Standards Association.
CoAP	Constrained Application Protocol.
DA	Device Application.
DSCL	Device Service Capability Layer.
ETSI	European Telecommunications Standard Institute.
GA	Gateway Application.
GAE	Gateway Application Enablement.
GCS	Gateway Communication Selection.
GGC	Gateway Generic Communication.
GIP	Gateway Interworking Proxy.
GRAR	Gateway Reachability, Addressing and Repository.
GREM	Gateway Remote Entity Management.
GSCL	Gateway Service Capability Layer.
GSec	Gateway Security.
HTTP	Hypertext Transfer Protocol.
IoT	Internet of Things.
M2M	Machine to Machine.
NIP	Network Interworking Proxy.
NSCL	Network Service Capability Layer.
SCL	Service Capability Layer.
SDO	Standards Developing Organization.
TC	Technical Committee.
TIA	Telecommunications Industry Association.
TTA	Telecommunications Technology Association.
TTC	Telecommunication Technology Committee.

Notation	**Description**
WSN	Wireless Sensor Networks.

Chapter 1

Introduction

The rapid growth of communications, specially regarding wireless technologies, has allowed an ever increasing number of small devices, like smartphones/tablets, single-board computers and day-to-day accessories (lamps and televisions, for example), not only to be connected together, creating a network of such devices, but in certain cases even be connected to the Internet, the network of networks. That growth, along with the evolution of technological areas like sensors and actuators, has given an impetus to the Internet of Things (IoT), and therefore, to Machine to Machine (M2M) communications as well.

However, the large number of different solutions (either closed or not) that developers and manufacturers have created to enable communications between their own devices, without concerning with interoperability between the different solutions, led to the creation of the so called isolation islands [1]. In there, communications within the same island work seamlessly, but otherwise are unsupported and unpredictable. Thus, in order to avoid that heterogeneity, and allow interactions between the different types of devices and networks, the emergence of a platform that enables the integration of those networks became crucial.

Moreover, the usage of M2M communications among these type of devices, meaning that they are able to interact without human intervention, has increased the amount of information that flows in these networks - regarding sensors, for example, information is periodically extracted from the surrounding environment and sent across the network for future exploitation. To get a better understanding of how much information these networks will have to support, in [2] and [3] is stated that, although each device, in general, doesn't send information to the network in a stream fashion, and the information sent each time is relatively small, the fact that there might be thousands of devices connected to these networks, periodically sending information, gives the sense that the networks are crossed by medium/large streams of information. Hence, using an appropriate platform not only to enable the interaction between the different type of devices/networks, as said above, but also to manage all the information sent by the devices is important for the evolution of these types of networks.

Given the importance of these platforms, efforts and developments have been done over the last years to standardize this segment of M2M, allowing technologies, and approaches of development, to gain some maturation. The large number of developed platforms and solution proposals that address not only the question about the integration of different devices and networks, but also other relevant questions, like optimization and standardization, is the product of those efforts.

Despite all the efforts that have been done, regarding the development and standardization of these platforms, there isn't still any concrete 'de facto' standard solution/proposal, used globally in an seamless way to enable, at a large scale, the integration of different types of

devices and/or networks. There are, however, well defined proposals and initiatives that provide concrete solutions to some of the existent problems.

1.1 Motivation

Apollo is a project funded by Portugal Telecom Inovação (PTIn), and developed in a partnership between PTIn itself and Instituto de Telecomunicações (IT).

Aimed at providing a horizontal platform that supports new services in the area of M2M, more specifically, management, control, and monitoring of heterogeneous networks/devices (like the ones previously mentioned), Apollo [4] is a project that not only tries to integrate different M2M technologies and solutions, but also stimulates external entities to take advantage of provided services. Moreover, Apollo also aims to support a vast set of M2M smart services and applications, such as smart metering and road monitoring.

To bridge the gap between the low-level layers, where the devices are, and the high-level layers, where the services are, thus enabling seamless communications between them, Apollo follows the M2M standard from European Telecommunications Standard Institute (ETSI) [5]. This standard, as explained later in this document, is part of the oneM2M initiative [6], a joint effort that tries to achieve a common and integrated horizontal M2M Service Layer.

Besides defining a horizontal architecture for M2M that encompasses the network domain and the gateway/devices domain, ETSI's M2M standard also defines a structure for storing data (including access permissions), as well as interfaces and messages for enabling interactions between the different entities (including security and integrity mechanisms that ensures that information exchanged not only is correct, but it's also being exchanged between the right entities).

Therefore, this Dissertation objectives are framed within the provisioning of such a platform for Apollo project, as the next section explains.

1.2 Objectives

The focus of this Dissertation is centered in the development of a platform that enables the aforementioned integration between different types of devices and networks. However, despite the platform being a proposal of the ETSI M2M standard, within the scope of this Dissertation is not foreseen an integral implementation of the whole solution, but rather the gateway component of the platform. The ETSI gateway can be considered the most important entity of the platform, since it not only allows the different types of networks and devices to interact with each other, but also provides to the exterior the requested/needed information, thus allowing other parties (users, for example) to manage, monitor and control the information present in the networks of devices, without worrying with the type of devices and technologies used - in other words, it abstracts the interactions of the lower levels, where the devices are present, with the higher levels, where applications/services are making the requests.

Additionally, this document aims also at providing an objective analysis of some of the most relevant solutions/proposals in this area. The idea is not to provide a full detailed analysis, but detailed enough so that, together with a description of the ETSI M2M standard, both help to give a perception of the differences between the solutions and their respective approaches.

1.3 Dissertation structure

This document is organized in 6 chapters. Given that the first chapter is the already presented introduction, and the sixth is where the final conclusions about the carried work are taken, the remaining chapters are:

- Chapter 2: provides a description of the state of the art. The core concepts of Wireless Sensor Networks (WSN), IoT and M2M are presented in this chapter. Additionally, it also gives an objective analysis of some solution proposals relevant to the area;
- Chapter 3: gives an overview of ETSI M2M standard, including its vision, approach and architecture. Other similar work, obtained from ETSI's workshops, is also presented in this chapter;
- Chapter 4: starts by enumerating and explaining the objectives that were defined for the implementation of the ETSI gateway. Later, it describes the architecture that was conceptualized and implemented during this work, along with an explanation of how the components of the architecture interact with each other;
- Chapter 5: presents the deployed demo, the tests performed against the gateway, the results obtained, as well as insights about those results. Furthermore, this chapter also provides a comparison between this implementation of the Gateway Service Capability Layer (GSCL) and Cocoon's implementation.

Chapter 2

State of the Art

2.1 Technological evolution

The technological revolution of the twentieth century, that led to the miniaturization of the electronic circuits, enabled the creation and, more importantly, the proliferation of small scale Integrated Circuits (ICs) and other electronic components. Consequently, other important electronic components, as micro-processors and micro-controllers, which are no more than ICs tailored for specific tasks, followed the same path.

Nowadays, although they may go unnoticed by most people, even knowledgeable people who are not looking for it, the reality is that these electronic components of small dimensions are widespread all over the world and we interact with them, directly or indirectly, on a daily basis. From our home, the alarm, the air conditioner, or even the microwave, to the office, the photocopier, or the printer, all these systems are composed of small sized electronic components, as the ones referred above, that work together to carry out the primary tasks - trigger an alarm sound after sensing an anomaly, warm up the bedroom temperature, etc.

Moreover, the small dimensions of such electronic components allowed the development of devices of increasingly small dimensions, creating an opportunity for them to be used in many different areas. Sensors and actuators are a good example of these components. They are used either as standalone devices, like smart readers, to take readings or actuate in their surrounding environment, or integrated in more complex devices, to provide additional functionalities to end consumers, and thus raising their interest. The existing smartphones, disseminated across the world, are an example of systems that recur to these small components, such as the accelerometer, the Global Positioning System (GPS), the proximity sensor, among others, that provide them additional capabilities.

In comparison with other electronic components, sensors and actuators have the particularity of being able to exchange information directly with the real world, which means that the sensors are able to withdraw information from their surrounding environment, by monitoring it, such as the actuators are able to actuate in it, modifying it, if needed. Hence, it's important to note that the sensors and actuators, together, allow to reduce the existing gap between the real and the digital worlds, since they make the necessary translations to enable interactions between both worlds.

The downsizing of sensors and actuators, as aforementioned, was an important step that allowed these devices to be integrated in society (either as standalone devices, or as integral parts of more complex devices). However, another equally important step for the wide adoption of these devices was their ability to interact wirelessly, i.e., exchange information between each other without the need of having any wires attached. In these interactions, the

emergence of wireless network interfaces, although not being pioneer, largely contributed for the massification of these small systems. Due to the freedom conceded to the equipments, wireless network interfaces allow these systems to disseminate throughout the globe, being smartphones and tablets the biggest examples of success. Nowadays many others systems already incorporate wireless network interfaces to communicate with the exterior, like fridges, vehicles, digital albums, printers, as many others - it's the connection of all of these 'things' to the Internet that creates the so called IoT. Moreover, the massive appearance of small dimension devices, as the sensors and the actuators, with integrated wireless network interfaces, will not only contribute to increase even further the number of devices connected around the world, but will also change the way as the information flows and is accessed in networks as the Internet.

The opportunities created around these sensors and actuators, given not only their portability, but also their amount of resources (that although limited are enough for performing some small tasks, like sensing/actuating and reporting results), and their relatively low cost, are not confined only to one specific area. Their versatility allows them to be applicable in virtually any area. Agriculture, for measuring the soil humidity, home automation, for temperature measuring, or package tracking are just some examples.

2.2 Networks of sensors and/or actuators

Given that sensors and actuators are only able to monitor and/or actuate upon limited boundaries around their area of deployment, the use of only one or a few sensors doesn't cover much use cases - although suitable for small or personal use cases, i.e. using them to measure the energy consumption of one's personal house. As so, it is common to aggregate a variable number of these devices and use them to more accurately monitor larger and more complex areas, like greenhouses, buildings, inventory tracking and industrial control, for example. Yet, the real advantages of these large aggregations of sensors and actuators are only possible due to the devices being able to interact with each other, through their communication capabilities (mostly wireless). These aggregations are referred as WSN, a term that although old, can be traced back to the Distributed Sensor Networks (DSN) program at the Defense Advanced Research Projects Agency (DARPA) in 1980 [7], as gained a new boost in recent years, helped by other concepts that will be approached later on, as M2M and IoT.

Albeit obviously more expensive than one or a couple of sensors, WSN are able to withdraw not only much more information than only one sensor, but the retrieved information is also more accurate and precise. For example, reporting that a whole plantation has been affected by a fungus would only be possible with a WSN distributed across the plantation. With only a few sensors, one could only take the conclusion that that section of the plantation (the one monitored by the sensors) has been affected by the fungus, which although critical doesn't require all the concerns/countermeasures as in the case of the whole plantation being infested.

Still, there is another (very useful) example that shows how this type of networks are useful for harvesting information across large areas [8]. There, over four hundred sensors were deployed to monitor temperature, humidity, pressure, light, air quality, motion, and both RF and audio noise levels during the 2013 Google I/O event. All those values were then gathered and analysed to provide heat maps and other data visualizations that can be seen/used by the rest of the world.

2.2.1 Motes

Motes are defined as being nodes in wireless networks of sensors, that besides their sensory capability, that allow them to collect information from the surrounding environments, are also able to process and communicate with other network nodes. Additionally, actuators are usually added to motes, with the objective of providing them a bigger interactive capacity towards the surrounding environment.

However, despite the processing and communication capacity of these motes, it's necessary to have into attention that the resources, independently of the manufacturer, are very limited, when compared with other devices like smartphones, or personal computers. These limitations arise not only due to their small physical dimensions, but also because of the trade-off that is necessary to make between performance and battery autonomy, which in many cases it's mote's only source of power.

Therefore, given the importance of these motes to IoT, not only because of their possible future abundance, but also because they will be the ones responsible for providing interaction with the real world, as already stated in the beginning of this chapter, a description about their common hardware composition, and also about some of the most relevant motes and Operative Systems (OSes) follows.

2.2.1.1 Hardware

Although there are many different types of motes, each one with its own characteristics, regarding their hardware composition they usually have the same main components, which are:

- Controller: This component is where all the instructions/tasks and data processing are executed. It is also responsible for controlling the other components. The most common choice among motes for this processing unit is a micro-controller, due to its low cost and power consumption.
- Transceiver: It gives to the mote the ability to transmit or receive information - almost all motes use radio for transmitting/receiving information, as Bluetooth, WiFi, ZigBee, 6LoWPAN, for example. This is usually the component that drains more energy from the mote.
- Memory: It's usually divided in two types, application or user related data (external memory), and program and data memory. Due to cost constrains, the former is usually Flash, while the latter is a small amount of RAM/SRAM.
- Power: It provides the energy that feeds the mote. Since motes are usually deployed and expected to work for extended periods of time without maintenance, sometimes even in hard to reach locations, this component's quality is crucial. Batteries (of lithium-ion, usually) are used as a source of energy supply across almost all motes.
- Sensor(s)/Actuator(s): These are the components responsible for doing the real world measuring/actuation, which is then converted to the digital world by and Analog-to-Digital Converter (ADC) in the case of a sensor, or converted from the digital world to an action in the real world in the case of a actuator. Some motes have more than one sensor/actuator, being able to monitor/actuate more than one property of the surrounding environment.

Figure 2.1 provides a graphical high-level representation of motes's common hardware disposition.

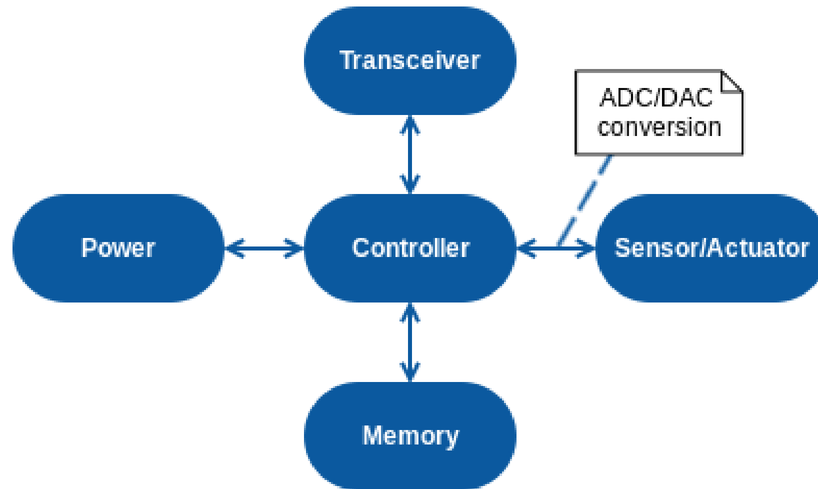


Figure 2.1: Graphical representation of a mote's hardware disposition

2.2.1.2 Generic Motes

Mica, Mica2 and MicaZ, were one of the firsts platforms that got coined with the term 'mote', [9], paving the way for the evolution that provided the variety of motes that are available today.

Therefore, this subsection goal is not to give a brief presentation about all the available motes, but rather the motes that are considered relevant for this Dissertation. Their relevance, however, is based on two factors: i) they are considered a good example of how motes have evolved, and are being integrated in WSN, ii) they were used practically, hence some experience were acquired through their use (Sun SPOTs and Waspmotes - Waspmotes, for example, are the devices that compose Apollo's WSN, which interacts directly with the gateway).

It's important to note that the sensors here presented, are generic motes, meaning that they don't fill a specific purpose in the WSN, but rather a general one.

2.2.1.2.1 *Sun SPOT*

The Sun SPOT mote, developed by Sun Microsystems, uses the JME (Java Micro Edition) to provide a small footprint Java Virtual Machine (JVM), named *squawk*, that is executed in the microprocessor and behaves like an OS. It's in this virtual machine that the developed applications, which ultimately dictate the behaviour of the device, are executed.

This mote has the ability to be programmed Over The Air (OTA), meaning that a developed application can be installed in the Sun SPOT using the OTA programming functionality. Additionally, this functionality can also be used to configure the devices without the need of local access. The basestations are used with that purpose, that is, they are connected to PCs (or any other machine used for the development) to allow their communications with the Sun SPOTs. Thus, anything from simple interactions, to the installation of applications in the devices, can be done remotely through OTA.

This mote is equipped with a 400 MHz microprocessor, 1 MB RAM, 8 MB FLASH, a rechargeable lithium-ion battery of 3.7V and 750 mAh, and a radio antenna of 2.4 GHz, compatible with the 802.15.4 standard [10].

The hardware upgrades that the Sun SPOT suffered through the times resulted in above referred specifications, that, when compared with others motes stay above the expectations, having a light, temperature and accelerometer sensors, and even giving the possibility of extending it with more sensors, connected through its extension inputs.

2.2.1.2.2 *Waspmote*

Despite its modest specifications, when compared with the Sun SPOT mote, an 8 MHz microprocessor, 8 KB SRAM, 4KB EEPROM, 128 KB FLASH and a maximum of 2 GB SD card (which concedes greater capacity for data storage), the Waspmote [11] provides not only a wider variety of communications, allowing the use of modules compatible with the ZigBee protocol (built over the 802.15.4 standard), GSM/GPRS, Bluetooth, Wifi and RFID/NFC, but through the use of the extension proto-boards, it also enables the use of a variety of sensors, as CO2, radiation, humidity and luminosity, for example. Depending on the module/antenna and technology used, communications can have a big range of distances. For example, using the ZigBee module it's possible to communicate from 500 meters up to 7 kilometers of distance. Additionally, there are still provided two batteries (one auxiliary), being one battery of 3.3V-4.2V and 100 mA and the other, the auxiliary, of 3V.

In this case, the programming language used for the development of applications that are going to be executed in the devices is C.

Regarding ZigBee, to enable the interaction between a Waspmote using a ZigBee module and a normal computer, one has to acquire a ZigBee basestation, which is no more than a ZigBee module attached to a serial cable. Thus, through the PC, using the serial cable, one can send commands that will be sent from the attached ZigBee module to the respective Waspmotes.

Just like the Sun SPOTs, the Waspmotes also support OTA programming, for the remote deployment of applications.

2.2.1.2.3 *Zolertia Z1*

Z1 [12], equipped with a 16 MHz microprocessor, 8 KB RAM, 92 KB FLASH, with an extra 16 MB of external FLASH and 2xAA/AAA as a power source, it's another general purpose mote with modest specifications. In comparison with the other presented motes, this mote as two important advantages, which is the fact that it supports communications using 6LoWPAN, enabling direct interactions with the Internet, through IPv6, and also that it supports the most widespread open source OSes in WSN, which are TinyOS and Contiki. The usage of an OS makes the development and deployment of applications to the devices much more friendly, for the reasons pointed below, in the section 2.2.1.3.

Besides that, this mote only provides a 3-axis accelerometer and a built-in temperature sensor. However, they still provide expandability, not only through the use of their Expansion Connector, which provides up to 52-pins for digital and/or analog I/O, digital buses and interrupts, but also through the soldering of their Phidgets (which are easy to connect sensors) to the its PCB, providing Z1 with more sensors and without much effort.

Finally, like the Sun SPOTs, these motes also support communications using the 802.15.4 standard.

For a complete picture, Table 2.1 provides a side to side view of each sensor main characteristics.

Sensor Name	Microcontroler	Communication	Program Memory	External Memory
Sun SPOT	ARM 920T (400 MHz)	802.15.4	1 MB RAM	8 MB Flash
Waspmote	Atmel ATmega 1281 (8 MHz)	ZigBee, GPS/GPRS, RFID/NFC, Bluetooth, Wifi	8 KB SRAM	128 KB Flash ROM, 4 KB EEPROM, 2 GB SDCard
Zolertia Z1	Texas Instruments MSP430F2617 (16 MHz)	802.15.4, 6LoWPAN	8 KB RAM	92 KB Flash

Table 2.1: Comparison between Sun SPOT, Waspmote and Zolertia Z1 motes

2.2.1.3 Operative systems

The number of OSes developed specifically for motes has been increasing. Although these OSes demand more resources from motes, for their execution, they have the advantage of abstracting the development of applications from the hardware, being the OS itself responsible for supporting concurrency (where possible), minimizing the energy consumption, exploring the mote resources in a efficiency way, and even supporting particular protocol stacks and algorithms that ease communications. From the existent OSes, the ones that have been gaining more recognition, by being already used in a variety of different motes, is TinyOS [13] and Contiki [14] (both open source).

Applications deployed in TinyOS are coded in the network embedded systems C (nesC) language, while in Contiki the applications are developed using the C language.

The major advantages of Contiki, in comparison with TinyOS, is the fact that the former provides a micro TCP/IP stack, that enables it to communicate with IPv4 networks, a IPv6 stack, meaning that it is able to communicate with IPv6 networks, and even allows IPv6 packets to be transmitted/received in networks based in the 802.15.4 standard, using 6LoWPAN. However, on the other hand, the greater advantage of TinyOS is that, being more mature, it contains a bigger community and has been more used/tested - being more stable.

To finalize, the Tinynode [15] and the Rene are examples of motes that support the TinyOS, while WisMote [16] and Redbee are motes that support Contiki. There are still other motes that support both OSes, like TelosB [17] and Zolertia Z1.

2.3 Internet of Things

2.3.1 Concept and Origins

With the appearance of small scale devices capable of being connected to the Internet, like sensors and actuators, and other everyday devices, like televisions, door keys, or even cars, together with the other devices that have been an assiduous part of the Internet for a long time, like personal computers, printers, smart-phones, it's estimated [18] that the number of machines connected to the Internet will exponentially raise in the coming years, with the possibility of reaching the dozens of billions of connected machines, Figure 2.2. Therefore, the dissemination of all the aforementioned systems and devices across the world, as well as their integration in society, gave origin to the IoT concept.

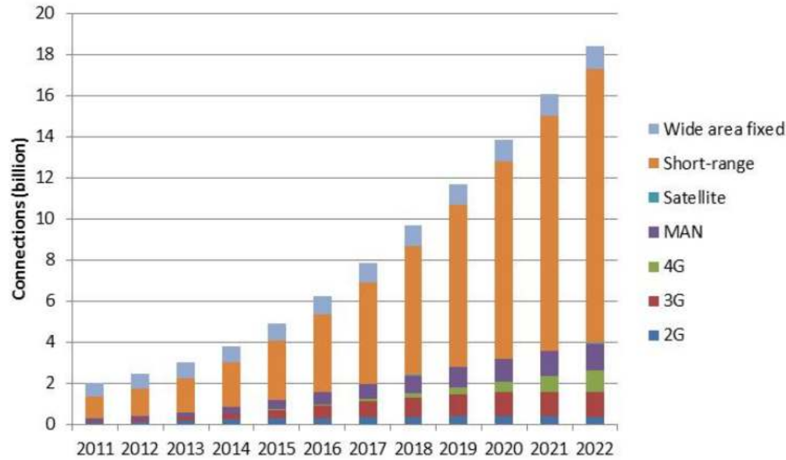


Figure 2.2: Global M2M connection 2011-22 by technology [18]

However, the concept of IoT is not entirely new. Although the term is relatively recent, from 1999 [19], the concept has the same paradigm as the concept of ubiquitous computing, which is an older concept.

Ubiquitous computing was introduced for the first time in 1991 by Mark Weiser [20], as a new paradigm where computing appears everywhere and anywhere, and not only in desktop computers. In that article, Weiser also envisioned how devices and the systems of the future would seamlessly integrate with our quotidian. From an alarm-clock asking if one wishes a cup of coffee, just before the awakening hour, and then interacting with the coffeemaker to prepare it, to electronic windows that would allow one to see, for example, who is in a specific division of the house, or even suspicious activities in the surrounding areas of the house. Although some of these visions have materialized, they lack the seamless integration with our lives, and only then, as Weiser states, they will become part of us - “The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it”.

The IoT, on the other hand, despite of also being based in a paradigm where computing appears everywhere and anywhere, just like in ubiquitous computing, it differs in that it focus on how the information is generated. Contrary to what has been happening until now in existent networks, where the existent information is mainly provided by humans, in the so called networks of the future, the M2M communications will be used at large scale, meaning that the information is generated and forwarded from machine to machine without the need f human intervention. This automation will allow not only a better reliability for information, since it is obtained directly from devices that are interacting with the real world (sensors and actuators), but that same information will also be ’fresher’, since information doesn’t have to be updated by humans, which delay the process.

The research surrounding IoT, however, is still in its childhood. The various names given to IoT, as Internet of Everything, Internet of the Future or Internet of Objects, show that no ’final’ definition has yet been achieved. Despite that, according to ITU-T [21] IoT is defined as “A global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies”.

The concept of IoT and the numbers presented in the first paragraph of this section only show that today’s Internet, as we know, is changing, and in the future (hence also

the concept of Internet of the future) will be composed by numerous networks, in a much superior quantity than in the actual Internet, and those networks will also be composed by a larger number of devices and systems of different dimensions and areas. All the devices and systems, organized in networks, will allow a much greater and better interaction with the real world, not only because of the sensing and actuation capability provided by the sensors and actuators, as referred before, but also because things are spread and incorporated into our quotidian, which will allow us to take advantage of the information present in the networks for self benefit (much of the time without even noticing it, as Weiser envisioned).

2.3.1.1 M2M

The term M2M and the term IoT are commonly used together to describe the next generation of the Internet. However, the fact that IoT is most closely associated with M2M, and even further, the fact that the term M2M is used in a widely and loosely way to designate many different scenarios can cause an understandable confusion about the relation between these two concepts.

The concept of Machine-to-Machine communications is not new, that is, communications between machines without the need of human intervention have been happening for a long time, and it's a mature market, it cuts costs, time and improves efficiency. Communications between machines, like routers and servers, and even between sensors and machines, for energy conservation and tracking ship fleets, for example, are a common use case.

However, the hype behind IoT, and the use of this term to describe the underlying communications has provoked, implicitly, an evolution of the term. An evolution that its acronym doesn't reflect. Thus, despite the term literally meaning Machine-to-Machine, what it means, in the context of IoT, is that Machines not only are able to talk to each other, without human intervention, as have been happening in the past, but they are also able to give meaning to the received information, in order to take automated actions.

To summarize, M2M can be thought as being a subset of IoT, in the sense that it's M2M that provides the connectivity that enables the IoT capabilities. Without technologies that enable M2M communications, i.e., if machines weren't able to assess the received information, IoT wouldn't be possible.

2.3.2 First solutions (Verticality)

Taking into account the simplicity of the first use cases, where WSNs were used in small scale environments to monitor one specific property (like the temperature of a room, for example), the first solutions created to take advantage of WSNs weren't very complex.

As depicted in Figure 2.3, generally, these systems architecture was composed by only two highly coupled components, the WSNs, and an application (service) which would take advantage of the network functionalities. The development of these systems, however, was mainly focused on the service layer and its interactions with the WSNs (to perform the requested operations), not having concerns with important issues, like scalability, and the ease of reutilising common functionalities, that are crucial in more complex systems.

Even nowadays, the initial simplicity of development and the ease of deployment provided by this model, referred to as vertical, are the main reasons that lead to its adoption for the rapid implementation of solutions that take advantage of networks of devices.

However, the use of this model gives origin to unwanted dependencies, since the developed services have previous knowledge of how to address the devices, and which protocols to use in order to interact with them for performing the requested operations. This strong coupling

dictates that evolving from such systems and maintaining them becomes a very difficult task, as enumerated below:

- The development of services is slow. Despite implementing the algorithms needed for processing the received information, as normal, services also need to implement the low level interactions with the devices gateways, or the devices themselves. Hence, services have the responsibility of dealing with the communication with the devices, which becomes cumbersome when the number of networks arise;
- There is no reutilization, meaning that functionalities instead of being shared across all services are replicated. Hence, the module(s) that allow the communications with the devices have to be implemented by all services;
- It's hard to accomplish interoperability among different solutions, since each one implements its services in a different way, using its own specifications and protocols to communicate with the networks of devices.

Figure 2.3 gives an overall idea of the architecture of a vertical system. As can be observed, each service is responsible for implementing its own infrastructure, and thus enabling the communication with the network of devices.

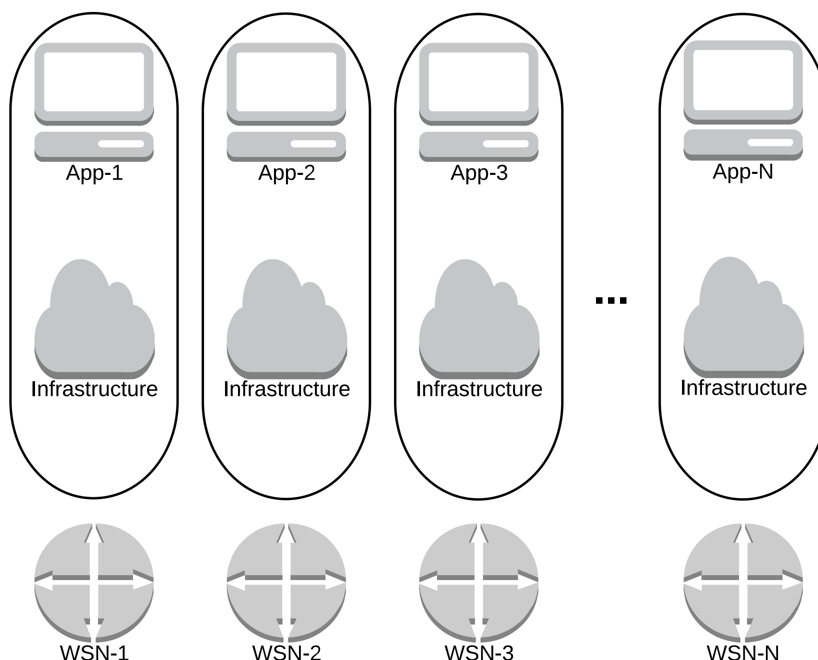


Figure 2.3: Example of a vertical solution

To overcome the above mentioned deficiencies imposed by the vertical solutions, horizontal approaches started to arise in more complex systems. The next section shows its main advantages against vertical systems.

2.3.3 Horizontality (vs Verticality)

The vertical solutions, while being able to fill the requirements of more complex systems (at least in an initial phase), are not indicated to be used in such systems, since, as described in the previous section, among other factors, the evolution and scalability that these systems

provide is very limited. Given that the objective is to surpass the vertical systems inflexibility, that are strongly coupled, researches and proposals have fallen under the study and development of more flexible horizontal systems that allow not only the execution of services above the networks of devices, as the vertical systems do, but also the easy integration of new networks of devices and services, and also the reutilization of common functionalities, like management of devices, routing, connectivity and security.

Horizontal systems try to overcome the deficiencies present in the vertical systems by providing one additional layer, a middle layer, that is used to abstract the communication between services and the WSNs, as well as providing common functionalities. Taking the communication burden from the services not only makes these systems more easy to maintain, since the implementation, or modification of services are no longer coupled to the devices and their specific technologies, but also makes them more scalable. Furthermore, the horizontal systems also have the upside of allowing, in overall, to minimize the costs on the operators side, since reutilization facilitates and speeds up the addition of new functionalities (the addition of a new service for measuring humidity, when one to measure temperature already exists, becomes an easy task, for example).

Figure 2.4, together with Figure 2.3, allows to graphically visualize the already referred big difference between the vertical and the horizontal systems, i.e., the main infrastructure is shared among all services, meaning that communications with the devices, among the other common functionalities already annunciated, have to be implemented only once.

Furthermore, the achievement of the IoT depends in great part in these horizontal frameworks, since they have the responsibility of allowing, in a scalable way, the integration of different networks of devices and services, as well as the interactions among every device.

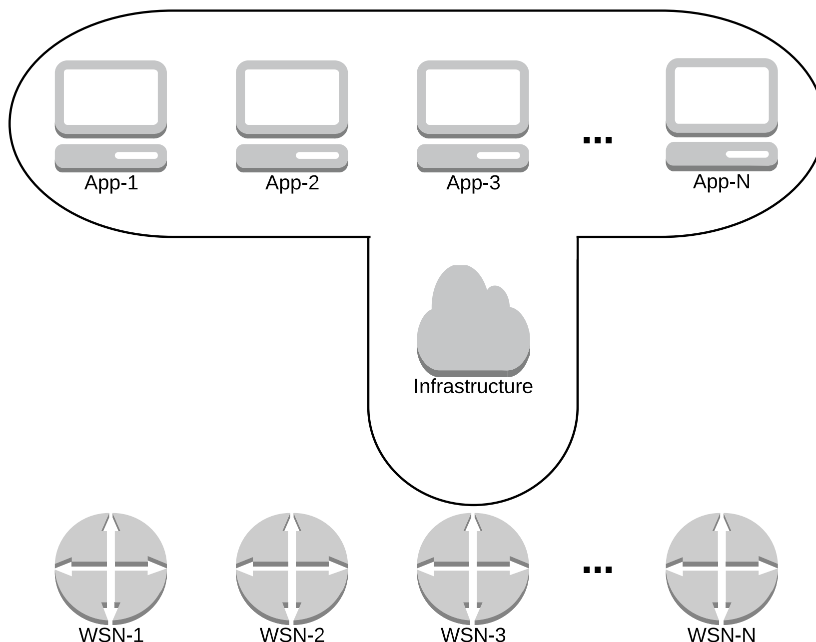


Figure 2.4: Example of a horizontal solution

2.3.4 Current proposals for Middlewares and Frameworks

Although the concept of horizontality is simple and easy to understand, in which it's only necessary to develop a framework that allows to integrate diverse networks of devices and

systems (as sensors and actuators), so that they can efficiently interact between them, the reality is much more adverse. As already referred in section 2.3.2, for a long time, and even nowadays, many of the developed solutions are vertical, focusing in only one immediate solution and not taking into account aspects like interoperability, scalability and evolution (integration of new devices/systems, or even networks, to existent networks).

However, the efforts done in this area have been helping in its evolution, not only with the development and specification of new architectures for the frameworks, but also with the definition of standards that contradict the 'anarchy' imposed by the numerous vertical solutions developed over the years.

Despite the differences amongst the defined architectures, and implementations of these frameworks, one thing that throughout the years has become more or less uniform are the requirements that these frameworks must comply with:

- **Heterogeneity:** represents the variety of networks of devices, or even nodes, that a framework has to support. This variety is caused by the many different existent protocols, technologies, and even manufacturers, that are used by networks and devices for communication.

  Thus, for a framework that has the goal of integrating multiple networks, it's important standardize the access to those networks, independently of the used protocols and/or technology;
- **Scalability:** defines the capacity that a framework exhibits for supporting great amounts of communications in its networks of devices, without a sharp decrease in the overall performance of the infrastructure. Since these frameworks are intended to be used in environments of considerable dimensions (for example, in a network operator domain), with numerous networks and users, it's important to maintain its performance, even when the number of networks and consumers (and respective communications) in the network increase considerably.
- **Models for representing information:** as in most cases the networks are composed by devices of different manufacturers, it's essential to ensure an uniform representation not only of the data and meta-data, but also of the sensors interface (which defines how to access their operations).

  As an example, to show the importance of achieving an uniform representation of data and meta-data, consider that a specific sensor performs a reading and sends the processed data to some specific entity. To enable the entity to process the data, it needs to know how the received data is represented, since only after knowing the representation the entity is capable of decoding that data. Posteriorly, after the data has been decoded, the entity gets to know that the data represent the number 22 (as an example). However, more information is necessary, since the number 22 can be related with temperature, humidity, pressure, among other innumerable possibilities. Thus, meta-data is responsible for providing additional information about the data, like identifying the phenomena (temperature, in this example), which unit (Celsius or Fahrenheit), which sensor performed the reading, etc;
- **Adaptability:** being the frameworks capable of integrating multiple types of devices, many of them mobile/portable, that are constantly entering and leaving the network (and thus could become unreachable), it's necessary that the framework itself provides adaptation mechanisms, that not only allows it to use the existent devices in an efficient way, but also enables it to deal appropriately if one of the devices becomes unreachable;
- **Management:** it's also important that users can perform configurations (manage) in order to reflect their needs. These configurations, depending of authentications and authorizations, can grant the users control of some entities and their respective resources,

allowing, for example, to define Quality of Service (QoS), Quality of Information (QoI), manage the usage of energy, among many other use cases.

Below is a description and a brief analysis of some of the most relevant projects for this area, that meet all of the requirements defined above.

2.3.4.1 IrisNet

IrisNet [22] is introduced not because it presents a very relevant and state of the art architecture, but for being one of the first proposals for integrating networks of sensors in a large scale, thus allowing a more broader and evolutionary vision of the work that has been done in this area for the last years.

The purpose of IrisNet was to allow information (data) coming from multiple heterogeneous networks of sensors, globally distributed, to be reused by numerous applications (denominated by sensing services by the authors) developed by third parties.

Its architecture, in Figure 2.5 is organized in Organization Agents (OAs) and Sensing Agents (SAs), being the OAs the architecture nodes that implement the distributed database that then will store the information coming from the SAs, which in turn are the architecture nodes (sensors) that implement a interface in order to provide uniform access to sensors of different types.

The OAs are just responsible for storing and organizing the information of a specific sensing service. A terminal can run various OAs, also providing failure tolerance. The way that this information is stored in the databases is defined by the developer of the sensing service, through the use of XML Schemas (XSDs), which in the authors perspective allow to adequately represent the data in an hierarchy form. Each OA is divided in diverse terminals (distributed), where each terminal stores a subset of the hierarchy of the OA, and the sensing services then take advantage of the information stored in the OAs using XPATH expressions. IrisNet ensures that the queries done by the sensing services are forwarded through the several machines where the pretended OA is distributed, and afterwards, that the response is also correctly forwarded back to the sensing service, which then can use the obtained information. To ensure that those queries are not done to the distributed OAs, IrisNet recurs to DNS to solve the names in IP addresses.

Sensing services are also responsible for developing executables that are then run in the SAs safe execution environment, developed for this purpose. These executables are named senselets and allow the developers of the sensing services to define how to process the data arriving from the sensors. This way, the processing is done close to the source of information, avoiding to waste unnecessary space by storing data that will not be used. The SAs still allow to execute more than one senselet by sensing service.

In the article are referred a few applications prototypes developed resorting to IrisNet infrastructure. From an application that allows a user to know which is the nearest car park that has an available spot appropriate to his needs, to an application that allows to monitor coast shores in search for anomalies. The main difference between these sensing services, besides their obvious different use cases, is the fact that the developed senselets perform completely different processing. For example, regarding the first case, the video stream provided by the cameras that are placed in the car parking lots is processed by the SAs senselets, in order to check for available spots. In the second case, the SAs senselets are used to gather images captured during 10 minutes, by cameras filming the coast shore, to search for anomalies, being the verifications done in 10 minutes intervals.

Despite of being one of the first proposals that appeared to integrate the scattered networks of sensors, as already referred, IrisNet still presents some relevant functionalities, as

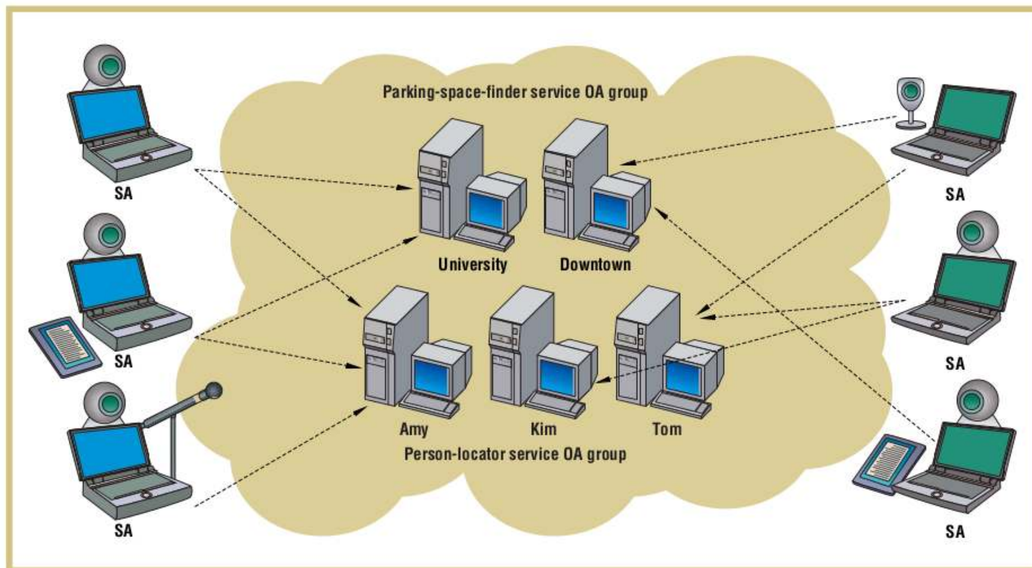


Figure 2.5: IrisNet architecture [22]

is the case of the reutilization of the sensors networks, the possibility of processing the data generated by the sensors, and still, the existence of caches in OAs, which store the results of recently made queries, with the goal of improving the response time of queries that are regularly performed. Furthermore, the fact that the data is stored in a distributed way, not only suggests that there is a good scalability in this architecture, but also provides primary and secondary replicas in order to tolerate failures, as the power down of a machine that contains part of the database, for example.

However, this architecture leaves some key points unanswered, as is the case of multiple sensing services not being able to use the same data, with the objective of decreasing the overall use of bandwidth, as well as the data stored in the database, and that the definition of one database for each sensing service might lead to the existence of very similar databases, while one database could be shared among the different sensing services, in such cases.

2.3.4.2 Hourglass

The goal of Hourglass [23] is to provide an infrastructure that not only allows the integration of different networks of sensors, but also allows to deal with aspects like their discovery, addressing, routing and packet aggregation, thus enabling the easy utilization of the services provided by these networks by the applications. That infrastructure receives the name of Data Collection Network (DCN) by the authors, and it's based in circuits to ensure that the data generated in the sensor networks, producers, is delivered to the applications, consumers, which have requested them. Through those circuits, operations can be made in data, performed by the so called intermediary services. This way, from a Hourglass perspective, circuits are just abstract connections that interconnect the producers of the information to the consumers, where may exist, eventually, as already referred, intermediary services that process the data that travel those circuits.

The services provided in this infrastructure are organized in Services Providers (SPs), and each SP, belonging only to an administrative domain, can be composed by one or more nodes of the Hourglass. The SPs, upon their creation, are obliged to follow the minimum requisites, i.e., the existence of a circuit manager and a registry, responsible for the creation

of the circuit, from the producers to the consumers and for the localization (in a distributed way) of the terminals responsible for the services, respectively. The registry is responsible for storing the existent circuits, and thus allowing that the circuit manager can optimize services that are active/running. The local services of a SP are registered by announcements, which contain several information that enables their identification, as the topic and the predicate, necessary to identify the primary topic to which the service belongs to, and the more specific predicate, which allows an unambiguously identification.

In a general way, in Hourglass, an application that wants to consume information from one or more producers of information, first sends an Hourglass Circuit Description Language (HCDL), which is just a query sent to the circuit manager in order to compose the requested circuit. The circuit composed by the circuit manager, in turn, recurs to the registry to localize the terminals that provide the services requested by the application. After knowing the services location, the circuit manager sends the information to those services to allow them to compose the circuit between them. Finally, after the acknowledgement that the circuit has been correctly established, the circuit manager indicates the entities of the circuit to start sending the data. An example of a established Hourglass circuit may be seen in Figure 2.6.

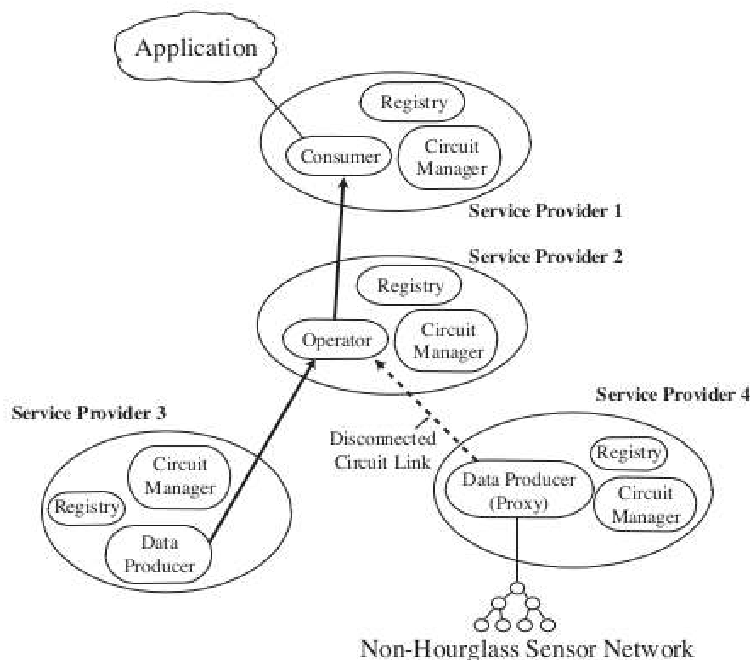


Figure 2.6: Example of an Hourglass system with one realized circuit [23]

Depending on the implementation of each SP, Hourglass allows them to resist to intermittent communication, through the use of a buffer that stores the information that should be sent during the connection break, sending the information after the connection has been reestablished. Another service that can increase the quality of the data provided to the applications, just like the buffer service, is the filter service, that allows the data consumers to restrain the data close to the source, which also allows to decrease the overall used bandwidth. Furthermore, it's also referred by the authors the persistence service, which allows applications to have a history of the data that has passed in the circuit. Lastly, Hourglass still allows to define QoS upon the implementation of the service, which will be taken into account by the circuit manager, upon the circuit composition.

One of the problems inherent to Hourglass, as announced in [24], and that can create problems of scalability, is not only the fact that the circuits are established only upon request of each application that intends to use the service, but also because the state as to be maintained in each of the nodes of the circuit. Although it's adequate for applications that want streams of long duration, it may not be appropriate for streams of small duration.

2.3.4.3 USN

This proposal [25] focuses primarily in providing a uniform access to networks of sensors rather different from each other, thus considered heterogeneous, recurring to the IP Multimedia Subsystem (IMS) platform, which, according to the authors, provides many benefits by enabling to solve, or at least trying to solve, some of the challenges inherent to the development of these scalable platforms of globally distributed WSN. The term Ubiquitous Sensor Network (USN) was used for the first time in a report of ITU-T to describe a globally distributed network of sensors that in the future may become ubiquitous, which means, being present everywhere (omnipresence).

The architecture of the USN platform was developed following an horizontal model, meaning that the networks of sensors and the applications (services) could evolve independently. The platform contains four layers, which are, bottom-up, the network of sensors, the USN-Gateway, the IMS core and the services layer.

The USN-Gateway is responsible primarily for two procedures. The first one is to standardize the communications with the sensors, making it homogeneous, and thus allowing that the communication of the IMS core with the sensors is independent of the type of sensors. This standardization is possible because the USN-Gateway implements the Session Initiation Protocol (SIP) protocol, which makes it able to convert the messages that are sent between the IMS core and the network of sensors. The second procedure is to provide a way to represent homogeneously the data coming from the different networks of sensors. This homogeneity is achieved thanks to the use of already existent standards, as is the case of Sensor Model Language (SensorML) and Observations and Measurements (O&M), used not only for the data representation, but also for the provision of additional information regarding the data itself (like meta-information).

The USN-Enabler is defined as being an interface that allows and facilitates the creation of services, where multiple services can use multiple networks of sensors, thus reusing them and increasing the return of the investment made initially. This component of the architecture provides, through existent standards and recommendations, basic functionalities, like the registry of sensors and their respective publication, management and discovery, subscriptions and notifications, filtering and processing of data, and storage of the data coming from the sensors.

This architecture is important for the fact that it uses many existent standards, as already said, for implementing its functionalities. By using the IMS platform, USN allows this architecture to be adaptable to the next generation networks, thus avoiding major posterior adaptations to enable its integration. Furthermore, another benefit of using the IMS platform is that it already provides important functionalities, like AAA (Authentication, Authorization and Accounting), that allows to ensure more security in the accesses made to the network of sensors.

However, USN's architecture presents a point that cannot pass unnoticed, which is its centralization around the USN-Enablers. Since the USN-Enablers are responsible for dealing with multiple services and multiple networks of sensors, it would be interesting to check what is the scalability that this architecture provides when faced with large networks of sensors, with thousands of sensors. Only then we would have a notion if the USN-Enablers would be

capable of dealing with the thousands (or even more) requests and information that would come from the networks of sensors.

2.3.4.4 Global Sensor Networks

GSN [26], from Global Sensor Networks, is a middleware platform that proposes to integrate, in a rapid and flexible way, heterogeneous networks of sensors. Besides that, GSN also proposes to overcome the problems inherent to networks of sensors, and the integration of such geographically disperse networks. Some of those problems are, for example, the processing, the storage, the querying, and the publication of data.

In the architecture proposed by GSN, the virtual sensors, as called by the authors, are its key component, allowing to abstract the access to the heterogeneous physical sensors. These virtual sensors abstraction are also the services that are provided and managed by the GSN, being used by external applications/services through the middleware.

A virtual sensor can have multiple input streams, that can come from a physical sensor, or even another virtual sensor, and just one output stream. To implement and use a virtual sensor it's necessary to define its specifications, which contains all the necessary information, like the meta-data information used in the discovery process, the streams structure that the virtual sensor consumes and produces, the type of processing performed in the input stream (or streams) by the virtual sensor, specified in SQL, and even functional properties, as persistence, error treatment, physical implementation and management of the life-cycle. This specification of the virtual sensors is made resorting to a descriptor, which facilitates and accelerates its implementation in the GSN.

The architecture is based in GSN containers, Figure 2.7, which are responsible for managing one or more virtual sensors, in a concurrent way. These containers are deployed in common terminals, like a desktop computer, and communicate, for example, for discovering and access the virtual sensors, following a Peer-to-Peer (P2P) model. The GSN container is then divided in a set of modules that implement the already referred functionalities, like the Virtual Sensor Manager (VSM), the Life Cycle Manager (LCM), the Input Stream Manager (ISM), and the Query Manager (QM).

The names given to each modules are enlightening by themselves, and together are responsible for managing the remote access and interactions with the networks of sensors, the security, the persistence, the data filtering, the queries, the concurrence and the resources that allow the processing of the data in real-time. Other modules are responsible for allowing third parties to access the provided services, through web services, for example.

The GSN architecture allows to take advantage of some interesting functionalities that are usually desired for environments that integrate networks of sensors, as the example of access control, that allows to ensure the integrity and the confidentiality of the data produced by the sensors.

Furthermore, although GSN allows the implementation of new services through the specification of virtual sensors, using a XML file, which is certainly simpler than developing low level executables that run directly in the physical sensors, GSN authors thought that, to also facilitate the implementation of new physical sensors and their recognition, by avoid as much human intervention as possible, it would be useful to also include the IEEE 1451 standard.

This standard allows the compatible sensors to be detected, configured and calibrated automatically. To be compatible with the IEEE 1451 standard sensors only need to have a description of its characteristics and functionalities stored under the form of Transducer Electronic Data Sheets (TEDS).

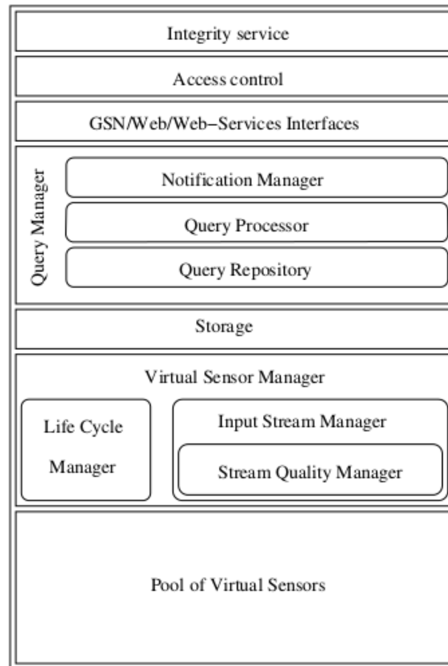


Figure 2.7: GSN container architecture [26]

As already stated above, the communication between the GSN nodes follow a P2P model, which suggests that this architecture might be scalable, being able to deal with large quantities of data streaming, and networks of sensors.

However, by resorting to SQL, this architecture might not allow a wide definition of processing and aggregation of the data coming from the sensors, [24]. Also, GSN does not allow to optimize streams by using the same virtual sensor(s) for different purposes (although similar).

At last, no references are made to mechanisms as buffering, for example, that can prevent temporary disconnections, which can avoid the loss of data in environments where the connection between the sensors and the rest of the infrastructure is weak, as happens when using wireless for communication.

2.3.4.5 WebDust

The primary objective of WebDust [27] is to allow heterogeneous devices to operate in the same network, allowing to control and manage those networks, through mechanisms that enable their visualization and enable that enable their manipulation.

This platform architecture follows a P2P model and it's based in three sub-domains:

- P2P network, composed by applications that run on terminals, like desktop computers, and which actuate as peers in the distributed environment;
- Nano-Peers, sensors that are clustered together in networks of sensors and mostly communicate through wireless;
- Gateway-Peers, responsible for abstracting and allowing communication between the Nano-Peers and the P2P network.

The key component of the WebDust architecture is the Gateway-Peers devices, which appear in the P2P network as participants with monitoring and sensing capabilities. These

devices act as stations that control the networks of sensors connected to them, thus being responsible for collecting all the data coming from them, as well as forwarding the queries that have sensors as its destination. The data that comes from these networks of sensors is stored in a relational database, organized in three categories - by device, by query and by state of the sensors and actuators of a specific device.

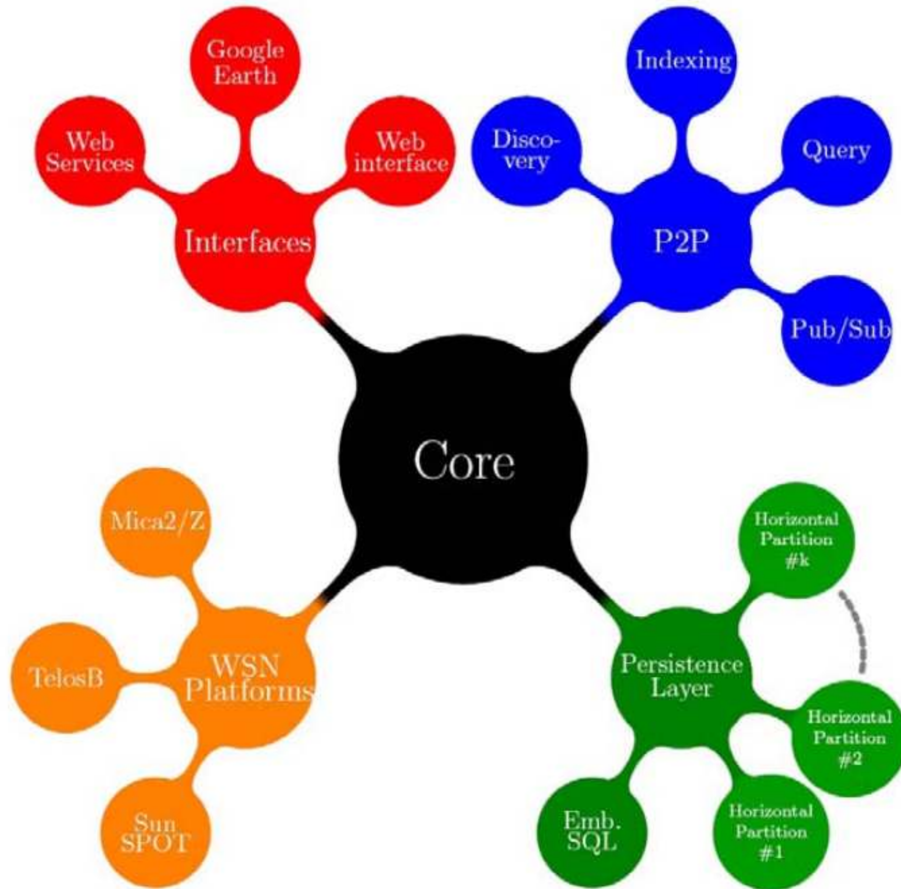


Figure 2.8: Overview of WebDust's architecture [27]

The structure of the P2P network peers follows a two layer architecture, being one the outer layer, and the other the inner layer. The latter, the peers inner layer is characterized for maintaining the information about the networks of sensors (its physical configuration, for example), the devices's specifications, and the results of the performed queries. This layer is referred as the core, since it's where is defined a minimum set of functionalities that will be then used by the high level services to interact with the networks of sensors. On another hand, it's in the peers outer layer that are executed the high level services, recurring to the functionalities provided by the inner layer. These layer services can be distinguished in 4 categories, the P2P services, the WSN platform services, the persistence services, and the web services and other interfaces. In general, for the P2P services, JXTA was used for the implementation of an environment that follows the P2P model, whereas for the WSN platform services are provided interfaces in order to support the low level functionalities offered by the sensors. Furthermore, for the persistence services, in order to allow the exterior interaction with these services, Hibernate was used, a Java Persistence API (JPA) implementation, and finally, regarding the web services and other interfaces, these are defined so that third party applications can take advantage of the provided services.

WebDust provides some relevant services that allow external applications to take advantage of the same. The buffering service is a good example, that allows to appease the data loss caused by connection failures that are usual in wireless networks. The data is stored by the peers present in the P2P network, and then resent when the connection is reestablished. However, not giving less importance to this service, this is only supported among the peers of the P2P network. WebDust still allows to define complex queries, like aggregating data of a set of sensors, aggregating data of a specific period of time, or even activate actuators, thus allowing to obtain a variety of information coming from a diverse set of sensors and devices. It's also possible to measure the performance and the state, not only of the P2P network, but also of the networks of sensors, allowing the developer of a specific application to access this service to configure the network and the sensors, in order to better adapt it to the application, increasing the overall performance. Still, regarding the previous service, functionalities that support management, mobility control, topology control, energy and time synchronization are also provided. Finally, a last relevant service allows the management of multiple networks of sensors, each one with its own control center (Gateway-Peer). This service is implemented at the cost of virtual sensor networks, that allow to abstract the real topology of the networks and thus, the sensors and the devices are controlled as if they were in the same network.

However, this architecture shows a few deficiencies, because although it's implicit that WebDust provides concurrent access by the applications, and that the queries executed in a distributed way are optimized in terms of resources usage and time, the truth is that there is no reference to mechanisms that allow to arbitrate the access to resources. The fact that the architecture is so dependent of the Gateway-Peers, which stores the information coming from the networks of sensors in a database, can raise scalability problems in case the number of networks of sensors and applications increases.

As a final point, it is stated by the authors that WebDust was implemented to add new sensors to the infrastructure in a easy and efficient way. However, despite the easiness, is still necessary to install specific drivers in each sensor, in order to enable the translation of messages exchanged between the Gateway-Peers and the Nano-Peers. This 'a priori' programming and configuration of the sensors undoes any easiness that could exist regarding their installation in the WebDust infrastructure.

2.3.4.6 Sharing Sensor Networks

In [28], the authors propose a way for presenting and accessing different services provided by different networks of sensors, based on the Universal Plug and Play (UPnP) standard. The primary objective, just like in the other platforms, is to allow services utilization by applications developed by third parties, without the need to know what type of networks of sensors are underneath.

The presented architecture can be divided in three main components, the WSN, the P2P bridges, and the P2P UPnP Gateway. While the P2P bridges are responsible for making the connection between the P2P network and the WSN, the P2P network is responsible for connecting all the WSN and P2P UPnP Gateways, which in turn provide the interfaces that enable applications to interact with the provided services.

Therefore, the P2P bridges receive the data transmitted by the WSN, and analyse the attributes of each node of the WSN. After that analysis, the P2P bridges announce the P2P substrate that a node with a particular set of attributes, including its position, for example, is available. The node, thus, stays associated to those attributes in the P2P network (knowing that it can be accessed by the P2P bridges that announced it), that uses Distributed Hash Tables (DHTs) to store that information. Besides that, the P2P bridges also have the function

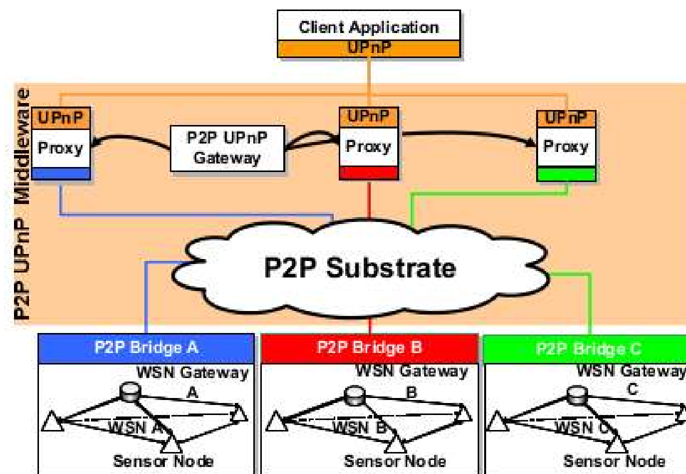


Figure 2.9: Services from remote sensor networks [28]

of passing the data from one side to the other, that is, from the WSN to the P2P network and vice-versa.

In the P2P UPnP Gateways, between the P2P substrate and the client applications, the service providers supply service descriptions, indicating what type of services are supplied by the WSN. These descriptions are then used by the Gateways to instantiate proxies, one for each service defined in the descriptions. The proxies, after instantiated, try immediately locate the pretended services, resorting to the search service offered by the P2P network. After that, the application can use the requested service, through remote invocations (Remote Procedure Call, RPC), being the proxy responsible for doing the mediation between the application and the service provided by the WSN.

The fact that this architecture is based in a P2P model and uses Distributed Hash Tables (DHTs) to store the associations between the sensors and their respective attributes, suggests that it has a good scalability, not having an overall decrease of performance when the usage of WSN and applications that interact with the platform increases. The usage of the UPnP standard also makes it easy to add both new P2P UPnP Gateways and applications to the infrastructure, as long as they also support UPnP. The applications can even subscribe to data coming from specific service, without establishing an explicit connection. Only then it's possible for the applications to be notified of events that only occur sporadically, like, for example, being notified when the temperature of a particular place, that is being monitored by a WSN, reaches a certain pre-defined limit. Otherwise, the applications would have to be frequently making requests to the WSN (pooling), or the WSN would have to be constantly sending the temperature (streamming), which would consume bandwidth unnecessarily.

However, regarding some points, this proposal becomes a little short. One of those points is relatively to the definition of more complex queries, since the authors don't refer any ways to aggregate data coming from diverse WSN, or to aggregate data from a specific period of time, thus not giving a lot of flexibility to manipulate the data drawn from the WSN. Another functionality that should have been taken more into account is the processing being done closer to the source of data, that is, the WSN, which would allow, for example, filtering data. Still regarding data optimization, it's not made any reference either about multiple applications being able to use the same stream of data, or about optimizing the used resources. However, this point can be important regarding the infrastructure scalability, because although being based in a P2P model, the fact that no resource and data optimization exists means that, an increase of the number of networks of sensors and applications that are

integrated in the infrastructure can cause a decrease on its overall performance.

2.3.4.7 Service-oriented framework for IoT

The authors of this proposal [29] focus in providing a multi-layer architecture that allows not only an efficient and safe interaction between the Small Programmable Objects (SPOs) and the Web (Internet), but also the management of the implementation, maintenance and operation of those SPOs. The SPOs are defined as sensors that provide Web services to expose their functionalities. In order to provide the Web services, each sensor implements Senselets, which is a minimalistic adaption of Java Servlets, thus allowing resource limited devices to resort to Web services. Senselets are developed and deployed in the SPOs by developers who wish to take advantage of a functionality.

The architecture was implemented with the following layer hierarchy, the SPOs are placed in the lowest layer, the controller in the intermediate layer, while the clients are located in the highest layer.

In SPOs, apart from the Senselets, where each one implements a specific functionality, there are also mechanisms that allow them to interact with the remaining infrastructure, as other SPOs, and even mechanisms that allow their discovery. Additionally, it's implemented in each SPO a 'nano' HTTP Server, as denominated by the authors, whose functionality is to parse the HTTP requests and identify which Senselet must be executed in order to satisfy the request. Each sensor is registered in a gateway and the gateways are responsible for allowing the communication between the higher layers, the controller, and the sensors. To achieve that, the gateways implement two interfaces, one for communication with the SPOs, through the Senselets, and other for communication with the controller, using Java Servlets.

The controller is the core of the architecture, where are offered most of the functionalities, being exposed through a Web service API that can be invoked by the clients. The controller is implemented in a modular way, as shown in Figure 2.10, and is divided in three modules:

- WSN Controller: provides the Web services that allow to perform administration operations in each network of SPOs, like, for example, turning on/off SPOs, define its virtual topology, or even deploy, run or stop Senselets execution.
- Routing: responsible for allowing the formation of organizations through the clustering of networks of SPOs, and also for implementing the virtual topologies defined among the networks of SPOs.
- Proxy: makes the publication and exposure of the functionalities offered by the Senselets, which are located in the SPOs, to the clients side.

It's still in the Controller that are also defined, recurring to the WiseML standard, network topologies, as well as information about the Senselets deployed in the SPOs.

In the higher layer, as already referred, is where the clients are, and they have two options in order to connect. Either through the Website, which offers an interface that allows to control and monitor the system in a global way, or through dedicated applications, which use the services offered by the Controller in order to directly interact with the sensors.

The primary objective of this architecture focus in a very important point when dealing with multiple networks of sensors, that is, the management and monitoring of those sensors. Furthermore, since that interaction is done through Web services, it's achieved in this architecture a great interoperability with the actual Internet, not having the need of using/adapting protocols/stacks, or even standards. Just as an example, it's possible to interact with the SPOs, or the overall infrastructure, just by using a terminal that is connected

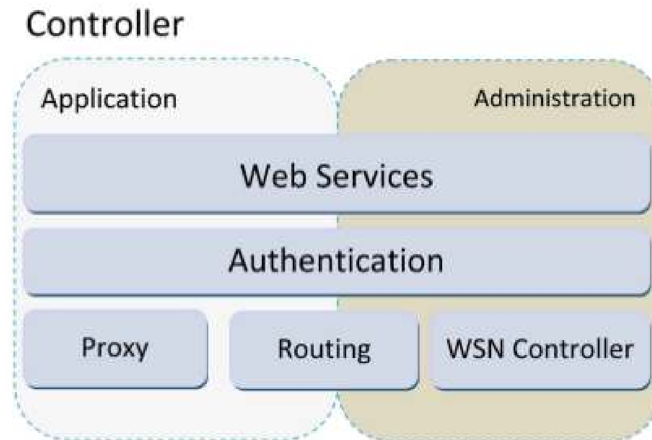


Figure 2.10: Controller stack [29]

to the Internet and runs a browser. It's also possible to discover new SPOs, and their respective registry in the gateways, in a almost automatic way, which confers a good and desirable simplicity while interacting with the infrastructure. Another advantage is the definition of virtual network topologies, which allow to compose a single network, recurring to various SPOs networks, thus enabling to configure networks according to one's needs.

However, the routing performed between networks is done recurring to the Controller, and being the Controller a central entity, this can become a problem with the raise of the number of SPOs, and the existence of virtual networks, since all the information will have to pass through this central unit.

To finalize, although the Web services provide a good interoperable solution to the architecture, as already stated, and the authors even refer that the use of the Web services in constrained resources devices, like the sensors, has been already examined, the truth is that the usage of Web services in the sensors requires from them too much resources, and no experiment done in a reasonable scale, in order to verify the behaviour of the sensors, is referred by the authors. This way, the Web services, despite the provided interoperability, can bring an overall decrease of performance to these already resource limited devices.

2.3.4.8 Practical realization of an IoT

The proposal in [30] describes an architecture that provides protocols for accessing basic services, as environment monitoring, for example, where WSN are located. Furthermore, the fact that the WSN nodes support the 6LoWPAN standard, means that they can be easily addressable from anywhere in the actual Internet, provided that IPv6 is supported. This factor brings additional advantages, since it not only allows applications to interact with the services provided by the infrastructure in a much simpler way, leading to a much rapid development of applications, as it also allows the integration of Web services in the nodes, thus raising the interoperability of the provided services.

Therefore, in this proposal, WSN nodes were installed with an implementation of Binary Web Services (BWS), developed by the authors for this purpose. In BWS, the resources are seen from a REST perspective, but the messages that are exchanged, in order to save space, are binary coded, using Efficient XML Interchange (EXI). Since the nodes located in the WSN are devices very limited in terms of resources, implementation tries to minimize resource usage. A good example of that minimization is the fact that the BWS module is

implemented in order to provide both the functionalities of the client and the server, being that both support simultaneous (concurrent) requests, that is, the clients can make concurrent communications with multiple servers, while the servers can answer to multiple parallel requests coming from different clients. Additionally, in this proposal, are also used interfaces compliant with Resource Publication Interface (RPI) standard, for the publication of resources, as well as interfaces compliant with the Resource Access Interface (RAI) standard, for accessing the resources.

The authors, throughout the article, also refer the importance of maintaining network operations efficient, and for that it's necessary that the architecture, which retains the functionalities provided by the network, is scalable and easily extensible. Thus, three types of nodes are distinguished, which allowed to create an efficient architecture, based in a resources oriented paradigm:

- Base Station Node (BSN): play the role of a gateway, making the interaction between the Internet and the WSN possible;
- Mobile Node (MN): don't have any association to BSNs or even to WSN, being only nodes that establish a temporary connection with the infrastructure in order to obtain the needed data;
- Special Nodes (SN): are the nodes responsible for executing the services that are provided by the infrastructure and requested by the client applications.

Although the architecture described in the article doesn't give much details, and leaves some points unanswered, it provides the notion of which protocols and standards should be used to integrate the WSN, and furthermore, how they interact. However, some of those points are necessary to be defined, because they become crucial not only for the development of the architecture, but also for the posterior usage of the infrastructure. One of these points is the fact of not being defined, or made any reference by the authors, of how applications are supposed to discover services. Furthermore, it's also not referred how to make optimization of resources, like, for example, the same stream of data being shared among several applications, avoiding the redundancy that will result in an unnecessary bandwidth consumption.

2.3.4.9 e-SENSE

E-SENSE [31] provides an architecture that integrates itself efficiently in mobile communication systems, like the IMS, and whose main purpose is to allow the integration of WSN.

It was an integrated project supported by the sixth European Framework Programme (FP6), whose ambition, besides of creating a platform for integrating WSN, was to shorten the gap between Europe and the rest of the world (North America and Asia), regarding the research done at the level of WSN, being the former lagging behind. Thus, during its two years life time, e-SENSE contributed to important standardization efforts, as is the case of the ZigBee Alliance and the IEEE 802.15.4, producing quality specifications that can become standards, and reached its primary objective: creating one of the first platforms that could integrate WSN worldwide.

The article refers that the e-SENSE architecture was quite influenced by the ZigBee specification, mainly for being a technology, in the authors opinion, that will be incorporated in many devices. However, despite being influenced by ZigBee, e-SENSE is not limited to ZigBee. It provides substantial improvements and evolutions, allowing enough flexibility of the underlying WSN to correspond to the different applications needs. Furthermore, e-SENSE was tailored from the beginning to be used in smart environments, like the integration

of WSN in cars, machines, buildings, or even ourselves, humans, through our clothing, for example, forming the IoT.

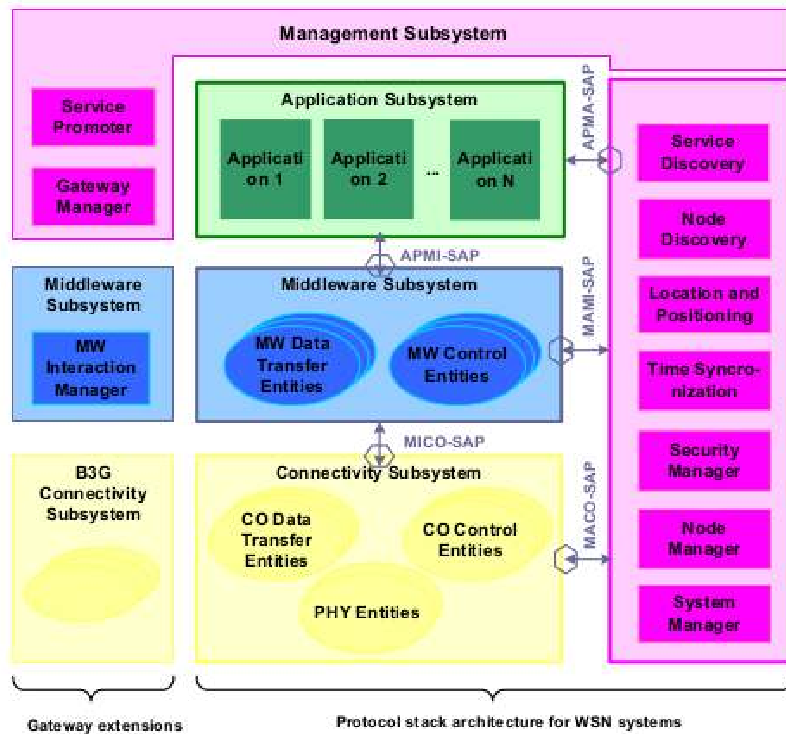


Figure 2.11: e-SENSE protocol stack architecture [31]

The proposed architecture can be divided in four main layers, or sub-systems, as called by the authors, and can be seen in Figure 2.11. Each sub-system can communicate with other sub-systems through their services, which can be invoked via the Service Access Points (SAPs). The communication between the sub-systems is not done in a strict way, where a specific sub-system could just communicate with the sub-system beneath, but rather in a more flexible way, meaning that a sub-system can communicate with sub-systems that are not immediately below/above it (cross-layer).

Thus, the lowest-level sub-system is named Connectivity (CO) and is responsible for functionalities like, composition of the network that will connect the WSN nodes, allowing new nodes to join the networks, exchange of data among the various nodes, and even control nodes's access to the communication medium. Furthermore, given the flexibility in the communication between the sub-systems, the e-SENSE allows that, depending on the requisites of an application, the management sub-system (Management, MA) can define the protocols parameters and the internal states, through CO's SAP (MACO-SAP), which will specify certain aspects of sub-system CO operations, as, for example, the routing of data (QoS).

On the top of CO there is the middleware (MI). The purpose of the MI is to allow the information originated in the WSN sensors to be processed in a distributed way. This way, MI supports that the data coming from applications can be transfered in two ways, node oriented, or data oriented. Regarding the former, the WSN nodes, where the information originates, sends the information to a node or a set of nodes, through an address. Moreover, in the latter, the data is transmitted making use of a publish/subscribe paradigm, in which a node or a set of nodes subscribe to a specific type of information, and posteriorly, when that information is available, it's delivered to them.

On the other hand, the management sub-system has the functionality of configuring and initializing the CO and MI sub-systems, according to the requisites provided by applications. The configurations and the initializations are done through the SAPs that the MA has with the CO sub-system (MACO-SAP) and with the MI sub-system (MAMI-SAP). However, the MA is not responsible for only providing those functionalities, but it's also responsible for others like, the discovery of nodes and services, that allow to find specific services and/or sensors in the network, the location and position, that allows to know the absolute and/or relative position of a specific node in space, the synchronization of time, the management of the nodes and the system, that ensures that the nodes and the system operate correctly, and even owns a database of information (parameters and attributes) regarding the sub-systems.

At last, as the highest-level sub-system is the application (AP), composed by the client applications, that resort to the MI sub-system (APMI-SAP) functionalities to send/receive data to/from the WSN, and to the MA (APMA-SAP) functionalities to configure the MI and the CO according to their needs.

As said initially, the e-SENSE was architected in order to integrate easily with mobile communication systems, particularly, with IMS. Thus, to facilitate the integration of the e-SENSE system with the IMS, gateway extensions are used, which add functionalities to two of the sub-systems, the CO and the MI. These gateways, through the use of the Session Initiation Protocol (SIP), and by providing interfaces for interaction, allow the communication of the e-SENSE systems, that contains the WSN, with the domain of the IMS system. This makes it possible that each e-SENSE system registers its availability in the IMS platform, and that each system registers the services it offers in the e-SENSE Service Enabler (SE), in order to make them accessible by the applications hosted in the Application Servers (AS), located in the IMS domain.

The objective of allowing e-SENSE systems to be integrated in IMS domains is so that IMS applications can benefit from the information coming from the sensors based on the context, that is, providing a specific context requisite, like the localization and the activity state (running, walking by foot, walking in the car, among others), the applications can receive more relevant information (customized) from the sensors. Otherwise, would have to be the applications to support the task of making multiple requests to different WSN, processing the obtained results, and even having to know which WSN exist, and what services they provide.

e-SENSE SE, which was already referred above, is responsible for the following role, that is, after receiving a contextual request (by an application), it is responsible for decomposing the request in several requests of lower complexity, and identify which one of the e-SENSE systems contains the WSN that can satisfy those requests. Then, upon the reception of the requests, the e-SENSE SE proceeds to processing the data, in order to compose the response that will be sent to the application that made the contextual request. This is the only point of contact between the applications that make requests of contextual information and the several e-SENSE systems, being the e-SENSE SE responsible for maintaining the information updated, about the available services and in which e-SENSE systems they are available.

The architecture presented by e-Sense provides a good way of integrating WSN, providing functionalities like the discovery of nodes and services, quality of service, or even the management of the nodes and system in general. The fact that the architecture has been developed in order to enable their incorporation in mobile communication systems that will be used in the future, like the IMS, is undoubtedly an asset, thus enabling their usage in future solutions without the need for big adaptations. Furthermore, together with the IMS integration, the project even refers a way to provide to applications, present in the IMS domain, contextual information, resorting to a special Service Enable to make the processing of the requests and responses, the e-SENSE SE.

However, there are still some aspects that must be taken into account regarding the e-SENSE systems, when integrated with the IMS, such as the solution's scalability, since all the information coming from the WSN and the applications pass through the e-SENSE Service Enablers, which can decrease the overall performance of the infrastructure. Since all of these mobile communication systems have to allow the delivery of data to a large number of users, scalability is an important factor, and the fact that in this case it may affect the overall performance, can lead to a large number of users being harmed. A solution for this problem, even though it is not mentioned by the authors, may pass through the usage of several SEs, increasing, however, the complexity of the interactions. On another hand, in the project there is also no reference made to optimizations of streams of data, as for example, the fact that multiple applications can use the same service, in which case all of those applications could be served by only one stream of data, rather than generating a stream of data to each application, hence saving bandwidth.

2.3.4.10 SENSEI

Supported by the seventh European Framework Programme (FP7), SENSEI [32] was an integrated European project that followed e-SENSE. The main motivations that lead to the proposal of SENSEI were:

- The importance of applications and services becoming increasingly more intelligent, through mechanisms that allow the awareness of the context;
- The lack of open frameworks that allow the easy integration of networks of sensors and actuators causes vertical systems to be used, that then prevent reusing the information for new applications;
- Allow Europe continue to innovate in an active way, through the investigation of areas related to IoT and WSN.

Although SENSEI follows up the e-SENSE project, and thus reuses some of its concepts, the concept of SENSEI was developed with greater ambitions, intending to innovate and shape IoT through the objectives defined below.

According to SENSEI, the world that we live in can be divided in the real world, and the digital world. The real world is composed by the physical world that we live in and its instruments, like the sensors and actuators organized in networks, that allow us to interact and monitor physical entities that we are interested in, like people, buildings, vehicles, among many others. The digital world, in another hand, contains representations of the physical world, like, for example, the resources (that represent the sensors and the actuators), or the users of the resources (that represent the real people or applications that interact with the resources), thus allowing to map the real world into the digital world.

The main objective of this project, just like it was in project e-SENSE, is to provide the necessary foundations for an easy integration of the networks of sensors and actuators in a global infrastructure, thus allowing the communication between the real and the digital worlds, that is, between users and/or applications and the available resources.

However, the SENSEI project goes even further, and having in consideration the socioeconomics and business aspects that the infrastructure can generate, defined as the SENSEI community, differentiates the resource suppliers (sensors and actuators), the service suppliers (that make use of the sensors and actuators), the SENSEI framework suppliers (since the framework can be supplied by several entities), and the users (that make use of the services), assigning them roles.

Furthermore, the SENSEI project also has the objective of creating an European testbed, that, with the cooperation of several project partners, will be equipped with several networks of sensors and actuators geographically scattered, allowing to test in greater scales other platforms like this.

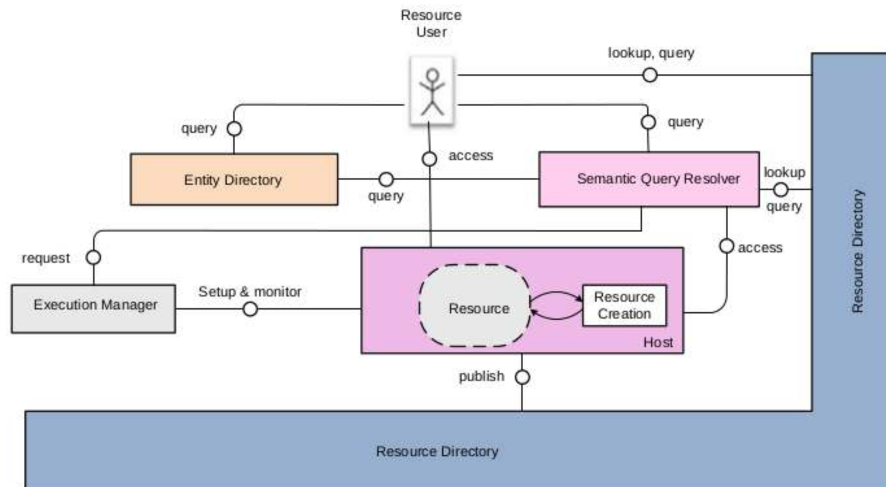


Figure 2.12: SENSEI support services [32]

With the purpose of allowing a user to discover which information/services are available, SENSEI provides what the authors call of rendez-vous functionality. This functionality is divided in two components (Figure 2.12), the Resources Directory (RD), that keeps the descriptions of the provided resources, and the Entity Directory (ED), that keeps the associations between the resources that provide the information and the properties of the entities of interest (which represent places, people, among many others). This means that the RD works like a low level component, allowing users to discover which resources satisfy their request, through the usage of key words, like temperature, humidity, pressure, etc, for the correspondence with the resources descriptions. In another hand, the ED works in a higher level, allowing users to find the resources through the properties of the entities that are associated to them, like city names, people names, etc.

One of the initial objectives of SENSEI was to provide context awareness mechanisms to applications, which is not possible with the simple rendez-vouz functionality described above. With that purpose, a more advanced rendez-vouz functionality was defined, denominated Semantic Query Resolver (SQR), in Figure 2.12, which takes advantage of already existent components, like the RD and the ED, and still allows the composition of services and the attribution of resources to multiple resources. This way, if a user wants to make a more complex request to SENSEI (using a declarative language), that request is decomposed by the SQR, which then will either identify the necessary resources to satisfy the request, or build an execution plan, in case the user's request involves interaction between the resources.

Still regarding the interaction between users and resources, SENSEI defines two types of resources, the so-called one-shot interactions, which are requests of only one response, and the long duration interactions, where the resources can send multiple responses over time, as notifications, for example. For supporting both of the interactions, a component was defined, named Execution Manager (EM), in Figure 2.12, which is responsible for receiving the SQR requests (the execution plans, for example), executing the requests in the resources, and return the results. Thus, regarding the one-shot interactions, the EM only acquires the

requested information an returns it to the SQR, which will then be returned to the user that made the request. However, regarding the long duration interactions, to avoid overloading the SQR, which would otherwise need to support all the streams of data between the users and the resources, the EM is responsible for initiating sessions between the respective users and resources, that will then be used to transport the information.

SENSEI also offers services that allow, not only community management, through the management of its entities, but also to control the access (AAA) and give privacy to users, being the latter a must in order to ensure confidentiality among the actions performed by the users of the platform.

Given that the platform enables the integration of multiple networks of sensors and actuators (resources), it's impossible to think that all these resources use the same protocols and technologies. This way, SENSEI standardizes the heterogeneity through the description of resources, which includes information like, the ID and the name of the resource, a set of tags that identify the capacity of the resource, a semantic description of the information and the operations that can be performed by the resource, and even a syntactic description of the interfaces (through a standard like Web Service Definition Language, WSDL) offered by the resource.

Regarding the information obtained by the resources, and in order to improve its representation, SENSEI decided to divide it in three layers:

- Original information: is defined as containing just the values obtained by the resources, with no modification made to them;
- Information O&M: provides meta-information regarding the original information, thus knowing what is the meaning of the values;
- Contextual Information: provides the context to the obtained information, thus allowing to identify what does the information regards to (a car, a person, a room, for example) and what is it's quality.

Besides the services of community management, SENSEI still allows that the users, depending on their status (operator or administrator of a network of sensors and actuators), can manage not only the networks of sensors and actuators, but also the other components of the platform. This management is useful since it enables, for example, to modify the software of some components of the platform, through updates, planning or even monitoring of specific parts of the platform. To facilitate the access of the users to these operations, SENSEI defined special resources, named management resources, which interact with users as normal resources do. The only difference from the management resources to the normal resources is that the former has to provide management interfaces that allows users, after identified, to proceed with the management of the resource.

The SENSEI project can be considered an ambitious project, resorting to various specifications and standards, in order to implement its objectives, providing scalability in the defined architecture and even good characteristics of integration and management. Furthermore, the fact that the SENSEI had also into account not only the definition of a socioeconomic and business model, but also of a community, in order to split domains (responsibilities), suggests a great evolution, since it stimulates the deployment of WSN testing infrastructures.

2.3.4.11 Summary

This section aims at providing a very brief summary of the platforms evaluated in the previous sections. As such, the following table 2.2 summarizes the key characteristics of each one of those platforms, focusing primarily in sensors and their integration in the platforms.

Name	Architecture	Sensors abstraction	Sensors communications	Main advantages
IrisNet	Distributed horizontal model	Sensors implement an interface	Uses XML Schemas	• Supports reutilization of sensor networks • Allows to process data closer to the sources
Hourglass	Horizontal model	Intermediary Proxy	Uses pre-established circuits	• Supports QoS • Offers buffering functionality
USN	Horizontal model	USN-Enabler maps SIP messages to WSN	Uses SensorML and O&M	• Contemplates IMS integration
GSN	P2P horizontal model	Virtual sensors	N/A	• Supports IEEE 1451 standard for discovering sensors • Offers access-control functionality
WebDust	P2P horizontal model	Sensors implement specific drivers	N/A	• Supports complex queries
Sharing Sensor Networks	UPnP horizontal model	P2P bridges	N/A	• Uses DHTs to map sensors to their attributes
Service-oriented framework for IoT	Horizontal model	Intermediary Proxy	Web Services (Java Senselets)	• Creation of virtual network topologies • Interoperable with the Internet
Practical realization of an IoT	P2P horizontal model	Base station Node	6LoWPAN (Web Services with EXI)	• Interoperable with the IPv6 Internet
e-SENSE	Modular horizontal model	Connectivity (CO) module	ZigBee (but not limited to)	• Contemplates IMS integration (through gateway extensions)
SENSEI	Modular horizontal model	Resource Directory	Description of resources (through WSDL)	• Defines the socioeconomics aspects around such environment

Table 2.2: Summary for current proposals for Middlewares and Frameworks

Chapter 3

ETSI: An European Standard for M2M

Founded in 1988, ETSI is an European Union recognized, non-profit, and independent standardization body whose goal is to create standards that can be applicable in the fields of Information Technologies, such as telecommunications, radio communications, broadcasting, among many others, thus enabling the telecommunications market to operate as a whole. Despite its worldwide projection, officially, ETSI is only responsible for standardizations within Europe.

As in other standardization organizations, ETSI also divides its areas of expertise among different Technical Committees (TCs), which are then responsible for conducting work related to their specific area, keeping in mind pre-defined interests/requirements, such as developing and maintaining standards.

The creation of these TCs is made according to the vision and strategy held by the main organization. In ETSI's case, for example, since one of its first areas of interest was radio communications, more specifically Global System for Mobile Communications (GSM), one of the first TCs that they created was the Special Mobile Group TC (firstly named TC GSM). Additionally, they also have created many others along the years (and also closed some), like the Smart Body Area Network TC, the End-to-End Network Architectures TC, the Aeronautics TC, the Powerline Telecommunications TC, among many others, in a total of 29 TCs, as shown in Figure 3.1.

One of the youngest TCs, created by ETSI to specifically deal with M2M topics, is addressed in the next section of this chapter, which covers what were the real motivations and interests that led to its creation.

Regarding the rest of this chapter, while section 3.2.1 is dedicated to explain the vision that the M2M TC has for the future of M2M systems, and how the standard could help to achieve it, the approach taken by the M2M TC to tackle the main existing problems is laid in section 3.2.2. Following, in section 3.2.4 is the standard's functional architecture, that delineates the most important parts of the architecture and workflows of operations, and for last, but certainly not least, in section 3.2.6 is given a brief introduction to some interesting companies/projects that stood out from the workshops organized by ETSI.

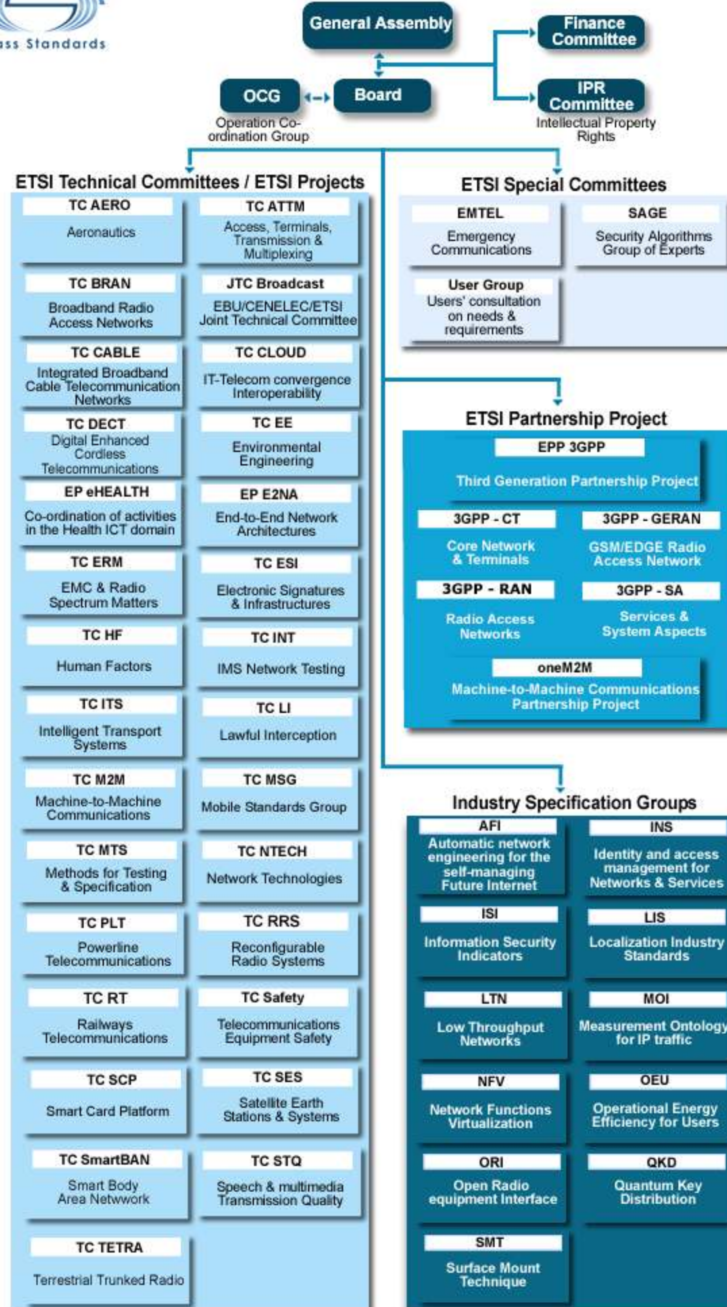


Figure 3.1: ETSI's organization chart [33]

3.1 TC M2M

As emphasised in the first chapters of this document (mainly in Section 2.3.1), not only M2M systems have been gaining momentum in recent years (approximately since 2011/2012), but as [18] forecasts, that momentum is expected to last.

Although in 2008 the momentum around M2M wasn't so perceptible as now, in [34] ETSI's M2M TC Vice-Chairman Joachim Koss states that it was already obvious the differences between the increasing saturation in human centric markets and the increasing number of stakeholders that were already interested in connecting machines to machines. Combining

that discrepancy together with the wide range of different applications that M2M systems have, since they can be applied in almost any area, a numerous number of attractive business opportunities started to emerge.

As so, in late 2008, after studying the importance of these kind of systems, not only at the time, but also in the future, through studies, reports and workshops - leading to the thought given in the above paragraph - ETSI decided to create a new TC for M2M (hence making M2M a topic of interest), assigning to it some core responsibilities, like developing and maintaining a high-level standard that would help to foster the implementation and deployment of full end-to-end M2M ecosystems, provide an ETSI main centre of expertise in the area of M2M and co-ordinate its M2M activity with that of others groups.

The creation of the M2M TC made ETSI, at the time, the only Standards Developing Organization (SDO) with well defined and concrete ideas to address standardization in the M2M world. Other SDOs, like Telecommunications Industry Association (TIA) and Alliance for Telecommunications Industry Solutions (ATIS), focused primarily in North America Information and Communication Technologies (ICT) standards, Association of Radio Industries and Businesses (ARIB) and Telecommunication Technology Committee (TTC), mainly focused in ICT standards inside of Japan, China Communications Standards Association (CCSA), mainly focused in ICT standards inside of China and Telecommunications Technology Association (TTA), with its focus in South Korean ICT standards, only started to take M2M into consideration afterwards, with TIA TR-50 created back in December of 2009, ATIS M2M FG created in August of 2011, ARIB M2M Study Ad Hoc Group established in June of 2011, TTC Smart Grid Advisory Group settled in October of 2010, CCSA TC10 created in February of 2010 and TTA M2M/IoT Forum established in October of 2009 - PG708 (TC7) created in early 2011.

Although all of the previous standards try to address more or less the same problem, i.e., ensure sustainable interoperability and convergence of M2M in their respective regions, not only there are differences among the different standards that make the global interoperability objective hard to achieve, leading to fragmentation among the M2M systems, but there are also common aspects that can be reused in order to avoid overlap of work.

To come to a consensus, a global initiative, under the name of oneM2M, has been signed in July of 2012. That initiative, signed by the seven major SDOs that publish telecom standards - TTC (Japan), ARIB (Japan), ATIS (USA), TIA (USA), TTA (Korea), CCSA (China), ETSI (Europe) - has the objective of stopping the individual work done among those SDOs, and join efforts to create a single unified standard. According to [6] and [35], this initiative tries to:

- Develop one globally agreed M2M specification, initially focused on Service Layer;
- Consolidate current M2M Service Layer standards activities into the oneM2M initiative;
- Partner/Collaborate with wireless and wireline SDOs and fora responsible for developing standards.

In a form of conclusion, as one of the signers SDOs of the oneM2M initiative, ETSI M2M TC now also has to coordinate their work with the community (other SDOs) in order to achieve the defined goals.

3.2 M2M Standard

3.2.1 Vision

ETSI's vision for the future of M2M systems highly resembles the vision of many others entities/organizations/individuals . Their main goal, just like ETSI's, is to achieve (or on the very least, help to) a common ground, i.e., a consensus among the IoT by disrupting the least possible already existing systems, so that an uniform, interoperable and sustainable way of connecting the IoT can be achieved in a near future.

With this standard, ETSI expects to start normalizing the high-level architecture that will sustain the future M2M systems, as well as the interfaces and protocols that prove adequate for these types of systems.

It's really important to state that, according to [34], ETSI's main objective was never to develop a new (M2M) standard from the ground, but rather take advantage of existing standardized systems and elements, evaluate them against the requirements of M2M systems, and fill in the gaps, i.e., enhance them as appropriate. This kind of guideline gives ETSI other degree of interoperability and integration with already existent systems, as it tries to adapt to them, rather than trying to modify them.

Although saying that the M2M number of communications expected for 2022 is about 18 billion may seem a little farfetched, when the number of M2M communications today are a mere 2 billion, the forecasts made in [18] shows that is indeed what's happening, making M2M and IoT in general the next 'boom' of the Information Technologies area. Taking those recent numbers into consideration, and joining in the fact that M2M systems is an already rapidly growing business that starts having its own market [18], interoperability (which, as said, standardization helps to achieve) is just a big necessary step to support that growth in a sane way.

Standardization is thus a necessary mark to stop the disruption effect that the ever increasing adoption of non interoperable solutions has on the evolution of M2M systems.

3.2.2 Approach

To achieve the ubiquity emphasised in the previous section, i.e., to achieve interoperability among M2M, and given the main technology problems and challenges/gaps affecting M2M systems [36], while conceptualizing the standard, ETSI started to tackle those problems by defining a set of characteristics, its key features, that define the gist and scope of the standard.

The next paragraphs are dedicated to explaining those key features, and how the standard, with them, tries to overcome those problems.

3.2.2.1 Standardization

Standardization is one of the most important characteristics when the objective is to achieve interoperability [37]. Making the communications between all the devices in the environment standardized, through the use of its well defined primitives (CREATE, RETRIEVE, UPDATE, DELETE - CRUD), for example, provides the first step to achieve an uniform consensus.

Another important step that ETSI takes to make this consensus possible is defining how data is structured. In M2M systems, because of the amount of different data that travels the networks, the way the data is structured is very important, and it's one of the things that makes most of the M2M solutions to not be interoperable. As a solution ETSI tries to keep a uniform way of storing data, independently of the type of data.

3.2.2.2 Verticality

According to ETSI , the solution for interoperability passes through substituting proprietary vertical architectures by standardized horizontal architectures, where M2M applications share a common infrastructure, environments and network elements that are responsible for connecting the lower layers, where the devices are, with the higher layers, where the services are.

Although this point was already addressed in Sections 2.3.2 and 2.3.3, it's relevant enough to state once again it's main advantage, i.e. high-level services/applications are independent of the infrastructure/devices used. Rather, applications can be used (and reused) in multiple scenarios and different infrastructures, giving varied perks, as stated before, like the easier creation of new applications and modification of existing ones, since they are no longer bonded to a specific technology, and a greater flexibility of the M2M system itself, which makes it more manageable.

3.2.2.3 Scalability

The word scalability in the area of computer science refers to an infrastructure/architecture/system that is able to handle larges amounts of work in a capable manner, meaning, in this case, that if much more devices are needed, the infrastructure doesn't need to change (or if it needs, it's not significant compared to the number of devices). It's also important to keep in mind that once the number of devices raise, the amount of data that travels the infrastructure also raises, stressing its capabilities (in a way that is dependent of how much and how constantly each device sends data to the infrastructure).

Given this scenario, ETSI M2M standard tries to address this problem in a divide and conquer way, dividing the architecture in two domains - the devices/gateway domain and the network domain. Thus, if more devices are required, the modifications that need to be made confine themselves to the device/gateway domain, given that the network domain isn't affected directly by how many devices are connected to each gateway.

3.2.2.4 Security

Given the variety of applications that M2M systems have, for ETSI it was truly important that security was approached from the beginning, as delaying it or keeping it aside would only create differences in implementations of the standard that would, in turn, lead to incompatibilities.

Thus, regarding security, two types of security are defined in the ETSI M2M standard. The first one is defined during M2M Services Bootstrap and Connection procedures. During the startup of the GSCL, it has to perform various steps in order to synchronize itself with the infrastructure, which guarantees that the gateway and the network domain agree, among other parameters, in the M2M Root key (Kmr). The Kmr is then used in the Service Connection procedure to perform mutual authentication of mId endpoints, as well as to derive the M2M Connection Key (Kmc), which, in turn, is used to derive M2M Application Keys (Kmas). Finally, this last key, the Kma, is used for encrypting data, authenticating and authorizing data that passes through the mId reference point. The second type of security is relative to data access. This security is made for guaranteeing that entities can only access resources of which they hold certain privileges, being that the privileges are specified in an attribute, named *accessRightID*, of the resource to what they apply to. The privileges are ensured through the use of READ, WRITE, DELETE, CREATE and DISCOVER flags. As

so, an entity can only read a specific resource, if under its permissions the entity is listed as one of the entities with read access, for example.

With this two types of security the standard ensures that not only communications are being performed among authentic endpoints and authorized applications, but also that the data transmitted by applications is secured and authentic. Moreover, since the previous security mechanisms don't prevent authorized and authenticated entities to access resources to which they don't have enough privileges, the standard reinforces security by providing a way of blocking entities from accessing resources that they don't have privileges to.

3.2.3 High-level Architecture

Within the scope of its M2M standard, ETSI divides the high-level architecture in two main domains, which are, the network domain and the M2M devices/M2M gateway domain - from now on, for brevity, M2M gateway and M2M devices will be referred simply by gateway and devices, respectively.

Although this subsection aims to give an overall explanation of the architecture and functional specifications of both the domains, more detail and emphasis will be given to the former (devices/gateway domain), since that is where the gateway is, and to the gateway itself in particular.

In order to avoid ambiguities when referring to different Service Capability Layers (SCL) (which as the standard explains, it's a layer that provides M2M functionalities), since different entities provide different capabilities in their respective SCLs, the standard refers to the gateway's SCL as GSCL (the 'G' stands for Gateway). Additionally, the device's SCL are referred as Device Service Capability Layer (DSCL) (the 'D' stands for Device), and the network domain's SCL are referred as Network Service Capability Layer (NSCL) (the 'N' stands for Network).

In Figure 3.2 it's represented the high-level architecture proposed by the standard, divided in the devices/gateway and network domains. As it can be seen, the devices/gateway domain is the lowest-level, since it's where the interaction with the real world (sensing and/or actuation) is done. Furthermore, this domain itself is divided in two entities, the devices and the gateways, being the two connected by an M2M Area Network:

- **Devices:** These entities are the ones responsible for doing the real world sensing and/or actuation.

  Regarding if the SCL is local or not, and if the devices are able to communicate with it, using the formats and protocols specified in the standard, the standard contemplates three types of devices (including a legacy one):

  - **Case 1:** It's the case where a device has a local SCL (DSCL), hence is able to connect directly with the NSCL in the network domain, not needing a gateway to intermediate the communication. Since this type of devices can intermediate the connection between the network domain and other devices that are not able to connect directly with it, they need to be a little more capable in terms of resources than the remaining (case 2 and legacy case), since they are going to perform procedures like registration, authentication, management, etc., with the network domain, for the other more limited devices.

    Moreover, this devices run Device Applications (DAs) that interact with the DSCL, in order to take advantage of the provided functions/services.
  - **Case 2:** In this case the device doesn't have a local SCL, but, however, it can communicate with one. Thus the communication with the NSCL in the network

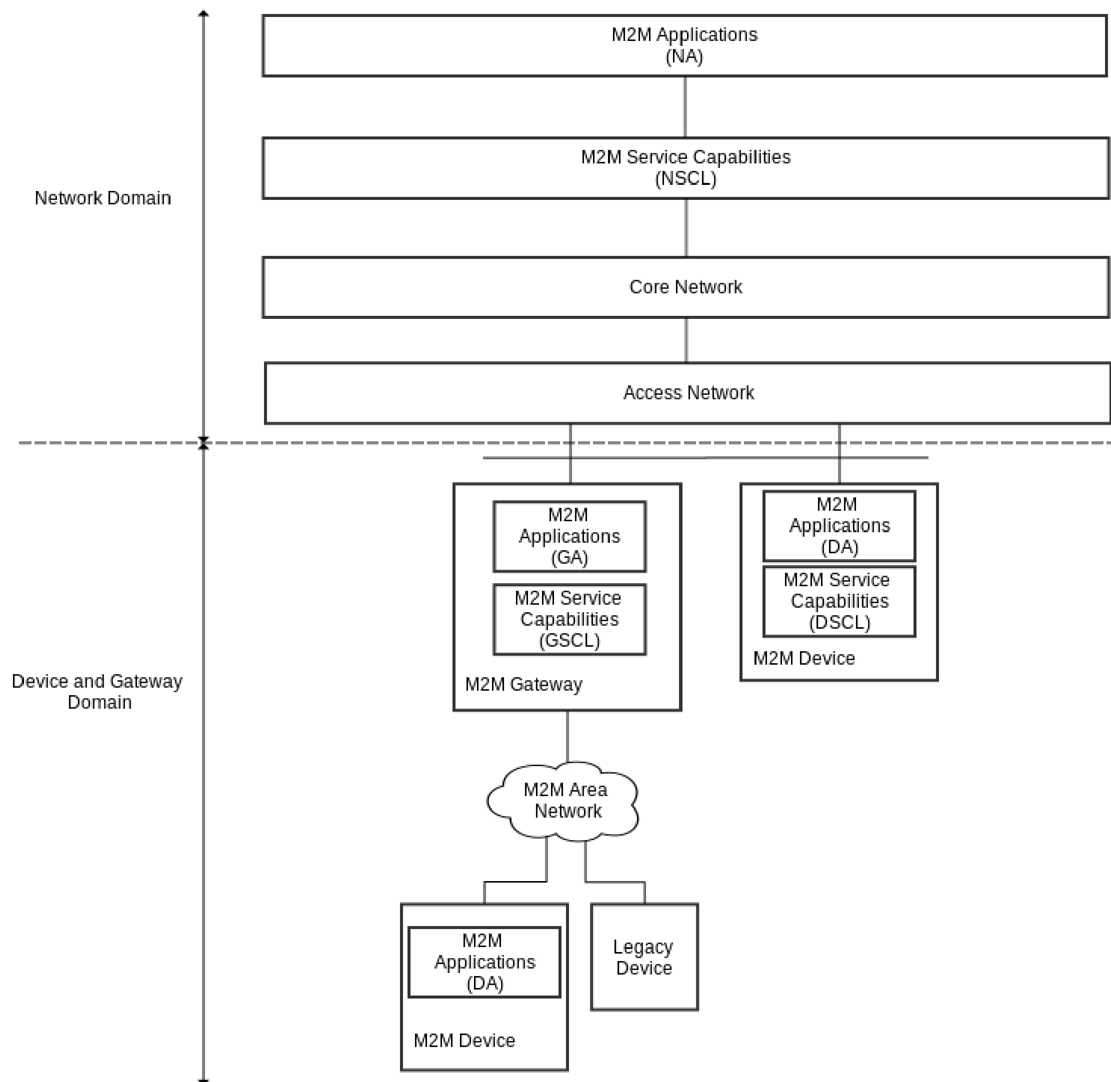


Figure 3.2: High level architecture for M2M [5]

domain is intermediated by a gateway (or a device with a SCL), that acts like a proxy. As so, in this case the devices connect to the gateway and not to the network domain, and the gateway, in turn, connects to the network domain.

This type of devices are also able to run DAs, but since they don't have a SCL, they interact with the SCL of the gateway/device.

- **Legacy case:** Just like case 2, in this case the device doesn't have a local SCL either, but unlike case 2, this type of devices aren't capable of communicating with a SCL, since they don't comply with the standard. Thus, this case needs to be supported by gateways/devices with a SCL that make the appropriate translations (both ways, i.e., legacy device <-> gateway) of whichever technologies used by the legacy device (e.g. ZigBee, 6LoWPAN, Bluetooth, among many others), enabling communications between the legacy device and the SCL.

  As a consequence, these devices applications are not denominated as DAs,

since they are not compliant with the standard. They are just simple applications that are normally executed in the device.

– **Gateway:** This entity is very similar to the case 1 devices, since it also runs M2M applications (Gateway Applications (GAs)), has a local SCL (GSCL), hence being able to connect directly with the NSCL in the network domain, and act like a proxy between the devices that are not able to directly connect with the network domain, providing GSCL functionalities to those devices.

  The only difference between the gateway and the case 1 devices is that the gateway is suppose to be even more resource capable, since more devices are supposed to connect to it, and the type of services provided are also more demanding.

  Given the importance of the gateway for this Dissertation, the next section will explain it in more detail.

On the other hand, the network domain is the most high-level domain, since this domain defines how services's data is presented to end users, i.e, information coming from the devices/gateway domain starts gaining meaning in this domain, then being used for drawing graphics for analysis, giving meaningful alerts, among many other use cases. As in the case of the previous domain, the devices/gateway domain, this domain also has to have at least one entity (e.g., a server) to run the M2M applications and the SCL (NSCL) that provides the services that expose M2M functionalities.

Finally, the connection between the network domain and the gateway/devices domain is made through an Access Network, that enables the interactions between both domains.

3.2.4 Gateway Functional Architecture

The previous section provided a quick and overall explanation about the role of the gateway in the proposed standard. This section, besides showing which components make up the gateway's architecture, has also the objective of explaining how they interact between each other, and additionally, how the communication between the gateway and the remaining entities is performed.

After showing in Figure 3.3 what are the components that compose the gateway, i.e., the dIa and mId Reference Points, the GA and GSCL, a more detailed explanation of each of those components follows.

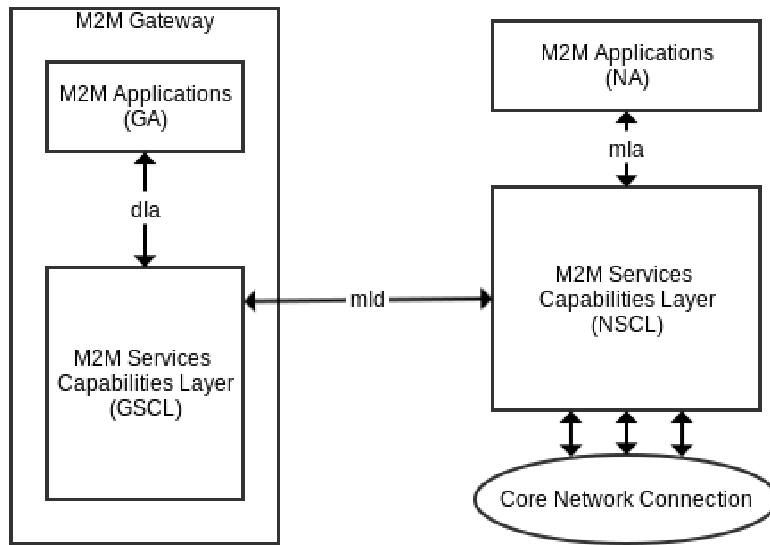


Figure 3.3: GSCL functional architecture [5]

3.2.4.1 Reference Points (mId and dIa)

In a broader definition, the reference points simply expose the capabilities provided by some entity's SCL. Thus, reference points can be seen as simple interfaces that expose specific functionalities (services), and where communication is made by pre-defined primitives, whose format is specified in the section 10 of [38].

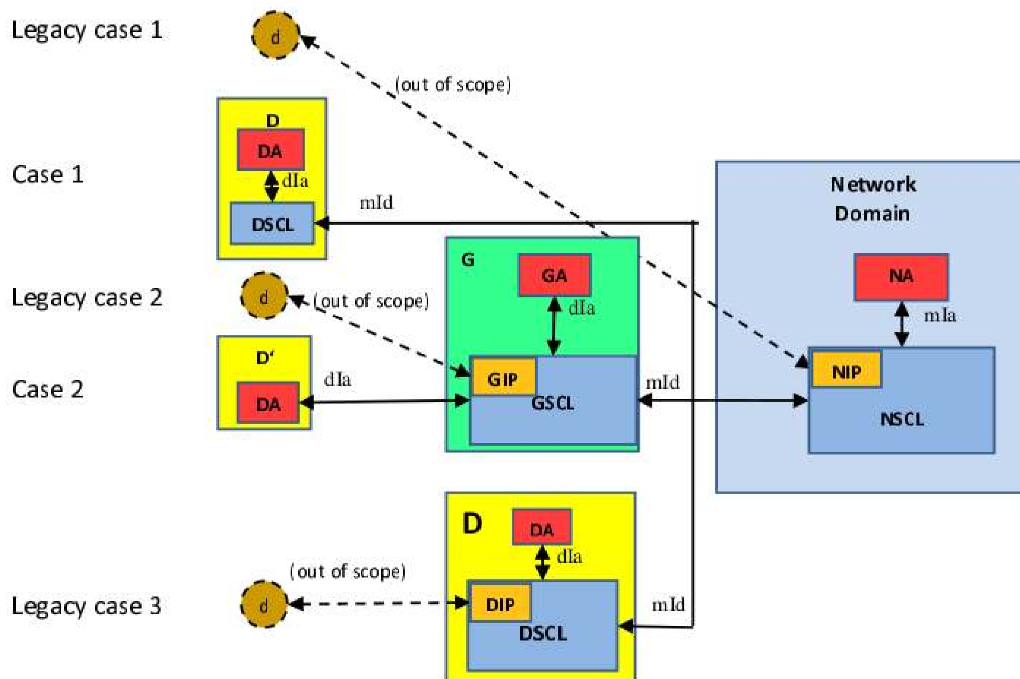


Figure 3.4: Mapping of reference points [5]

Regarding the gateway, as can be seen in Figure 3.4, there are only two distinct interfaces

defined by the standard that enable the communication (local and/or remote) with the GSCL. Those interfaces are the dIa and the mId interfaces.

3.2.4.1.1 mId

The mId interface allows a GSCL, residing in the devices/gateway domain, to access the NSCL, present in the network domain, and vice-versa. Hence, the communications between the gateway and the network domain is made through this interface.

3.2.4.1.2 dIa

On the other hand, the dIa interface not only allows DAs to access a GSCL, but also allows GAs to access the local GSCL. As said in the previous section, devices that are compliant with the ETSI M2M standard, can have, or not, a DSCL. Hence, in the first case shown in this paragraph, where a DA accesses a GSCL through the dIa interface, the device running the DA would not have present a DSCL, thus needing the gateway SCL to act like a proxy.

The protocols that the standard recommends for enabling the communications between the gateway, the devices and the network domain, using the reference points explained above, are Constrained Application Protocol (CoAP) [39] and Hypertext Transfer Protocol (HTTP) [40]. However, this recommendations are just mere suggestions, and are not impositions made by the standard, and as so, other protocols can be used to perform those communications.

3.2.4.2 GA

This type of applications are executed in the gateway, and are only responsible for performing the tasks for which they were programmed to. For that, as explained before, GAs may access the GSCL, through the dIa interface, to retrieve some desired, or even subscribe to some device's data.

For example, considering one of the scenarios presented in [41], where a set of traffic cameras, forming a Wireless Local Area Network (WLAN), are installed in motorway overpasses, or in remote stretches of the roadway for measuring the average speed of passing cars. Going a little further, imagine also that that network of cameras is connected to an ETSI M2M gateway, which receives and stores the values (for example, the average speed and the license plate) measured and sent by the cameras. However, only storing the data doesn't help much, since data must be processed to check for offenses. Therefore, for processing this example's values, a GA could be implemented. That GA would subscribe to the received values and process the received values for any illegalities, and if necessary, send the license plate and respective infraction upwards, to the upper domain. This is a good example of the usefulness of this component, since besides of processing values, it also helps to reduce the overall bandwidth used (only values that correspond to infractions are sent to the core network).

Despite of it's simplicity, the above example sets the idea that the data sent by the gateway to the upper domain (the network domain), doesn't need to be the data originally received by the gateway (sent by the devices), but rather data already processed that can be, in some way, useful to services residing in the network domain. This component, GA, is thereby essential to confer the gateway with extra flexibility, avoiding restraining it to being a simple data pipe.

Other much more complex examples can also be performed, as is the case of aggregating either a subset or all the received information, or even assisting the less resource capable devices that send data, by storing/processing part of the information, for example.

Finally, it's important to note that the applications of a specific gateway can only be registered in the local GSCL, from which they will withdraw the information that they may eventually need.

3.2.4.3 GSCL

Since it is the GSCL who provides all the capabilities that allows the gateway to store and manage not only the data received from the devices, but also the information regarding those devices, this is considered its main component. Some of those capabilities are then exposed through the reference points explained before, allowing external entities, as the devices, or even others in the network domain, to take advantage of those functionalities.

The following points will be dedicated to enumerate and describe the capabilities that compose this GSCL, as stated by the section 5.3 [5] and shown in Figure 3.5.

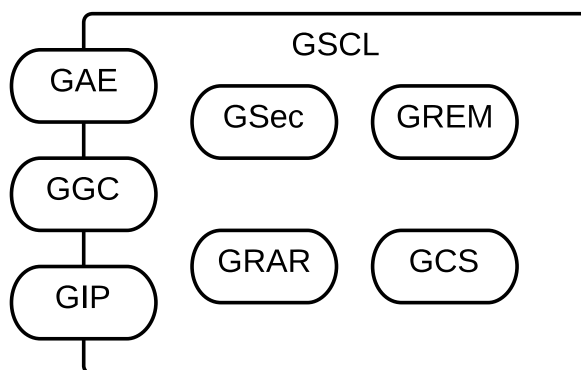


Figure 3.5: GSCL capabilities

3.2.4.3.1 Gateway Application Enablement (GAE)

This capability is mainly responsible for exposing, via the dIa reference point, the functionalities implemented in the GSCL.

By using this capability, not only GAs and DAs are able to register to the GSCL, but messages exchanged between applications registered in the same GSCL are also routed directly (intra-routing).

GAE also validates the syntax of all the received requests in dIa. After validating the requests, routes them to the appropriate capability(ies) (for example, Gateway Reachability, Addressing and Repository (GRAR)), that will further process the request.

Furthermore, although the details are left out of the scope of the current standard, this capability **can** also provide authentication and authorisation of GAs and DAs before allowing them to access the GSCL.

At last, and if applicable, it's also responsible for generating the charging records pertaining to the use of the capabilities.

3.2.4.3.2 Gateway Generic Communication (GGC)

The GGC capability, just like the GAE capability, is also responsible for exposing the functionalities implemented in GSCL, although in this case via the mId reference point. Resorting also to other capabilities, this capability is able to validate the syntax of all the received requests in mId, routing them for further processing, deliver messages in accordance with the

Service Class, which allows to define priorities among different types of messages and handle name resolution for requests originated within an Area Network.

Furthermore, GGC is also responsible for transport session establishment/teardown and security keys negotiation. The former uses the keys provided by the Gateway Security (GSec) capability, if the establishment of the transport session is performed in a secure manner.

Using the key material derived from the session establishment, this capability also provides encryption and integrity protection of data exchanged with the NSCL. Thus, when security is required (such as encryption and integrity protection), for transmitting data from the gateway to the NSCL and vice versa, GGC ensures secure delivery. If needed, different applications can also have different security associations, using the symmetric Kma keys. More specifically, after a successful authentication and key agreement, GGC receives from GSec application-specific keys for applications that are associated with the gateway and are authorized to access the NSCL. Those keys are then used for authentication and authorization of the gateway applications with the NSCL, over the mId interface. As such, for every application that requires secure data transport, will be established one secure session over the mId interface, using the corresponding application-specific key.

Finally, GGC also reports transmission errors.

3.2.4.3.3 GRAR

GRAR is the capability that provides the mapping between a device's name, or a group of devices, and a set of information, as the reachability status, a set of routable addresses, or even a schedule pertaining the reachability of the device.

It's also responsible for managing subscriptions and notifications, which enable the NSCL, for example, to be notified about a modified resource, and allows to create, delete and list a group of devices, which is useful to manage groups of devices that are related to each other in some way.

Furthermore, it not only stores information about DAs, GAs and the NSCL, so they can be reached later, if needed, but also stores and manages all the received data, not only enabling further processing, but even supporting cases where the source is offline when the data is requested (the device, for example).

3.2.4.3.4 Gateway Communication Selection (GCS)

This capability provides a way to, based on defined policies, choose which network should be used so that the gateway can access the NSCL. This functionality is useful when there are multiple networks and the gateway needs to opt by one to make the communication.

In case of a communication failure using the first chosen network, it also provides a way to choose an alternative network to be used instead.

3.2.4.3.5 Gateway Remote Entity Management (GREM)

GREM is the capability that provides the remote management functionalities. Thus, it not only allows the gateway to act as a remote management client, enabling the execution of a set of functionalities on itself, but also allows the gateway to act like a remote management proxy for a set of devices, enabling the network domain to manage the devices in an abstract way. Hence, through this capability, the network domain can execute commands in specific devices, independently of the type of network or technology that is being used by them.

3.2.4.3.6 GSec

This capability, being the one that provides the security functionalities, is responsible for supporting the service bootstrap of the gateway, for supporting a key hierarchy for authentication and authorisation, for initiating mutual authentication and key agreement, and also for the storage and handling of M2M Connection Keys.

Additionally, GSec **can** also report the integrity validation status to the NSCL and react to posterior actions triggered by the NSCL.

3.2.4.3.7 Gateway Interworking Proxy (GIP)

GIP allows that non ETSI M2M compliant devices (non ETSI devices) can also take advantage of the above described capabilities. For that, what a GIP does is, convert all of the commands sent from the non ETSI devices to the gateway, in messages that the gateway understands, and also convert all the messages that the gateway sends, in commands that the non ETSI devices understands, so that both can communicate.

It's important to note that the use of this capability is optional, meaning that it is only deployed when needed.

3.2.5 SCL Resource's management

In a general sense, the data and the information kept by a SCL (regardless if it's a SCL that is residing in a device, in a gateway or in the network domain) is stored in a tree structure, well defined by the ETSI M2M standard, in the form of resources. As stated in [5], those resources can be thought of as buckets that can hold some application specific data. Those buckets then have properties themselves, relations, and the way they are structured is shown in the following Figure C.1.

The resources's structure above was developed following a RESTful architecture style, meaning that each resource is addressed in an unique way, and operations upon them are made through the four basic, well defined methods - also known as CRUD methods:

- CREATE: create child resources;
- RETRIEVE: read the content of the resource;
- UPDATE: rewrite the content of the resource;
- DELETE: remove the resource.

Additionally, in [5], ETSI M2M defines two additional actions, described as useful, which are:

- NOTIFY: used to indicate the operation for reporting a notification about a change in a resource that as been subscribed to;
- EXECUTE: for executing a management command/task which is represented by a resource.

Thus, by using a RESTful style, ETSI mandates that the management of the resources residing in a SCL is only done with the above methods, and by transferring representations of those uniquely addressed resources. The above methods are mapped in [38] as SCL primitives (primitives), which are used to interact with the interfaces (dIa, mIa and mId).

In order to manage the resources, the standard foresees two ways of interacting with a SCL: either locally or remotely.

A good example of the latter is a local interaction between an application and the respective local SCL, where in this case, according to the standard, the local interaction can be done through some kind of internal API (library) or through the use of the interfaces.

For the former, which for example can be the interaction between two SCLs, the remote interaction is only possible through the use of the interfaces.

To finalize this subsection it's important to note that the primitives are no more than a format defined by the standard to express an operation that is to be executed in a SCL resource (CRUD operations). In remote interactions, a protocol, like HTTP or CoAP must be used to transport the information held by the primitive. In [38] there are two annexes, C and D, that show how to do the mapping between HTTP or CoAP and the primitives, respectively.

3.2.6 Workshops results

With the objective of testing different implementations and disclosing the standard, ETSI has been organizing workshops and presentations since 19th October of 2010, when the first workshop was held.

From those workshops, three very different projects have had more attention, each one, with its own perspective, bringing something new to the M2M world. Those are, Cocoon from Actility, Sensinode, and OpenMTC from Fraunhofer and Technical University of Berlin (Technische Universität Berlin, TUB).

The next chapters will be used to analyze both of the three projects, and their evolution since their start.

3.2.6.1 Cocoon

Cocoon, entitled as the first ETSI M2M Application Store and Market Place, is an open source project from Actility that started in 2011, with the purpose of providing a way to interconnect developers that want to produce M2M applications and hardware manufacturers that want to build compliant ETSI M2M gateways for the masses, and thus help to disseminate the IoT, as stated in [42].

Since its start, Cocoon has always been intended as a centralized M2M application store (ThingStore, which ought to be open this year, 2013), taking advantage of ETSI M2M RESTful primitives to provide a development framework which is independent of the underlying hardware.

To achieve that ecosystem at a faster rate Actility is developing, in parallel with ThingStore, its own ETSI M2M compliant gateway, thus not having to wait for 3rd party implementations of an ETSI M2M gateway, whose architecture is represented in Figure 3.6.

Cocoon's gateway uses Java as the primary language for its implementation. Additionally, it also uses OSGi, more specifically Knopflerfish OSGi 3.0.2, which gives the following bundle services, among the most important, right out of the box:

- logging;
- bundles management;
- telnet;
- console;
- serial port;

Using OSGi, Cocoon is not only able to maintain a more modular architecture, but also to provide more attractive services right from the beginning (like the ones stated above).

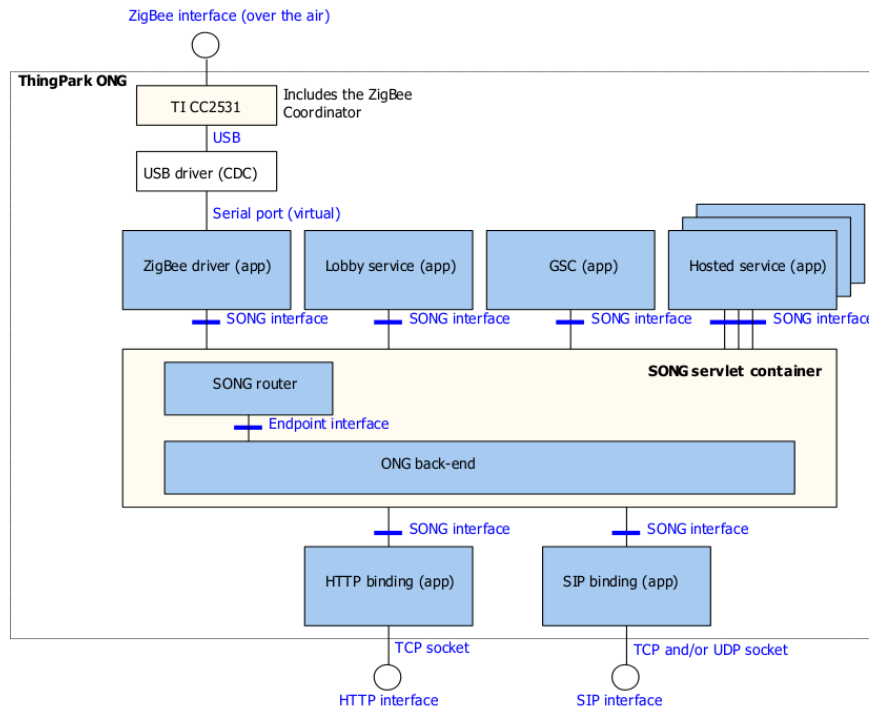


Figure 3.6: Object Network Gateway (ONG) internal architecture [43]

Regarding their business model, profit is intended to come from the M2M application store, where are running the automation applications that interact with the M2M gateways and devices to collect data. With this model, developers can build their own applications, and take advantage of Actility's M2M environment, but any profit that the application may return must be shared with Actility too (Revenue sharing).

3.2.6.2 OpenMTC

The main vision of this project, which stands for Open Machine Type Communication, goes beyond of simply providing/implementing a ETSI M2M compliant middleware platform for M2M applications and services, [44]. It also aims to bring IMS into the platform, as a form of integrating the M2M world with the telecommunication operators (telcos) world. The authors state that IMS, and other state of the art operator network architectures, are not viable solutions for large communication scenarios, as it's required for M2M communications. Thus, if telcos want to become part of the M2M world, that is, M2M operators, other solutions must be found.

By coupling IMS together with ETSI M2M standard, and integrating it into the M2M environment, they not only aim to enable IMS applications to take advantage of information coming from M2M environments, like temperature, humidity, among other infinite possibilities, but also to enable the M2M environments to take advantage of IMS well known services, like the presence service, and access to the Home Subscriber Server (HSS).

OpenMTC M2M middleware implementation [45] is composed by two SCLs, a GSCL, and a NSCL, as specified by ETSI M2M standard. Additionally, it's also scheduled a DSCL for Android, enabling smartphones to be used as sensors/actuators in the lower layer of the middleware.

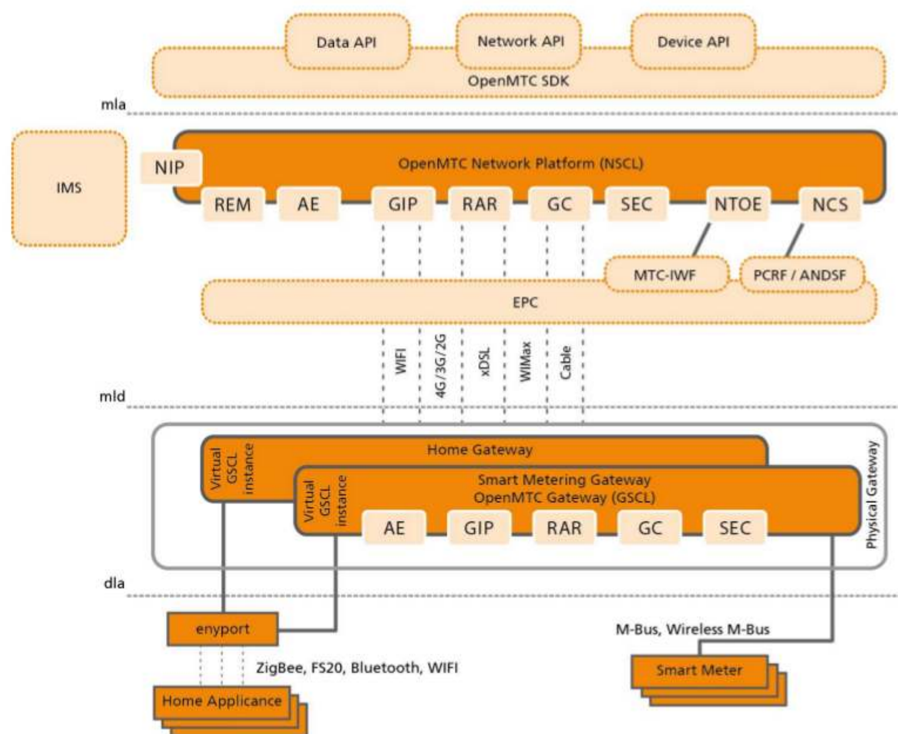


Figure 3.7: OpenMTC architecture [45]

As we can see in the architecture shown in Figure 3.7, OpenMTC integrates the network technologies in Packet Network Core, which is supported by the Evolved Packet Core (EPC) Fraunhofer's own implementation (OpenEPC). Thus, for being able to access the network domain, or vice-versa, the gateways have to communicate through the EPC. Moreover, the integration between IMS and the M2M middleware is made through a Network Interworking Proxy (NIP), that enables communications between the IMS and the network domain.

Finally, it should be mentioned that Fraunhofer also has a IMS implementation, named OpenIMS, which shall be used in OpenMTC.

3.2.6.3 Sensinode

Although the other two projects aim to implement the ETSI M2M standard (even if only part of it) to provide a M2M environment, thus enabling interoperation from the lower layers all the way to the applications used by the final consumers, the main objective of Sensinode is not to provide a M2M environment, but rather to define minimal and efficient protocols/stacks to enable constrained devices to communicate with higher levels of the environment.

Sensinode is a company that highly defends the adoption of the IP stack, through the use of 6LoWPAN, for constrained devices, giving devices the power to interact with the Internet. As Sensinode presents [46], with an IP stack many problems would be resolved right from the beginning, as using IP would allow a much more uniform communication between the different layers, and thus keep a distance from the pejorative vertical stove pipe solutions that have been stopping the evolution of M2M systems.

Such as emphasized in the presentation, although devices can talk with the Internet directly, gateways are still needed, thus not making their vision inconsistent with ETSI's. The reasons are:

- bridge heterogeneous networking technologies;
- local storage;
- data processing, filtering and analysis;
- semantic annotation and metadata;

Still from [46], Figure 3.8 shows the true scope of their vision, with sensors able to communicate directly with gateways through CoAP/UDP, without the need for Interworking Proxies, as defined by ETSI.

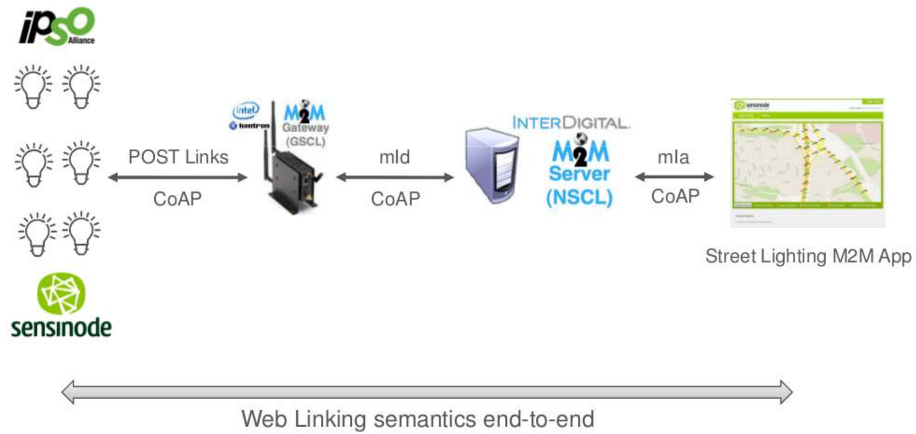


Figure 3.8: Embedded Web using ETSI M2M [45]

Chapter 4

Implementing an ETSI M2M compliant gateway

In the M2M standard, ETSI describes in detail the proposed high-level architecture, as well as the functional architecture (which includes the reference points, the flow of events and the service capabilities), and the security and resource procedures. However, as a standard, it doesn't detail anything regarding the implementation, giving the developers enough freedom for implementation.

Thus, this chapter aims not only to explain how the implementation was done, showing diagrams and snippets of code when appropriate, but also to explain the reasoning behind the decisions that were made throughout the implementation, giving a more powerful insight of the work done in this Dissertation.

4.1 Defined objectives

Despite being ideal, having a totally functional and tested implementation of a ETSI M2M gateway by the end of this Dissertation would be unrealistic, considering not only the amount of work, but also that the standard is not yet finalized.

Hence, keeping in mind that in the final delivery a functional implementation of an ETSI M2M gateway had to be presented, a set of functionalities and objectives, considered to be the most relevant for an initial implementation, were chosen.

Apart from the functionalities and objectives mentioned below, note that the architecture defined in this document, more specifically in Section 4.3, also takes into account the GREM, GSec and GCS capabilities (whose functionalities were not implemented).

- GAE:
    - Allow applications to register;
    - Ensure that the requesting entity has enough access rights to perform the requested operations;
    - Provide validation of primitives and subsequent routing of requests to the appropriate capabilities;
    - Allow the creation and validation of groups;
- GGC:
    - Allow the GSCL to register in the NSCL;

  - Validate primitives and route them to/from the NSCL;

- GRAR:
  - Provide storage of information about registrations;
  - Provide storage of the received data;
  - Allow the retrieval of the stored data;
  - Provide management of access permissions;
  - Provide management of subscriptions and notifications.

- GIP:
  - Provide a GIP template architecture and an example implementation that supports it.

- GA:
  - Provide a way to create GAs in a simple way;
  - Provide local and remote connectivity with the GSCL, through the dIa reference point;
  - Implement an example that supports it.

- Communication Modules:
  - Provide HTTP and CoAP proxies;
  - Provide HTTP and CoAP clients;
  - Support XML and JSON representations/mappings of the primitives;

To finalize, it's important to emphasize that, since the gateway was intended to be used in the project Apollo, which had a demo sooner than this Dissertation final delivery date, most of the above functionalities and objectives needed to be implemented earlier than that.

4.2 Resources representation

A large part of the implementation of the gateway concerns the resources, their representation, their relations, and how the information is stored in the gateway. In that regard, the implementation started by the representation of the resources as defined in the standard, and showed in Figure C.1.

However, it should be noted that by 'resources representation' this document does not refer to the mapping of the resources structure into Plain Old Java Objects (POJOs), but rather its representation in XML Schema Definitions (XSDs), as explained in [38], Annex B. Moreover, sill in that annex, the existence of a XSD for representing each resource is not stated as optional, but as mandatory, whereas only one other language (besides XML) is contemplated by the standard, which is JSON, and is obtained through the mapping of XSDs.

As such, to take advantage of the XSDs, the Java Architecture for XML Binding (JAXB), was used to generate the POJOs from the XSDs, as well as to marshall those POJOs into appropriate XML streams, and vice-versa, i.e., unmarshall. Since JAXB is only a specification, the implementation used in this work was EclipseLink MOXy [47].

The use of JAXB becomes particularly useful in this situation, when compared with Java XML parsers like Document Object Model (DOM), and Simple API for XML (SAX), where the standard has not yet been approved, meaning that resources representations may have to

be changed unexpectedly. Thus, using JAXB, if the resources's structure happens to change, only the XSDs need to reflect that changes.

Snippet 5 (B) shows the XSD that represents the application resource, while the Snippet 6 (B) shows the correspondent JAXB generated POJO.

Apart from representing the resources, it's also important to enable their storage and retrieval, in order to facilitate its handling/manipulation. To achieve that, a local embedded database, designed appropriately to accommodate those resources, is used in this gateway. The next section addresses that topic.

4.2.1 Database model

One of the most important aspects about the implementation of the database is that it should carefully reflect the relations between the stored resources. Thus, to represent the resources hierarchical structure, defined by the ETSI M2M standard, a hierarchical model was implemented in the database, hence enabling the retrieval of the resources, their respective attributes and associated relations, when needed, without losing any information

For the database, SQLite, an open source embedded Relational Database Management Systems (DBMS), was chosen. Appendix A explains in detail the reasoning behind that decision.

This subsection will explain the process behind the conception of the database hierarchical model. From the class model, where the entities and relationships are identified through the gathering of the requirements, to the physical model, where tables and columns with Primary Keys (PKs) and Foreign Keys (FKs) are already present.

4.2.1.1 Class Model

In order to obtain the (hierarchical) class model (used to represent the conceptual model of a database), it's highly important to analyse the kind of information that needs to be supported. The following paragraphs will be dedicated to explain, gradually, how the final class model was obtained, as well as how the physical model, that will be deployed in the database, was derived from it.

4.2.1.1.1 ***Collection Resources***

The first analysed information is the collection resources. Each collection resource, to be appropriately represented, needs:

- ***path:*** points to a unique location, which is where the collection resource is stored;
- ***key:*** identifies the type of collection resource (i.e., *scls*, *applications*, *containers*, *contentInstances*, for example, among many others);

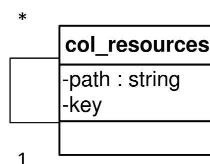


Figure 4.1: Representation of collection resources, in the class model

To support the above information in the database class model, as can be seen in Figure 4.1, it is necessary to create a class, named *col_resources*, that will be responsible for storing representations (i.e., its *path* and *key*) of collection resources.

This *col_resources* class has a recursive relation with itself, thence the denomination of recursive class for this types of classes. The recursive relation, being of the type one-to-many (1-*), means that a specific collection resource can have zero or more collection resources associated, despite of these latter collection resources may only have the initial collection resource associated.

It's important to note that it's the recursive relation that enables the representation, in the above class, of relations between collection resources, that is, cases where a collection resource is a parent of another collection resource.

4.2.1.1.2 *Collection Resources's normal Attributes*

Additionally, collection resources also have child attributes, which can be categorized as normal, or special attributes. The normal attributes are represented by:

- ***key:*** identifies the type of attribute (i.e., *accessRightID*, *creationTime*, *lastModifiedTime*, etc);
- ***long_value:*** stores data of type long/integer;
- ***double_value:*** stores data of type double;
- ***date_value:*** stores data of type date.
- ***bool_value:*** stores data of type boolean.
- ***blob_value:*** stores data of type blob.
- ***string_value:*** stores data of type string.

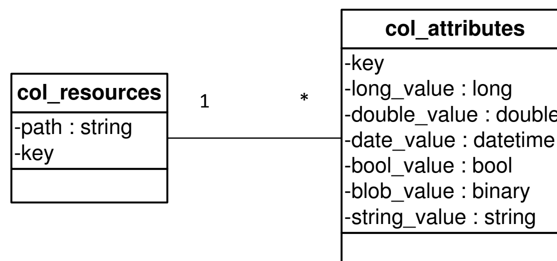


Figure 4.2: Representation of collection resources and respective normal attributes, in the class model

The Figure 4.2 shows the new class added to the class model, i.e. the *col_attributes* class, which allows the storage of the attributes of a specific collection resource. The relation between the classes *col_resources* and *col_attributes* is of the type 1-*, since each collection resource can have zero or more normal attributes, but each normal attribute can only have one collection resource associated.

4.2.1.1.3 *Collection Resources's special Attributes*

On the other hand, the special attributes have a very similar representation to the normal attributes, shown above. The only difference is that the special attributes **can** have a list of values, each one identified by its own *id*. Thus, to store that *id* information, a new field, *string_id* is needed:

- ***string_id:*** name/id of the attribute;
- ***key:*** identifies the type of attribute (i.e., *accessRightID*, *creationTime*, *lastModifiedTime*, etc);
- ***long_value:*** stores data of type long/integer;
- ***double_value:*** stores data of type double;
- ***date_value:*** stores data of type date.
- ***bool_value:*** stores data of type boolean.
- ***blob_value:*** stores data of type blob.
- ***string_value:*** stores data of type string.

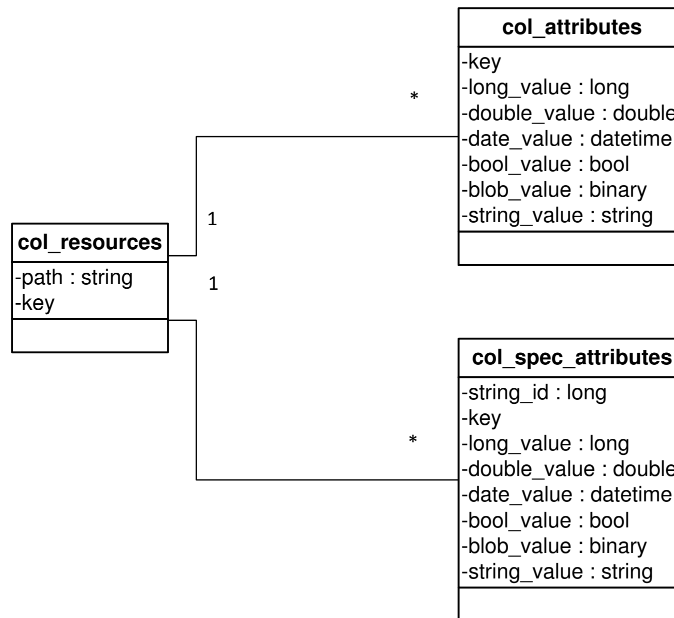


Figure 4.3: Representation of collection resources and respective normal and special attributes, in the class model

Just like in Figure 4.2, Figure 4.3 shows how the three classes, *col_resources*, *col_attributes* and *col_spec_attributes*, relate to each other. The *col_resources* and *col_spec_attributes* classes, for the same reasons, have the same relation that is present between the *col_resources* and *col_attributes* classes, i.e. the *col_spec_attributes* class has a relation of * to 1 with the class *col_resources.*

Regarding the previous *col_attributes* and *col_spec_attributes* classes, it's important to point that the approach of dividing the different types of data (long, double, date, ...) was taken to simplify the processes of searching/filtering the stored data. With the previous approach, searching for collection attributes with a date prior to today's date, for example, is not a resource consuming process, since dates are stored directly. If another approach had been used, and every types of data were stored in the same field 'value' of type string, while another field 'type' was used to identify the type of data stored in the 'value' field, it would slow down the performance of search/filtering queries. Using the previous example, in this last approach the dates would be stored in the string format, thus, either they would be converted to the date format for comparison, or compared as is (i.e., strings), and neither of these options would be better than the chosen approach.

At this point, the three classes already explained, and also shown in Figure 4.3, enable the database class model to represent collection resources and their associated attributes.

However, as explained before, there are two types of resources, the collection resources, already addressed, and the single resources. To additionally enable the representation of single resources in the class model, the analysis made previously for the collection resources, will have to be done now for the single resources.

4.2.1.1.4 *Single Resources*

Although equal to the collection resources, regarding their representation, and, as so, they could possibly be represented in the same class, there are two reasons that led to the explicit distinction between collection and single resources (thus forcing the existence of two classes, one to represent each type of resources): firstly, the fact that single resources don't have other single resources as children, only collection resources; secondly, for management and speed improvement, since splitting a bigger table into two smaller tables provides faster searches, mainly because collection and single resources are not scrambled in one big table, but rather appropriately organized between two tables.

Therefore, single resources are represented by:

- ***path:*** points to a unique location, which is where the collection resource is;
- ***key:*** identifies the type of collection resource (i.e., *scls*, *applications*, *containers*, *contentInstances*, for example, among many others).

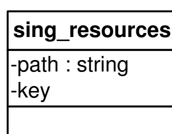


Figure 4.4: Representation of Single resources, in the class model

As mentioned in the previous paragraph, and as one can see in the Figure 4.4, the *sing_resource* class is very similar to the *col_resource* class, being the only difference the fact that it isn't a recursive class, since a single resource can't have another single resource as parent.

4.2.1.1.5 *Single Resources's normal and special Attributes*

Regarding the representation of the attributes of single resources, both the normal and the special attributes representation is equal to the representation shown above, of collection resources's attributes. As so, Figure 4.5 shows the relation between the *sing_resources* class, and it's associated *sing_attributes* and *sing_spec_attributes* classes. As in the case of *col_resources* with its attribute classes (shown in Figure 4.3), the relation between *sing_resources* and *sing_attributes* is also 1 to *, as well as the relation between the *sing_resources* and *sing_spec_attributes* classes.

Until this point, explanations were only about the collection and single resources (including their respective attributes and relations). Explanations regarding relations between resources only covered relations among resources of the same type, i.e., if a collection resource was parent of another collection, leaving aside relations between resources of different types (a collection resource being parent of a single resource or vice-versa).

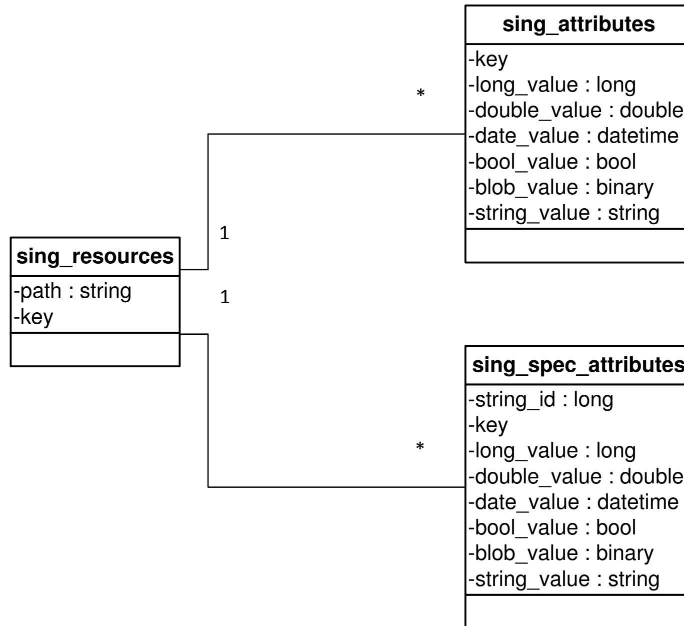


Figure 4.5: Representation of single resources and respective normal and special attributes, in the class model

4.2.1.1.6 *Relation*

Thus, to cover relations between resources of different types, there needs to exist another class that stores the relation between collection and single resources:

- ***relation:*** stores the relation between a specific collection resource and a normal resource. For example, if a specific collection resources a parent of a normal resource, or vice-versa.

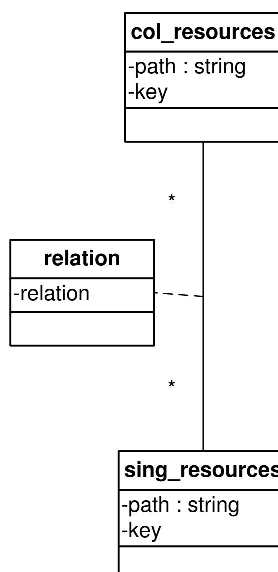


Figure 4.6: Relation representation, in the class model

The class *relation*, which is an association class between the *col_resources* and *sing_resources* classes, is thus responsible for storing the relation between those two classes. As so, the class only has one attribute, the *relation*.

Furthermore, as can be seen in Figure 4.6, the classes *col_resources* and *sing_resources* have a many to many relation, meaning that collection resource can have zero or more single resources, or vice-versa.

Finally, in Figure 4.7 is shown the obtained class model.

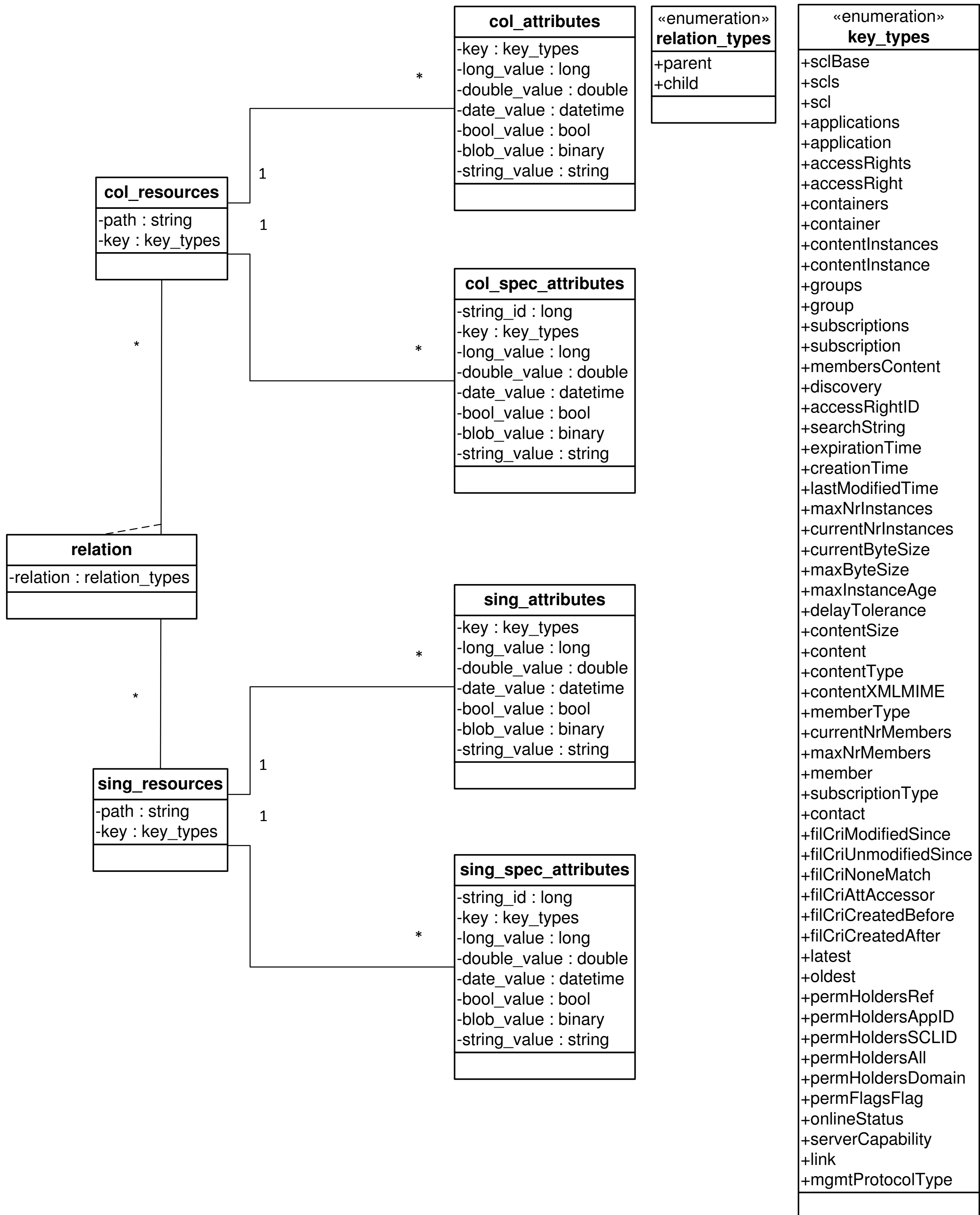


Figure 4.7: Final class model

4.2.1.2 Physical Model

After defining and analysing the information that needs to be supported, and conceptualizing an appropriate structure to store it (class model), the next necessary step is to obtain the physical model, by mapping the previous class model to a physical model.

The physical model is the closest representation that exists of a database, since it already represents the information as relations, attributes and tuples.

Additionally, this section doesn't aim to explain in detail the process of mapping a database class model to the correspondent physical model, but rather give a brief explanation of how it was done.

The final physical model, obtained from the class model, is shown in Figure 4.8. As it can be seen in the final physical diagram, in comparison to the class model, new attributes needed to be added to the relations. These additional attributes, which are the PKs and the FKs, are essential for representing associations (i.e., without those keys, the database wouldn't know how to keep associations between tuples of different relations).

More specifically, a PK is used to uniquely identify a specific tuple in a relation, hence the reasoning for PKs having to be unique.

On the other hand, FKs are used to reference PKs of other relations - see relation *col_attributes* in Figure 4.8, where the *col_resources_id* attribute refers to the *id* attribute of the *col_resources* relation, or to reference the PK of the same relation, in case of a recursive relation - see relation *col_resources*, where the *col_parent_id* attribute refers to the *id* attribute of the same relation.

Besides the addition of the aforementioned attributes, one can easily see another difference, in comparison with the class model, which is, the cardinalities are 1-1..* in the physical model, instead of the 1-* present in the class model. This difference is the result of converting a class model into a physical model, where physical model associations don't have the same meaning as class model relations.

Although the physical model already represents the type of data associated with each attribute, for the less obvious attributes, follows a more detailed explanation for why a specific type of data was chosen:

- ***col_resources*** and ***sing_resources***
  - ***path:*** 256 characters sized string;
  - ***key:*** Integer. Each integer represents a type of resource. For example, 2 represents the resource *scls*, 4 represents the *applications*, 8 represents the *containers* and 10 represents the *contentInstances*. For the complete mapping refer to the class *key_types* in Figure 4.7;
- ***col_spec_attributes*** and ***sing_spec_attributes***
  - ***string_id:*** String. Unique string (id) that identifies the attribute.
- ***relation***
  - ***relation:*** Integer. Each integer represents a type of relation. Number 1 represents that the *col_resources* entry (referred by *col_resources_id*) is the parent of the *sing_resources* entry (referred by *sing_resources_id*), while 2 represents that the *col_resources* entry is a child of the *sing_resources* entry. See the class *relation_types* in Figure 4.7;

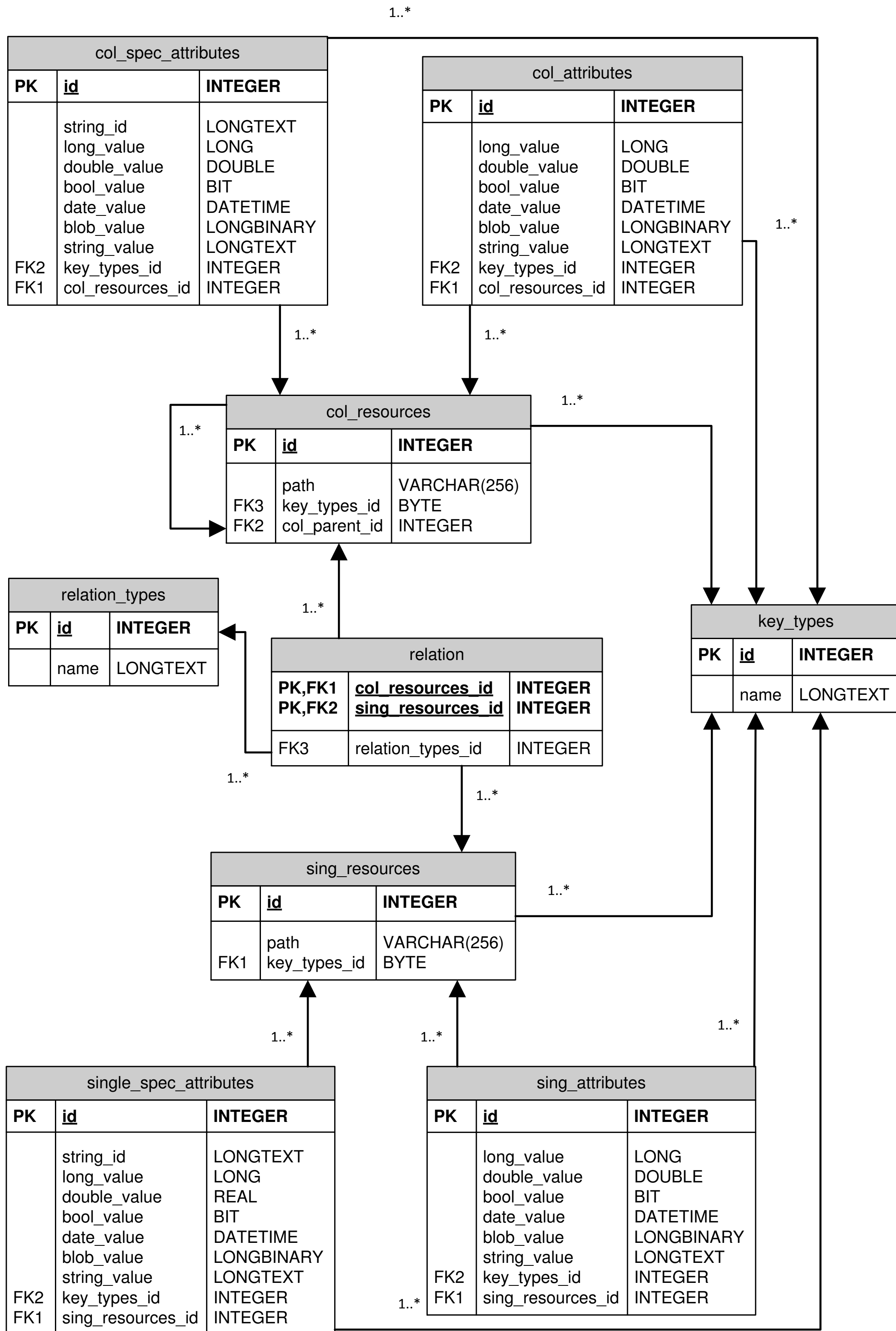


Figure 4.8: Final physical model - implemented in the database

4.3 Gateway's Architecture

Given the degree of freedom conceived to the developers for implementing the standard, it becomes important to provide a better understanding of how the gateway internals were designed and implemented to ensure the desired functionalities, and guarantee that they follow the standard specifications.

As such, this section aims to describe in detail how each component of the gateway, the GSCL component, the GA component and the communication modules, were implemented.

4.3.1 GSCL component

Being the GSCL the core component of the gateway architecture, since it is responsible for providing the gateway's base functionalities, naturally, it is the first component to be presented in this document.

Figure 3.5 (from the Section 3.2.4.3), shows how the standard recommends to logically divide the different functionalities, by the GAE, GGC, GRAR, GSec, GREM, GCS and GIP capabilities.

Even though that division was recommended, and not mandatory, it was adopted for this implementation because it already provided a grouping of related functions, i.e., all functions responsible for providing operations to the NSCL are grouped in the GGC capability, while all functions that store and manage the applications and devices data (including subscriptions and notifications) are grouped in the GRAR capability, for example, as explained in Section 3.2.4.3.

As so, following the above logical division, the packages that compose the architecture, as well as their respective relations, come more or less naturally. The architecture of the GSCL component, divided by packages, is represented in Figure 4.9.

Each package in the diagram, with the exception of the *dataAccess* package, represents a capability of the GSCL, and as such, it's responsible for providing the functionalities well defined in the standard.

The following paragraphs will describe each one of the packages and their respective relations.

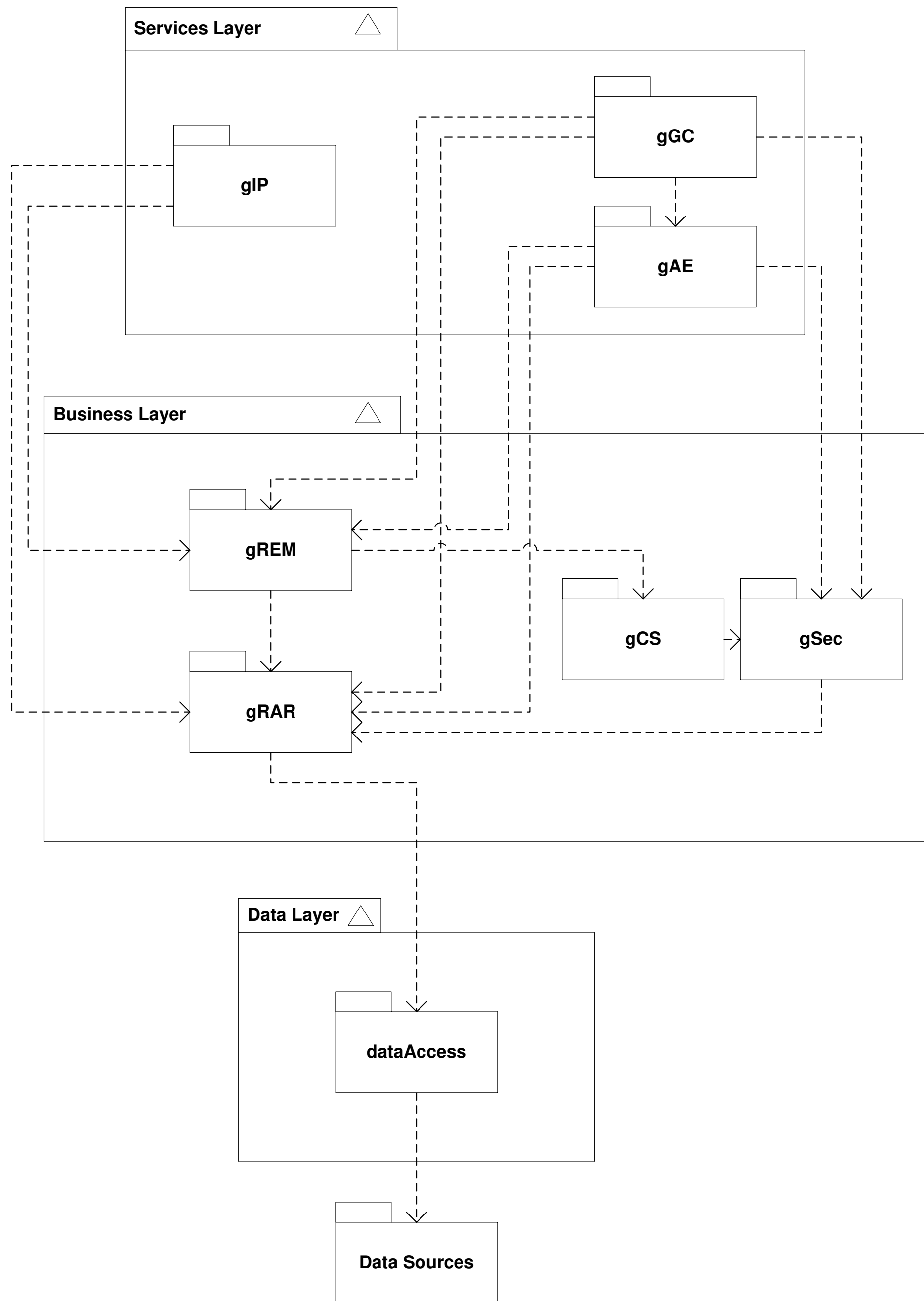


Figure 4.9: Package diagram representing the GSCL architecture

4.3.1.1 *dataAccess Package*

Starting from the bottom of the diagram, the package *dataAccess* represents the layer that provides the access to the database, enabling the GSCL to store, retrieve, update or delete any data. The database model, as explained in section 4.2.1, represents the persistently stored data according to the structures and data types defined in [5] and [38].

Furthermore, in order to interact with the database, this package deals directly with SQL queries, manual handling result sets and object conversions, and thus does not rely in any Object-Relation Mapping (ORM) tools, like Hibernate [48] and TopLink [49]. The choice of not using any ORM tool was taken to avoid having to cope with their 'particularities' and

performance overheads, which wouldn't bring much advantages in this case, given the humble database model.

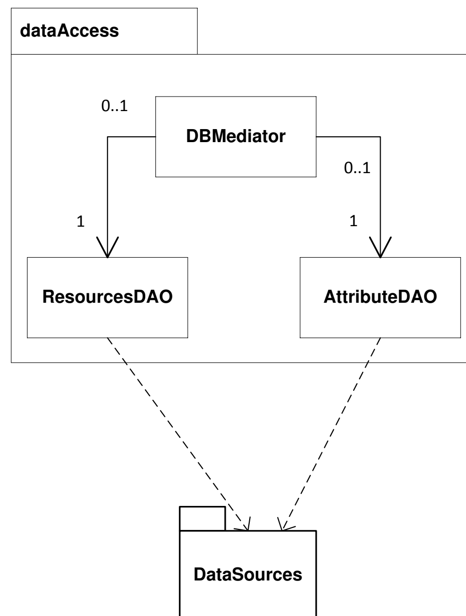


Figure 4.10: Internal representation and respective relations of package *dataAccess*

This package is composed by three main classes, *ResourcesDAO*, *AttributesDAO* and *DBMediator*.

As said, since *dataAccess* accesses the database through direct SQL calls, there needs to exist at least a class that provides a set of methods that executes those SQL calls. In this case, a division was made between the resources and the attributes, originating two classes, the *ResourcesDAO*, which provides the SQL calls that deal with the resources, and the *AttributesDAO*, which has the methods that execute the SQL calls related to the attributes. This approach is based on the Data Access Object (DAO) pattern.

The *DBMediator* is the class that mediates the access to the database connection, providing connections to both of the previous classes, so they can execute their SQL statements. Additionally, it also merges the classes *ResourcesDAO* and *AttributesDAO* functionalities into one, abstracting the access to lower level methods.

Moreover, to help with the mapping between objects's high-level information and database's low-level information, there are three more secondary classes, which are the *ResourceType*, the *RelationType*, and the *Keys* enumerators.

4.3.1.2 *gRAR Package*

The *gRAR* package, which represents the GRAR capability, is responsible not only for managing information about the M2M devices and about subscriptions/notifications, but also for allowing the storage and management of data sent by devices, applications and other SCLs. In order to accomplish that, it needs to have access to the package that enables interactions with the database, hence its dependency on *dataAccess* in the above diagram. In the provided architecture, this package is the only one that interacts with the *dataAccess* package.

Since this package does the mappings between the objects (that contain the information of the resource) and the database, together with *dataAccess*, both these packages form the lowest layer of this GSCL architecture.

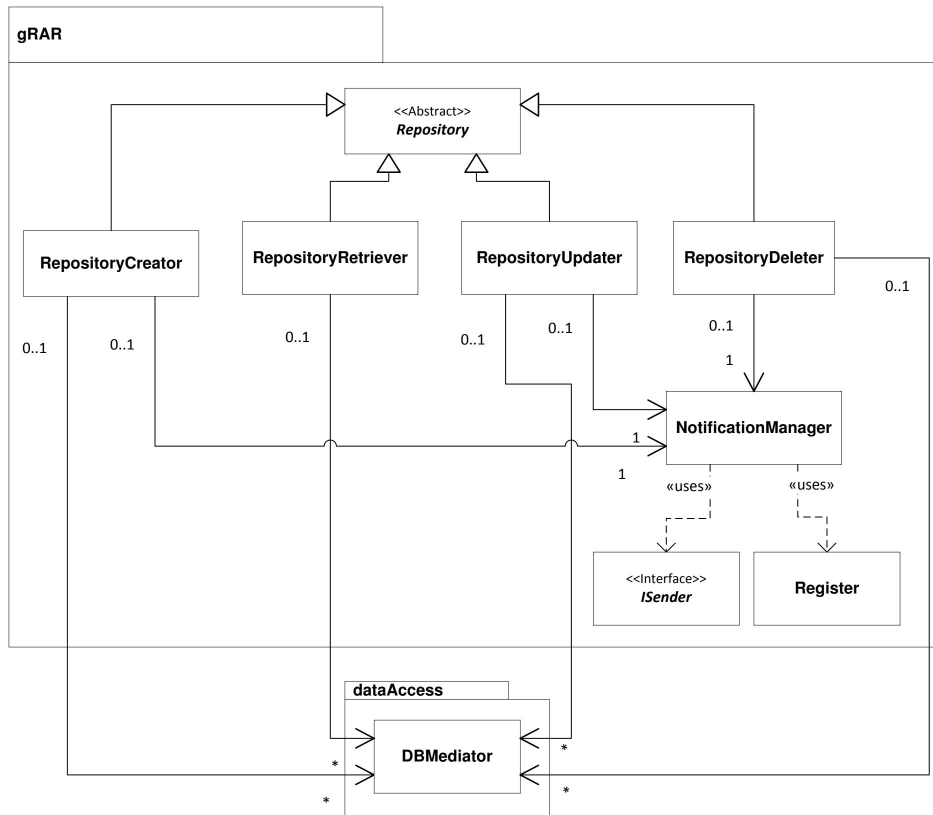


Figure 4.11: Internal representation and respective relations of package *gRAR*

Regarding the storage of information, this package has four classes, divided by the types of operations (CRUD). They are the *RepositoryCreator* class, the *RepositoryRetriever* class, the *RepositoryUpdater* class, and the *RepositoryDeleter* class. These classes provide the necessary methods to store, retrieve, update or delete the collection resources and their respective sub-resources and attributes in the database, thus, for example, if an application resource is passed to the appropriate method in the *RepositoryCreator* class, it is stored in the database. All these classes extend the *Repository* abstract class.

Regarding the management of the stored information, there are two more classes, the *Register* class, and the *NotificationManager* class. Being the latter responsible for processing and sending the notifications resulting from subscribed-to resources, and the former responsible for keeping a record of how to address/reach other entities (applications, devices or NSCLs). The *NotificationManager* resorts to the *Register* class for finding how to reach the subscribers, and to the *ISender* interface for sending the notifications.

Finally, the above classes, the *RepositoryCreator*, the *RepositoryUpdater*, and the *RepositoryDeleter*, depend on the *NotificationManager* class for triggering notifications.

4.3.1.3 *gAE Package*

This package is responsible for exposing the dIa reference point functionalities, which allow registrations, authentications and authorizations of DAs and GAs, as well as validations and routing of requests. Therefore, since it is responsible for routing the received requests to the appropriate GSCL capabilities, for further processing, the *gAE* package has dependencies on most of the existent packages, like *gRAR*, *gREM* and *gSec*.

Particularly, its dependency on *gRAR* allows it to call *gRAR*'s functionalities to process incoming requests, like GAs/DAs registrations or CRUD operations on resources, for example, while its dependency on *gREM* allows devices and applications to interact with the GSCL for remote management purposes. In *gSec*'s case, however, the dependency is not used to process specific requests, but rather authenticating and authorizing the requests before allowing them to interact with any other capability.

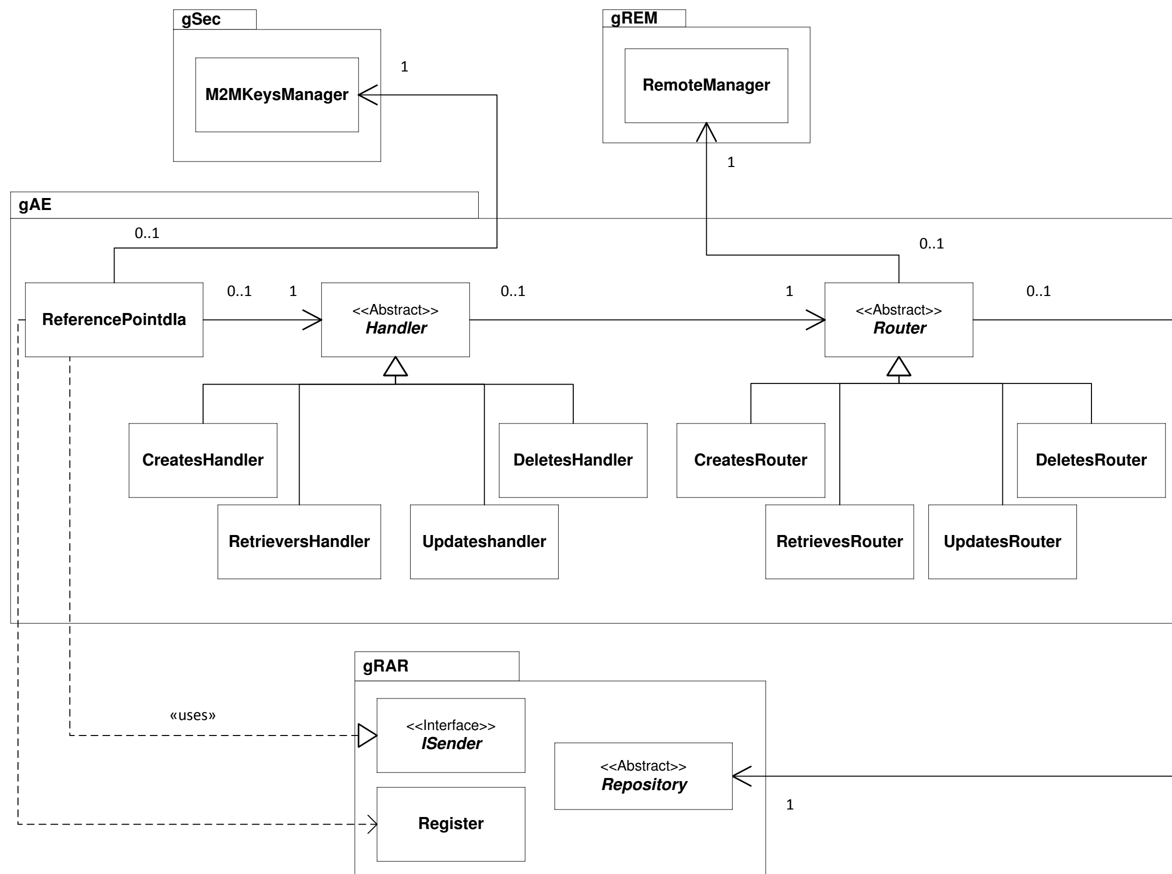


Figure 4.12: Internal representation and respective relations of package *gAE*

Regarding *gAE*, this package is mainly divided in two types of classes, the handlers, and the routers. The handlers are responsible for handling an incoming request, by validating the syntax, the requesting entity, the resource representation and also the permissions, while routers are responsible for routing the request to an appropriate capability, like *gRAR*, for example, that will finish processing the request.

However, since the standard describes different procedures for the different types of requests, i.e., the create, retrieve, update, and delete request, and to take precautions, because this is still not a final standard, there are four handlers, one for each request, the *CreatesHandler*, the *RetrievesHandler*, the *UpdatesHandler*, and the *DeletesHandler*. Each one of these handlers extend the abstract class *Handler*, and are responsible for validating their respective type of requests.

After the validation of the handlers, the request is passed to the appropriate routing class (hence the dependency of *Handler* abstract class on *Router*), which routes the request to the appropriate capability. Just like in the handlers case, there are four types of routers, the *CreatesRouter*, the *RetrievesRouter*, the *UpdatesRouter*, and the *DeletesRouter*, where all extend the abstract *Router* class which, in turn, depends on the *Repository* class in

gRAR package, and also on the *RemoteManager* class from *gREM*, to route possible remote management requests coming from interactions between the GSCL and dIa enabled devices.

Finally, there is also a simple delegate class, named *ReferencePointdIa*, that represents the dIa reference point, and whose only functionality is to receive and forward the received requests, either externally or locally, and register requesting entities. For that, this class has three dependencies. First, it implements the *ISender* interface from the *gRAR* package, creating methods that allow to forward information to dIa enabled devices. Second, it also uses the *Register* class from *gRAR* to register new requesting entities. Finally, it depends on the *M2MKeysManager* class from *gSec* for performing authentication and authorization of the requests.

4.3.1.4 *gGC Package*

The package *gGC*, which represents the GGC capability, is responsible for exposing the mId reference point functionalities. It can be thought as a superset of the GAE capability, since it takes advantage of GAE's capabilities - hence its dependency on the *gAE* package, but provides even more functionalities, like more complex remote management scenarios, encryption/integrity protection on data exchanged with the NSCL, reporting of transmission errors, transport session establishment and teardown, and ensures secure delivery of application's data to/from the GSCL to the NSCL. To provide all the security related functionalities, the *gGC* package depends on *gSec*, in order to get the necessary security functionalities. Finally, it also depends on the *gREM* package for forwarding remote management incoming requests.

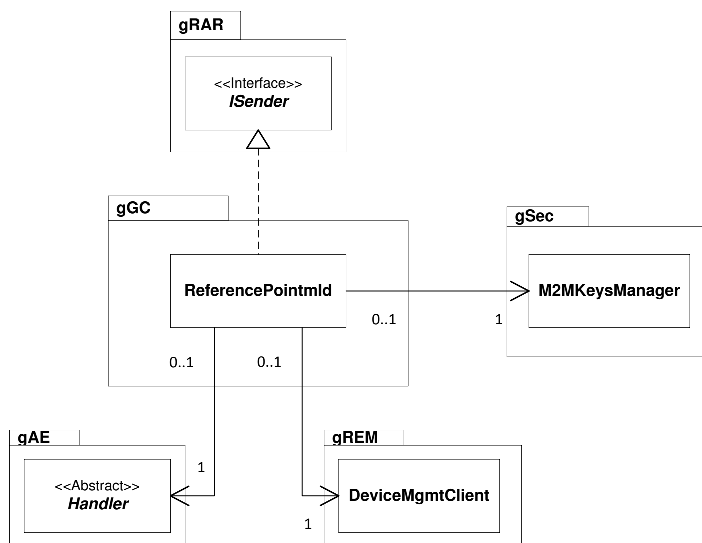


Figure 4.13: Internal representation and respective relations of package *gGC*

The *gGC* package, equivalently to the *gAE* package, has the class *ReferencePointmId*, which has the same role that the class *ReferencePointdIa* has, that is, to represent the mId reference point.

Meanwhile, unlike what happens in package *gAE*, the package *gGC* has no handler classes. The *ReferencePointmId* only delegates the requests to the handlers of the package *gAE*, since there is no difference in how the requests are processed in the dIa reference point and in the mId reference point. The only differences are related to security and remote management.

Regarding security, the difference between dIa and mId resides in the way how authorizations and authentications are done, which can be different among the two reference points,

thence the dependency of *ReferencePointmId* class on *M2MKeysManager* class from package *gSec*. Furthermore, the mId reference point (still through the use of *M2MKeysManager*) also provides the establishment of transport sessions, their teardown, key negotiations, and encryptions.

Regarding remote management, as said in Annex E of [5], the mId reference point has to provide interfaces for supporting interactions between the remote management server, in the NSCL side, and the remote management client, in the GSCL side. That is the reasoning behind the dependency of *ReferencePointmId* on *DeviceMgmtClient* of *gREM*.

Finally, it also implements the *ISender* interface from the *gRAR* package, creating methods that allow to forward information to entities on the other end of the mId reference point.

4.3.1.5 *gSec Package*

As a first responsibility, this package has to perform the service bootstrap of the GSCL, where the necessary credentials will be provided - These credentials include Keys that will not only be used to authenticate and register the GSCL with the NSCL, but also for deriving other important keys that will allow the gateway to operate securely.

Secondly, it is also responsible for performing the connection procedures, in order to authenticate (using the credentials obtained during the bootstrap) both ends of the mId reference point, that is, the GSCL and the NSCL.

Regarding dependencies this package has only one dependency, on package *gRAR*. This dependency is necessary to allow the registry of relevant parameters obtained during the service bootstrap and the connection procedures, like the SCL-ID or NSCL points of contact.

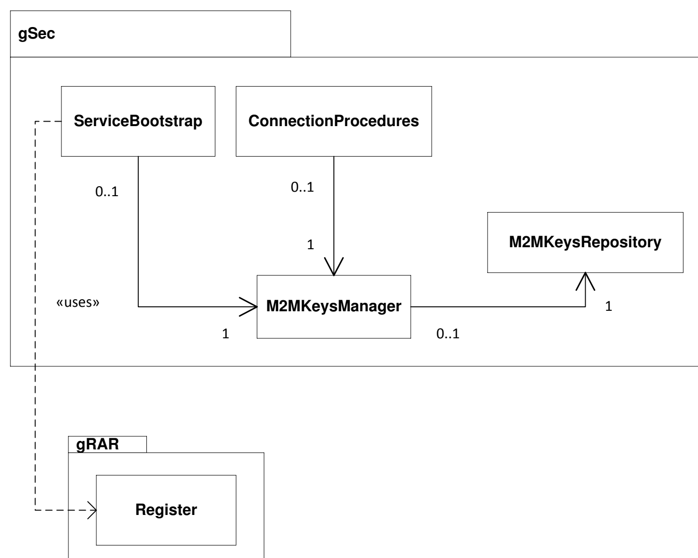


Figure 4.14: Internal representation and respective relations of package *gSec*

This package is composed by the classes *ServiceBootstrap* and *ConnectionProcedures*, which together are mainly responsible for bootstrapping the Root key into the gateway, perform mutual authentication of mId endpoints, and ensure M2M Connection keys agreement.

The *ServiceBootstrap* class stores some of the obtained M2M parameters, like the M2M Root key, in the *M2MKeysRepository* class, through the use of the *M2MKeysManager* class, and others, like a list of possible NSCL points of contact is registered in the *Register* class of package *gRAR*, for addressing purposes.

Regarding the *ConnectionProcedures* class, it also needs to access the *M2MKeysManager*, in order to have access to the M2M Root key, and be able to perform the necessary procedures. The information derived from these procedures is also stored in the *M2MKeysRepository*, through the use of the *M2MKeysManager* class.

4.3.1.6 *gCS Package*

In turn, the *gCS* package is used to enable the connection of the gateway with the network access layer. As said in Section 3.2.4.3, the GCS capability is only responsible for two things, i) providing network selection, when multiple networks are available, ii) providing alternative communication, when failures happen using the current network (by choosing another network, for example).

However, in the GSCL architecture proposed in this document, the *gCS* package, besides the previously mentioned functionalities, also provides another functionality, one that it's not contemplated by the standard. That functionality is to allow the gateway to bootstrap and register itself in the access network, through a priori provided configurations.

Finally, the package *gCS* depends on the *gSec* package. This dependency reflects the case of when the bootstrap procedure is assisted by the access network, and as a result, the necessary credential keys are provided to the *gSec* package by the *gCS* package.

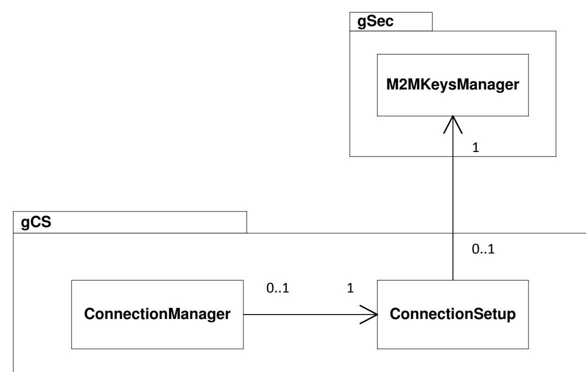


Figure 4.15: Internal representation and respective relations of package *gCS*

Used to provide a way to select, establish and manage network connections (to interact with the NSCL), this package has two classes, the *ConnectionSetup* and the *ConnectionManager* classes.

While the class *ConnectionSetup*, as the name implies, is responsible for establishing the connection with a previously selected and configured network, the class *ConnectionManager* is responsible for managing the connection. If the connection drops, it is responsible for calling the necessary procedures in the *ConnectionSetup* class, so that the connection gets reestablished, or if not possible, another network gets chosen to establish a new connection - thence the dependency of *ConnectionManager* on *ConnectionSetup*.

Finally, regarding the class *ConnectionSetup*, it depends in the class *M2MParametersManager* of *GSec* for the storage of the credential keys that will be used during the bootstrap, in case it's assisted by the access network.

4.3.1.7 *gREM Package*

Given that the package *gREM*, in the below diagram, provides the GREM functionalities, it is responsible for providing remote management of the gateway itself and also act as a remote

management proxy for the devices connected to the gateway.

Thus, this package depends on the *gRAR* package, for storing appropriate information, and on the *gGC* package, for enabling this package to control the connectivity of the gateway, for the purpose of remote entity management.

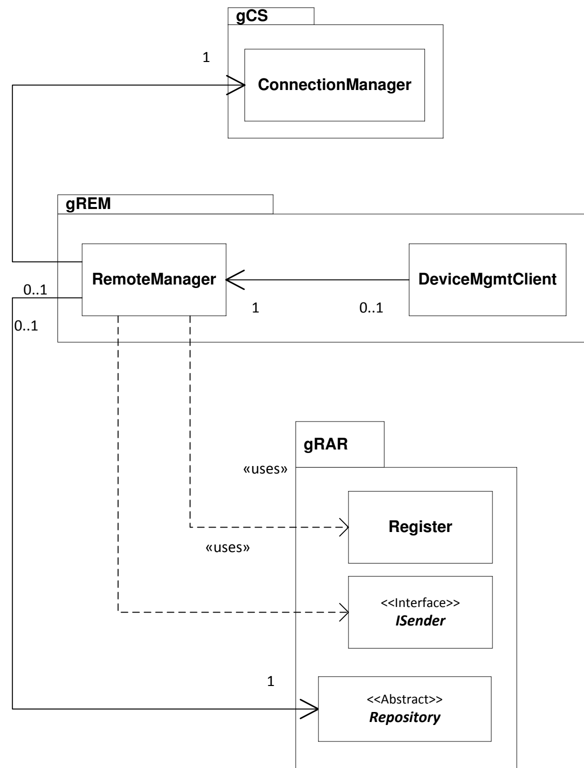


Figure 4.16: Internal representation and respective relations of package *gREM*

The package, has two classes, the *DeviceMgmtClient* and the *RemoteManager*.

The *DeviceMgmtClient* class executes the commands sent by the remote management server, present in the NSCL, in the gateway itself. Additionally, it also interacts with the class *RemoteManager*, in order to perform its procedures, like creating, retrieving, updating or deleting corresponding *mgmtObj* and *mgmtCmd* resources in NSCL, via the mId reference point, or even connect the gateway to other networks. The class *RemoteManager*, in turn, besides the previous functionalities, also provides a way to manage devices in the dIa reference point.

4.3.1.8 *gIP Package*

Finally, regarding *gIP*, it's a package that represents the modules that can be created in GSCL for allowing the gateway to interact with devices non compliant with the ETSI M2M standard. Thus, *gIP* modules can be used to translate ZigBee, Bluetooth or Z-Wave requests, for instance, and map them to appropriate functionalities of the GSCL, allowing their interaction with the GSCL as if those request had come from devices compliant with the standard.

Therefore, this package needs to depend on *gRAR*, for being able to store the data coming from the translated requests, and also on *gREM*, for allowing the remote management of ETSI M2M non compliant devices.

gIP is only composed by only two classes. The *Receiver* class and the *Translator* class.

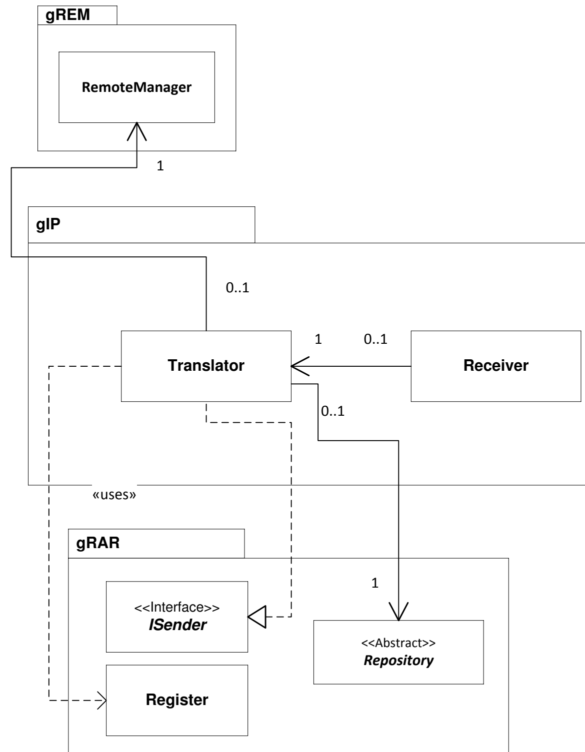


Figure 4.17: Internal representation and respective relations of package *gIP*

The *Receiver* class is responsible for, as the name implies, receiving the information. Here, receive can be something from reading the information from a text file, to reading the information from a serial cable connected to a base station, for example. This class then sends the received information for translation in the *Translator* class, thus its dependency on it.

On the other hand, the class *Translator* does the translation between the legacy information, received from the *Receiver* class, and appropriate resources to be used for calling functionalities in GSCL. The called functionality is based on the translation. Since *gIP* can call almost any functionality, it depends on the *Repository* and *Register* classes of the *gRAR* package for creating, updating, retrieving, deleting resources, and registering new requesting entities, and also depends on the *RemoteManager* class of the *gREM* package, for remote management purposes of the legacy devices.

Finally, still regarding *gRAR* it also implements the *ISender* interface, creating methods that allow to forward information to legacy devices.

4.3.2 GA component

As explained in Section 3.2.4.2, GAs are the applications that reside inside the gateway, being able to access the different functionalities provided by the GSCL through the dIa reference point.

As an example, a GA could be a simple application responsible for making a conversion of temperature, sent by a particular set of sensors in Fahrenheit unit, to the Celsius metric unit. A graphic representation of the previous example can be seen in Figure 4.18.

In this implementation, GAs are built by implementing the interface class *IGApp*, in Figure 4.19. As can be seen, the GA component doesn't have a defined architecture as the

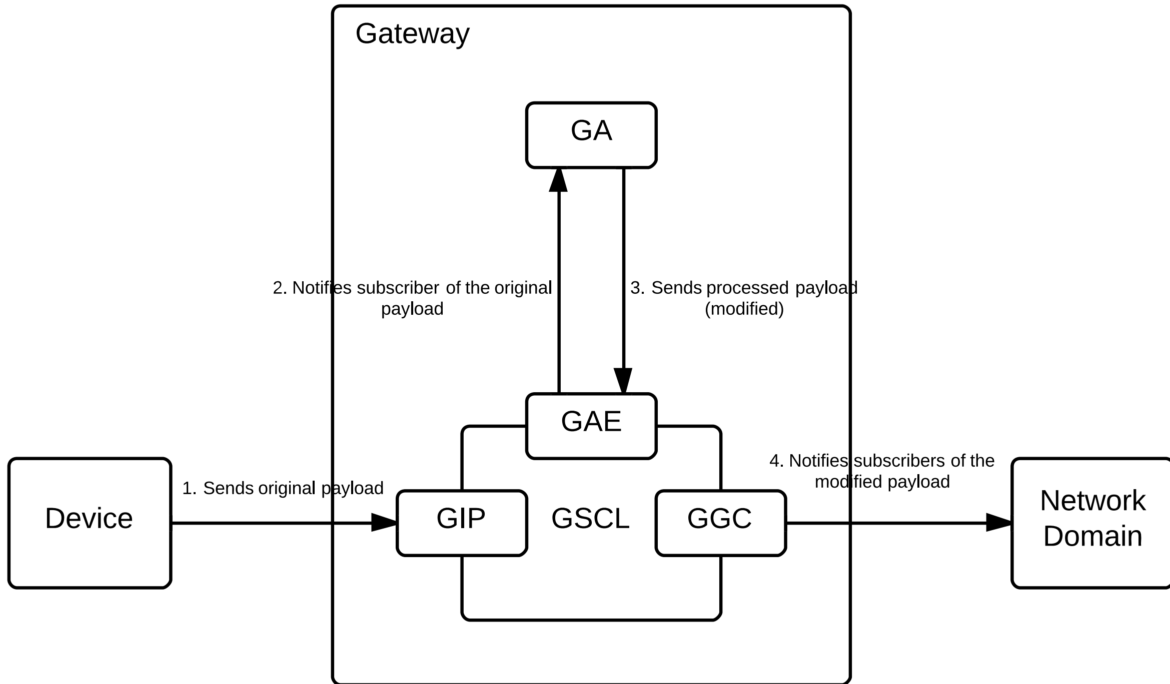


Figure 4.18: Use case example of a GA

GSCL, nor it would made sense, since there isn't one architecture that would fit the needs of all possible applications. Thus, to develop and run a GA, in this gateway, developers need first to implement an interface with a pre-defined structure. After that, and despite that initial structure, they are free to implement the GA the way they find most convenient.

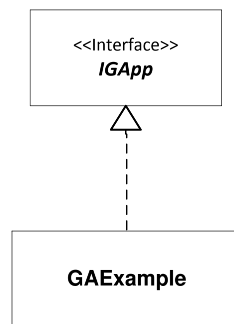


Figure 4.19: Example of a GA

After the development of the application, it's only necessary to put the GA in the *gApps* folder, present at the root of the installation. The GSCL, using reflection, will then load and run the GA.

A snippet of the schema of a GA is shown in Snippet 1.

```
package pt.it.av.apollogw.gApps;

public class Template implements GApp {

  @Override
  public void run() {}

  @Override
  public void
  notification(SubscriptionNotifyRequestIndication subNotify) {}
}
```

Snippet 1: Template of a gateway application

The method *run* in the above snippet is the main method of a GA, since it's the first to be called when a GA starts. The method *notification*, on the other hand, is called, with the appropriate notification resource passed as an argument, whenever a notification to a particular subscribed resource arrives.

In conclusion, the above paragraph just showed why GAs need to implement a interface with a well defined structure, which is, to be able to start running a GA and to deliver notifications, the GSCL must know which function to call (like the method *main* in a Java class).

4.3.3 Communication modules

For interacting with the outside world, that is, to provide a way to transport the operation primitives to remote entities, the gateway provides two modules, the HTTP and the CoAP bindings, which are the main bindings recommended by the standard.

Regarding the HTTP bindings, as visible in Figure 4.20, the package *HTTPComm* is the one that provides those binds. Regarding the package internals, while *HTTPProxy* class is used to receive the incoming requests and forward them to the appropriate associated reference point, the class *HTTPClient* is used to make HTTP requests, mapping the primitives to a HTTP request as defined in the standard, and forwards them to a specific entity.

The package *CoAPComm* in Figure 4.21, that provides the CoAP bindings, is very similar to the package *HTTPProxy*, since it also has two classes, named *CoAPProxy*, which has the same functionality as the class *HTTPProxy*, and *CoAPClient*, equivalent to the class *HTTPClient*.

Figure 4.20 and Figure 4.21 show that both proxies depend on the *IReceiver* interface of the package *gAE*, which, as said before, is implemented by the classes *ReferencePointdIa* and *ReferencePointmId*. This way, after mapping the incoming HTTP/CoAP requests into appropriate primitives, the proxies forward them to the associated reference point.

On the other hand, the client classes implement the *ITransmitter* interface present in the package *HTTPComm*. As shown in Figure 4.20, the classes *ReferencePointdIa* and *ReferencePointmId* take advantage of the interface for forwarding the primitives, in HTTP or CoAP.

These dependencies in the *IReceiver* and *ITransmitter* classes are necessary to avoid linking a reference point (*ReferencePointdIa or ReferencePointmId*) to a specific communication module (*HTTPComm* or *CoAPComm*). Thus, upon starting the application, it can be defined if the reference point dIa is going to support HTTP or CoAP, and ditto for the mId.

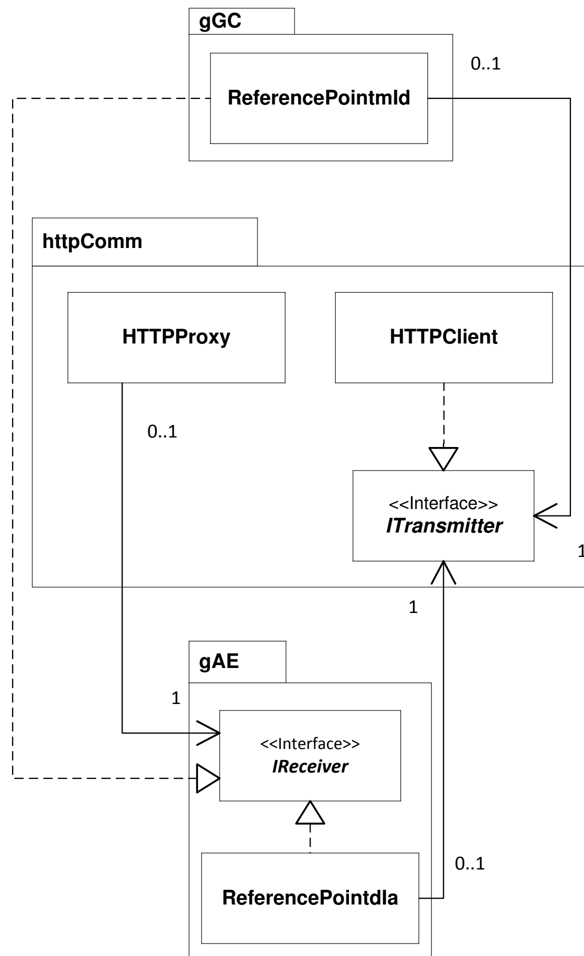


Figure 4.20: Internal representation and respective relations of package *httpComm*

At last, both the proxies and clients also depend on the *MarshallManager* package, in order to map the XML/JSON contents (resources) of HTTP/CoAP requests to the appropriate resource objects, and vice-versa.

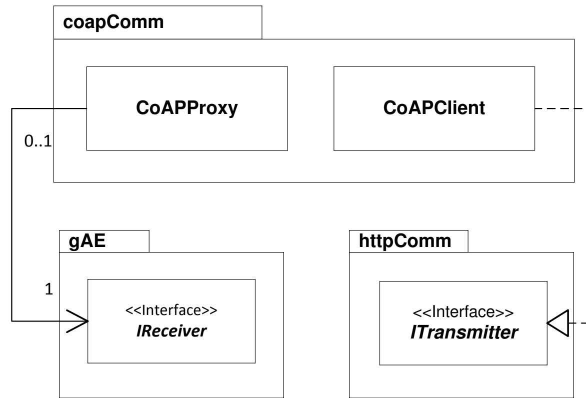


Figure 4.21: Internal representation and respective relations of package *coapComm*

4.4 Procedures

In chapter 9.3 of [5] is presented a theoretical high-level description of the steps involved in the execution of each procedure, from the issuer to the receiver.

Contrary to that, the purpose of this section is rather to give a more detailed explanation of how those procedures are executed internally in the GSCL architecture implemented for this Dissertation.

Moreover, for brevity, this section will not describe every single procedure implemented, but only the procedures that are more relevant in a normal use case, as the deployment scenario described in the next chapter. The explanation of these procedures provides a good notion of how the different capabilities interact with each other.

4.4.1 Create Application

This clause describes how the GSCL deals internally with the creation of an application resource. This explanation assumes that the request is sent by a GA.

Step 01: The issuer (a GA) sends an application create request primitive to the GSCL, through the dIa reference point. That primitive identifies the requesting entity, that is, the application that is requesting the creation of the resource, the path where the resource shall be created and the resource representation itself;

Step 02: The *ReferencePointdIa* class receives the primitive request and passes it to the method *processRequest* of the class *CreatesHandler*, for processing. Based on the type of request, that method will delegate that request to even another method, named *processAppRequest*, of the same class, that will process it accordingly. The processing of the request includes checking the syntax of the request, the correctness/validity of the requesting entity, the existence and validity of the resource representation, and also check if the requesting entity has the access permissions for the operation;

Step 03: After the verifications, and in case they are succeeded, *procccessAppRequest* method calls the class *CreatesRouter* method's *routeRequest*, passing the target path and the application resource to be created as parameters;

Step 04: The *routeRequest* is then responsible for calling the method *createApplication* of the *RepositoryCreator* class, with the target path and the resource as parameters;

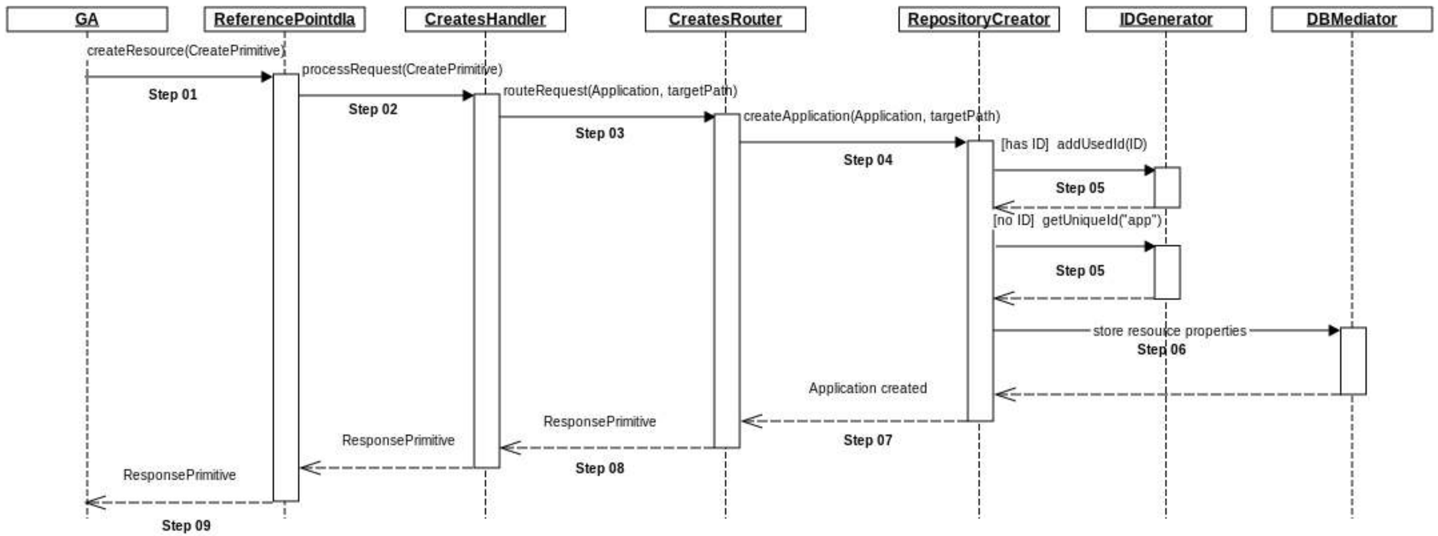


Figure 4.22: Sequence diagram of a create resource.

Step 05: The *createApplication* method will first check if the passed resource has an ID or not. In case it hasn't, it calls the method getUniqueId() of the class *IdGenerator*. Otherwise, it calls the method addUsedId(), of the same class, to check if the provided ID is unique and can be added to the list of used id's;

Step 06: Secondly, through calls to the *DBMediator* class, the *createApplication* method stores, in the database, all the attributes and sub-resources of the application resource under the provided target path;

Step 07: After storing the resource, it returns with the target path of where the resource has been stored.

Note: if some error occurs during the steps 4 and 5, a custom exception is generated;

Step 08: *routeRequest* receives either an exception with an error message, or the target path of where the resource has been stored. In both cases it is responsible of generating the response primitive, of success, or insuccess (with the appropriate error message), and returning it;

Step 09: Finally, when the primitive returns to the class *ReferencePointdIa*, a response is sent to the GA request.

4.4.2 Retrieve ContentInstance

This clause will be very similar to be above Create Application clause, since the major change is in the primitive type of the request.

Step 01: The issuer (GA) sends a content instance retrieve request primitive to the GSCL, through the dIa reference point. As in the previous case, that primitive also identifies the requesting entity and the target path (although in this case the target path refers to where the resource should be retrieved from);

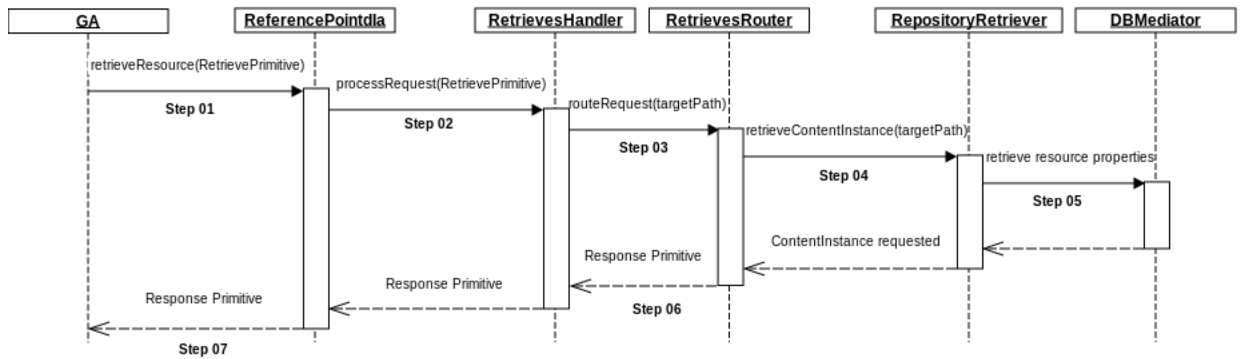


Figure 4.23: Sequence diagram of a retrieve resource.

Step 02: After receiving the request, the *ReferencePointdIa* class sends it to the method *processRequest* of the *RetrievesHandler* class, for processing. That method will then check the type of request and delegate it to another method, named *process-ContInst*, that will process it accordingly. The processing of the request includes checking the syntax of the request, the correctness of the requesting entity, and also checking if the requesting entity has the right permissions for the operation;

Step 03: After validating the request, it calls the class *RetrievesRouter* method's *routeRequest*, with only the target path as a parameter;

Step 04: The *routeRequest* is then responsible for calling the method *retrieveContentInstance* of the *RepositoryRetriever* class, using only the target id as a parameter;

Step 05: The *retrieveContentInstance* method will gather, through method calls to the *DBMediator* class, all the attributes and sub-resources of the resource that resides under the provided target path.

After getting the complete resource, returns it.

Note: if an error occurs during this step, a custom exception is generated;

Step 06: *routeRequest* receives either an exception with the error message, or the resource that has been retrieved. In both cases it is responsible for generating the response primitive, of success, that contains the retrieved resource, or insuccess, with the error;

Step 07: Finally, when the primitive returns to the class *ReferencePointdIa*, it is sent as a response to the GA request.

4.4.3 Update Subscription

This clause will also be very similar to be above Create Application and Retrieve ContenInstance clauses.

Step 01: The issuer (GA) sends a subscription update request primitive to the GSCL, through the dIa reference point;

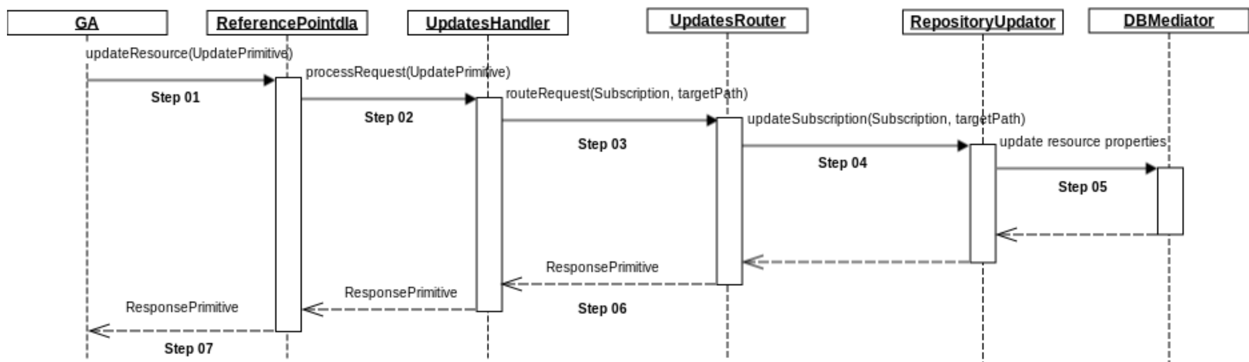


Figure 4.24: Sequence diagram of a update resource.

Step 02: *ReferencePointdIa* receives the request and delegates it to the method *processRequest* of the *UpdatesHandler* class, for processing. That method will check the type of request and send it to even another method, named *processSub*, that will process it accordingly. The processing of the request includes, checking the syntax of the request, the correctness of the requesting entity, the existence and validity of the resource representation, and also check if the requesting entity has the right permissions for the operation;

Step 03: After those verifications, it calls the class *UpdatesRouter routeRequest* method, using only the target target and the updated resource as parameters;

Step 04: The *routeRequest* is then responsible for calling the method *updateSubscription* of the class *RepositoryUpdater*, using the target path and the updated resources as parameters;

Step 05: To update the resource information, the *updateSubscription* method makes calls to the *dbMediator* class.

Note: During the update of the resource (step 4), if some error occurs, a custom exception is generated;

Step 06: *routeRequest* receives either an exception with the error message, or nothing (void). In both cases it is responsible of generating the response primitive, of success, or insuccess, with the error message, and return it;

Step 07: Finally, after receiving the response primitive, the *ReferencePointdIa* sends it as a response to the GA request.

4.4.4 Notify

The notification is only triggered by the create, update and delete procedures, since the retrieve procedure doesn't modify any resource.

Thus, this example will be based on the Create Application procedure that was described above, imagining that a subscription had been done in the parent Applications resource.

Step 1: The method *registerSub* of *NotificationManager* class is called with a target path and an old version of the resource as parameters. The target path is used to

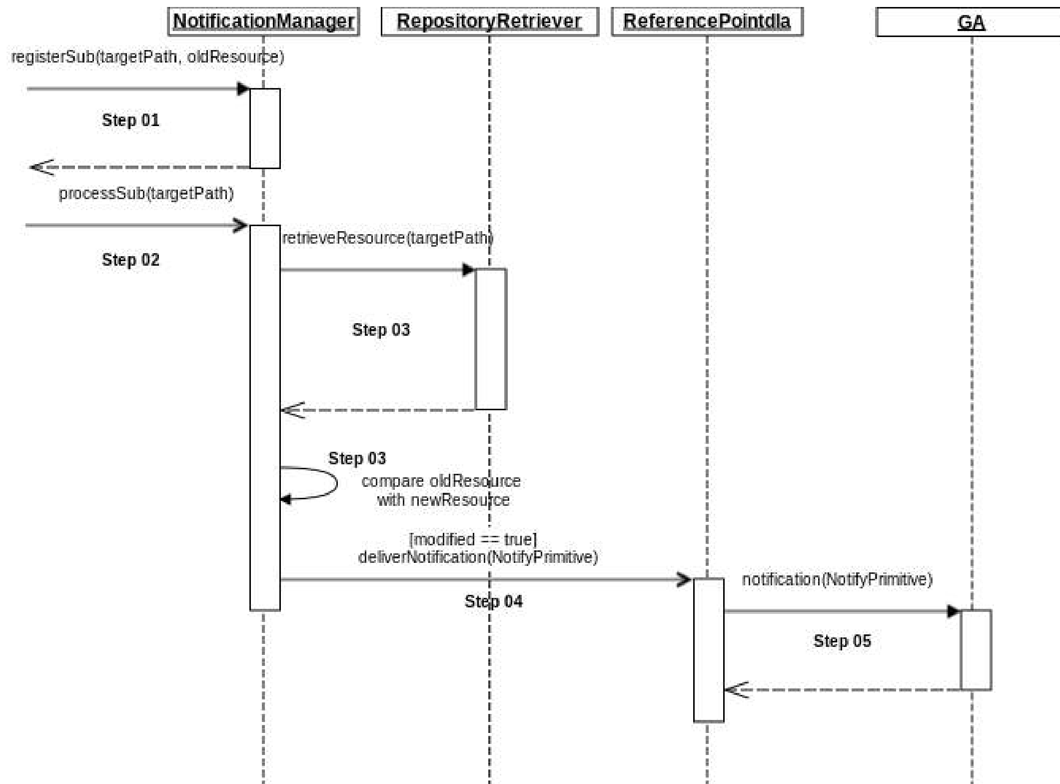


Figure 4.25: Sequence diagram of a notification resource.

identify that that target is registered for notifications. The old version of the resource is also associated to the target path, so that it can be referenced later;

Step 2: Afterwards, the method *processSub* is called with the target path as a parameter. This marks the target path as a candidate to be processed;

Step 3: When a target path becomes a candidate, the method process starts to handle the subscriptions for that notification. It compares the old version of the resource with the one just modified, and according to the filter criteria of each subscription, it can trigger a notification or not;

Step 4: If there are modifications that match the filter criteria, it means that a notification must be sent to that respective subscriber. As a result, a primitive iss constructed with the notification;

Step 5: Finally, the method *deliverNotification* of the class *ReferencePointdIa* is called with the notification primitive as a parameter, which in turn delivers the notification to the appropriate destiny.

Chapter 5

Evaluation and Results

5.1 Deployment Scenario

The gateway implemented for this Dissertation, as said previously in Sections 1.1 and 4.1, was developed as part of the Project Apollo.

As one of the project's milestones, a demo, scheduled for July of 2013, was performed at Coimbra, in a greenhouse of Escola Superior Agrária de Coimbra (ESAC). The purpose of the demo was to deploy and test in a real scenario the software and hardware developed until then.

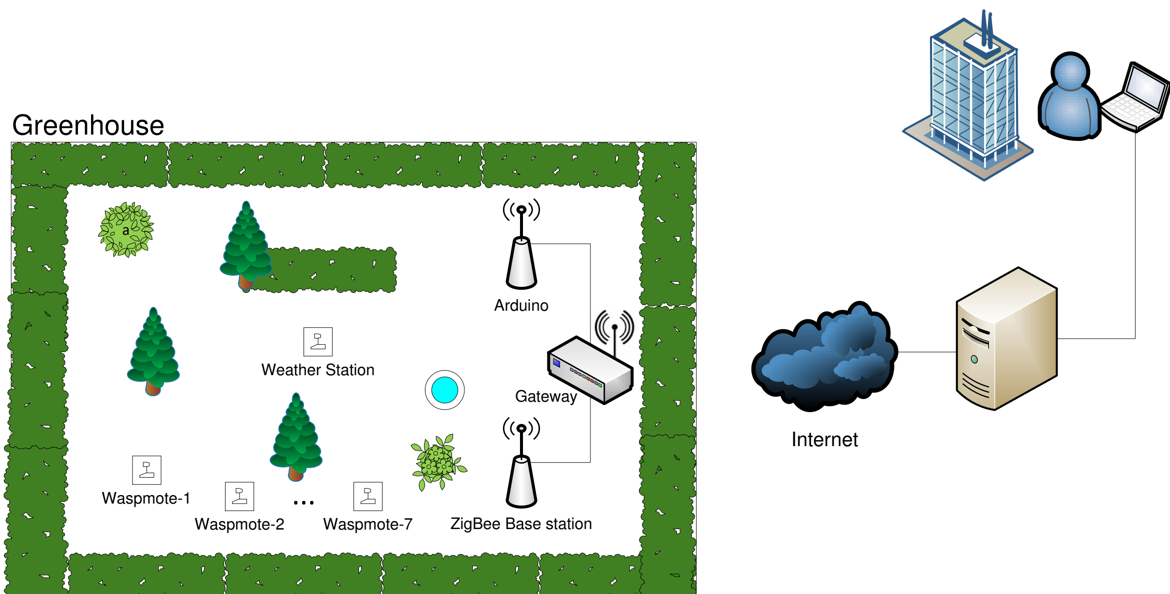


Figure 5.1: Demo Illustration

Figure 5.1 shows the testbed mounted for the purpose of the demo. In this testbed, there were 7 Waspmotes and 1 Arduino sending data to the gateway, which was running on a Alix 3D2 board with a 500 MHz CPU (AMD Geode LX800) and 256 MB of DRAM.

Regarding the gateway, which is the only part that will be addressed in this section, the GSCL and two GAs were installed for the demo.

Moreover, the GSCL used for this demo had one GIP implemented to deal with the Arduino legacy device, i.e., non compliant with ETSI M2M standard. The Arduino device

received data wirelessly, from the weather station installed in the greenhouse, and relayed that data through a serial over USB port to the gateway. Thus, the information that arrived to that serial port was processed by the GIP, responsible for receiving the weather station data and store it appropriately in the GSCL.

To receive the data sent by the Waspmotes devices, no modification needed to be done in the GSCL, since those devices were compliant with the standard (as explained in section 3.2.3, they were devices of the type case 2). The protocol used to transport the information from the devices to the gateway was CoAP. Thus, as soon as the CoAP proxy received a CoAP request, it would process it, mapping the request to an appropriate primitive and deliver the primitive to the GSCL, through the dIa reference point, for processing.

Finally, the two GAs installed together with GSCL were programmed to subscribe to the data sent by the devices, one GA subscribed to the data sent by the Arduino device, while the other subscribed to the data sent by the Waspmote devices. Afterwards, when the data sent by the devices arrived to the GSCL, a notification would be generated and sent to the appropriate subscribed GA, which would send that data, through an HTTP request, to a platform where it could be graphically visualized. However, those HTTP requests were not done through the mId reference point, using the primitive mappings defined in the standard, but rather using simple JSON POSTS, since there was no NSCL available at the moment.

5.2 Results

Testing the implementation of the gateway in real cases, or at least in scenarios close to what might be real cases, is crucial to check the gateway's ability to adapt to a high variety of environments, as well as to uncover cases where the gateway misbehaves.

Despite of using the already deployed demo (described in the previous section) to test the gateway, other tests were also performed in order to test it in more demanding scenarios. Not only in terms of processing, but also in terms of communications.

Thus, this section aims not only to present the results obtained during the tests performed to the gateway, but also provide an insight over the obtained results.

At last, the tests performed against the gateway were:

- Measure memory usage;
- Measure CPU usage;
- Measure how long the GSCL takes to process a notification, given different numbers of loads;
- Measure how long the GSCL takes to process a request, given different numbers of loads;
- Measure the time taken to serve a CoAP request, as well as the overhead involved.

5.2.1 CPU & Memory Usage

For this test, the monitoring of the memory and CPU used by the application was done in the demo already described before. Thus, for a continuous period of ~6 hours, the gateway Alix 3D2 was continuously receiving data from 4 devices, more specifically, three Waspmotes and one Weather Station. The samples were taken with intervals of 7.5 minutes between each other.

Starting with the CPU, Figure 5.2 shows the CPU usage along the 6 hours.

From the graph in Figure 5.2, one can see that in overall Alix's simple CPU didn't got overloaded. Its worst period, in terms of CPU usage, was during the start of the application,

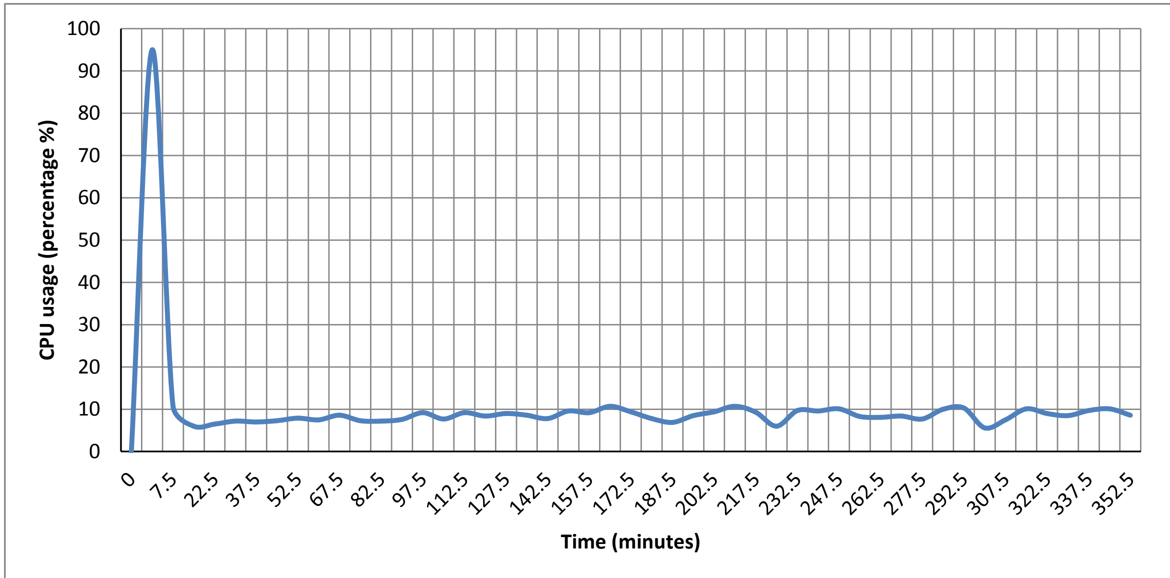


Figure 5.2: CPU usage

where reached 95% of load in just a couple of seconds. However that kind of load is normal, since it's not only when the GSCL starts all needed threads and loads the GAs (two of them, in this test), but also when it interacts with the database, proceeding with the necessary initializations. The load of the JVM, although not directly related to the GSCL, has also to be taken into account, since it also consumes CPU. The rest of the graph is rarely above the 10%, which only happened 7 times in six hours. While the number of requests wasn't high, with approximately 5 request per minute in this particular test, the processor behaved according to the expected.

Regarding memory, the graph that shows the memory usage along the 6 hours is present in Figure 5.3.

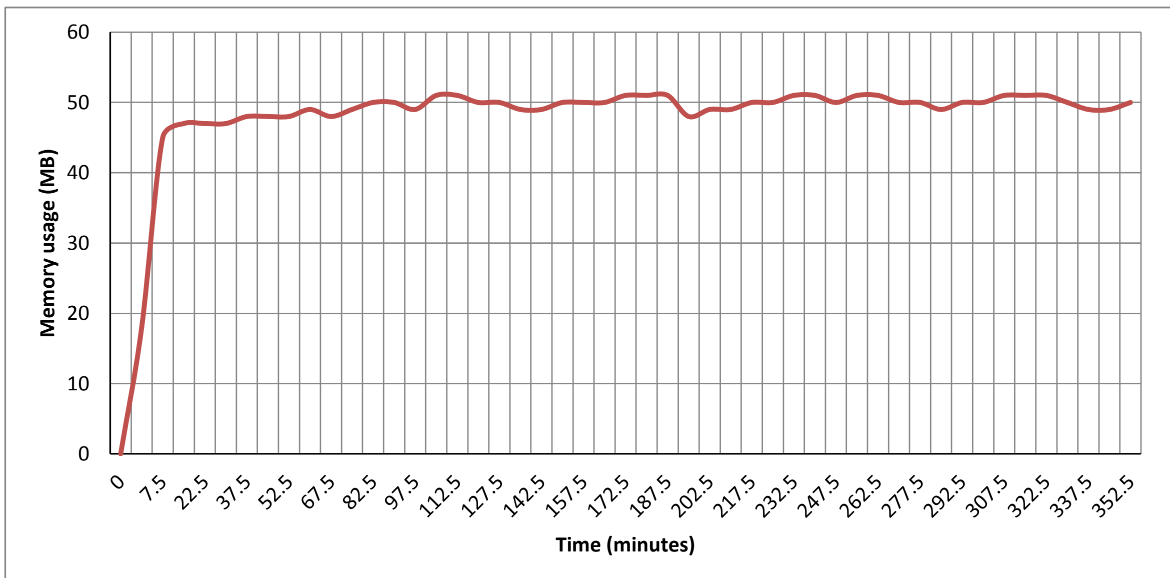


Figure 5.3: Memory usage

Just like the CPU load, the memory also increases exponentially in the first moments

after executing the application, for the same reasons presented when describing Figure 5.2. Afterwards, the memory keeps increasing until it stabilizes around the 50 MB. During the six hours, the maximum memory used by the application was 51 MB, and considering that despite the GAs and the GSCL threads, there were also a HTTP and a CoAP embedded servers running, for HTTP and CoAP interactions, a total of 51 MB of used memory seems to be a reasonable value.

5.2.2 Notifications measurement

To test how GSCL's notifications functionality would behave under different loads, a second test setup was used for measuring the time taken by the GSCL to trigger, compose, and deliver the appropriate notifications. The reason was, since the setup used for the previous section, for measuring the CPU and memory usage, wasn't much request-hungry, i.e., it didn't made requests in a very high rate, a second setup had to be used for simulating a more heavier load. Thus, using a threaded CoAP application developed to deliver a specific number of requests, under different concurrency levels, allowed to better test the limits of the application.

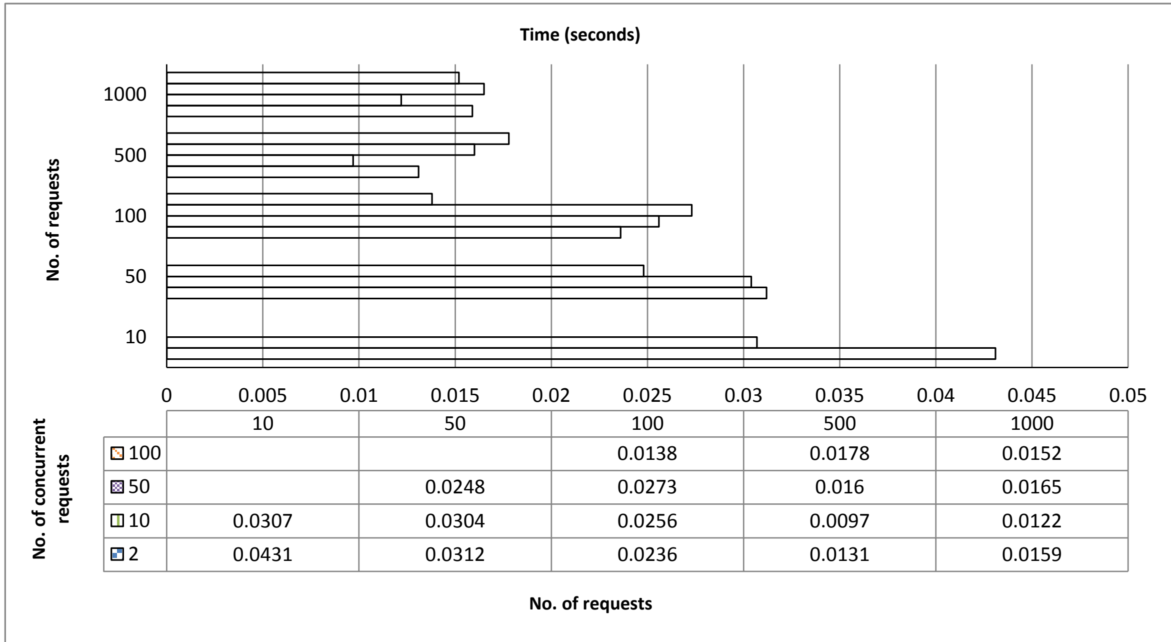
Therefore, the tests performed are enumerated below:

- 10 requests with a concurrency level of 2;
- 10 requests with a concurrency level of 10;
- 10 requests with a concurrency level of 50;
- 10 requests with a concurrency level of 100;
- 50 requests with a concurrency level of 2;
- 50 requests with a concurrency level of 10;
- 50 requests with a concurrency level of 50;
- 50 requests with a concurrency level of 100;
- 100 requests with a concurrency level of 2;
- 100 requests with a concurrency level of 10;
- 100 requests with a concurrency level of 50;
- 100 requests with a concurrency level of 100;
- 500 requests with a concurrency level of 2;
- 500 requests with a concurrency level of 10;
- 500 requests with a concurrency level of 50;
- 500 requests with a concurrency level of 100;
- 1000 requests with a concurrency level of 2;
- 1000 requests with a concurrency level of 10;
- 1000 requests with a concurrency level of 50;
- 1000 requests with a concurrency level of 100.

By analysing the previous list one can see that, the load of requests that the GSCL has to process increases with each test, first by increasing the number of requests done in the same period of time, and then by keeping the same amount of requests, but increasing the concurrency level, i.e., the number of different clients (in this case simulated by using different threads) making the requests.

Given that for the GSCL, from a performance point of view, receiving requests in a sequential way is different than receiving requests concurrently, these tests can be used to compare how the GSCL behaves when it has to respond to a large number of requests, specially when those requests arrive concurrently.

All the results obtained with this test are presented in Figure 5.4, where is shown a table with the obtained results, along with the respective graphic representation.



No. of concurrent requests	10	50	100	500	1000
100			0.0138	0.0178	0.0152
50		0.0248	0.0273	0.016	0.0165
10	0.0307	0.0304	0.0256	0.0097	0.0122
2	0.0431	0.0312	0.0236	0.0131	0.0159

Figure 5.4: Average time taken to address a notification (concurrently)

Although at first sight the graphics in Figure 5.4 may indicate that the obtained results are very disparate, it's important to note that the times from these experiments are highly variable, since they depend on the arrival of the requests, that is, the simultaneous arrival of two or more requests demands a higher load from the GSCL than just one request.

By analysing the results more closely, however, one can see a pattern. And that is, generally, the higher the requests load, the lower are the notifications time. This may seem a little unnatural, but the pattern can be explained by the way the *NotificationsManager* works. Simply explaining, *NotificationsManager* is a daemon that works by processing notifications that are present in a queue, if there are any, otherwise it just waits for the arrival of notifications to process.

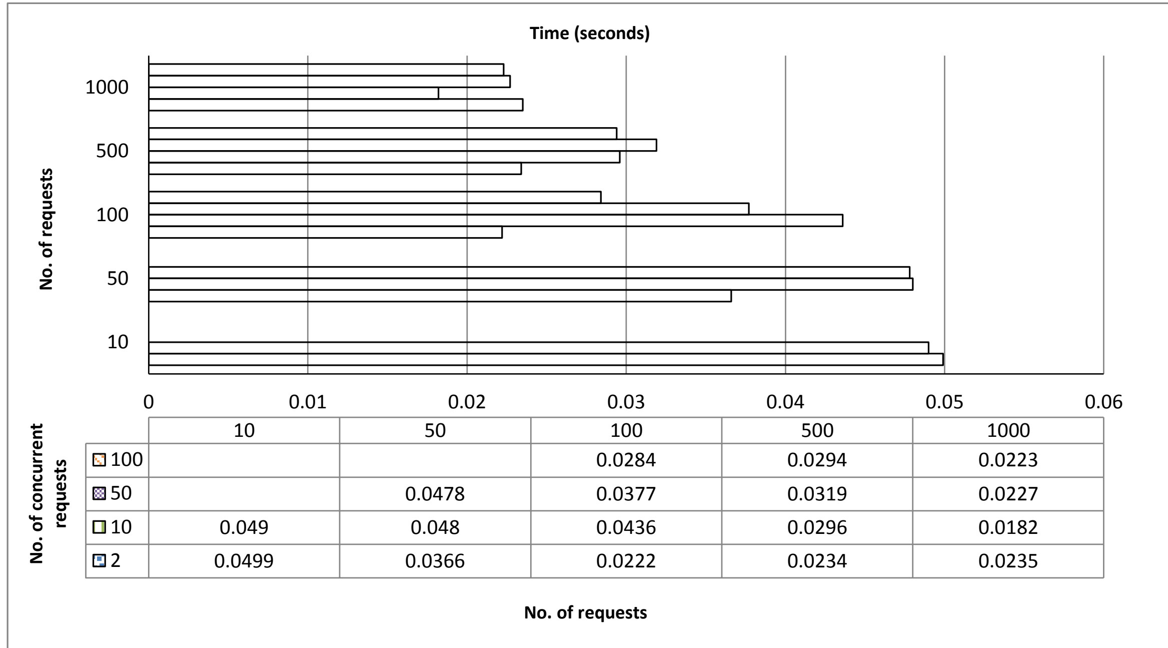
With that explained, consider the case where requests arrive with a certain delay, like the case of 10 requests with a concurrency of 2. In that case, notification time would be higher than in the case of 10 requests with a concurrency of 10, for example. The reasoning behind that is that the daemon in the former case would be waiting for notifications, and the process of notifying it (waking the daemon) to process the notifications is slower than when a higher load of requests exist (the latter case), in which case the daemon doesn't need to wait at all, and is constantly processing notifications.

However, what is important about these results is that even under higher loads the notifications time isn't severely affected. For example, even in the case of a 1000 requests made with a concurrency level of 100, the gateway had a notification time of 0.0152 seconds, which isn't much higher than the time obtained for the 100 requests also made with a concurrency level of 100. In other words, with ten times more requests, the notifications were only delayed by more 0.0014 seconds.

5.2.3 Requests measurement

Measuring how the GSCL responds under different loads of requests is also an important aspect to test, since the gateway has the responsibility of not only interacting with the lower level sensors and actuators, but also with the higher level services. Thus, the second setup used for testing notifications time, in the previous section, was also used here to test the average time taken by the GSCL to attend a request.

In Figure 5.5 is shown a table with the obtained results, as well as a graphic representation of those results.



No. of concurrent requests	10	50	100	500	1000
100			0.0284	0.0294	0.0223
50		0.0478	0.0377	0.0319	0.0227
10	0.049	0.048	0.0436	0.0296	0.0182
2	0.0499	0.0366	0.0222	0.0234	0.0235

Figure 5.5: Average time taken to address a request (concurrently)

Analysing the obtained results one can see that the results are generally higher than the results obtained for notifications, which is normal taking into account that responding to requests involves more resources and operations than generating notifications.

Besides that, one has also to take into account that requests time, in this case, is dependant of the notifications time, since the requests made also generated notifications. In addition, the pattern explained in the previous section is also seen in these results.

As a final point, the obtained values can be used to analyse how the different loads affect the requests attending time of the GSCL, and confirm the conclusion already taken while measuring notifications time, i.e., the GSCL is not significantly disturbed by higher loads of requests.

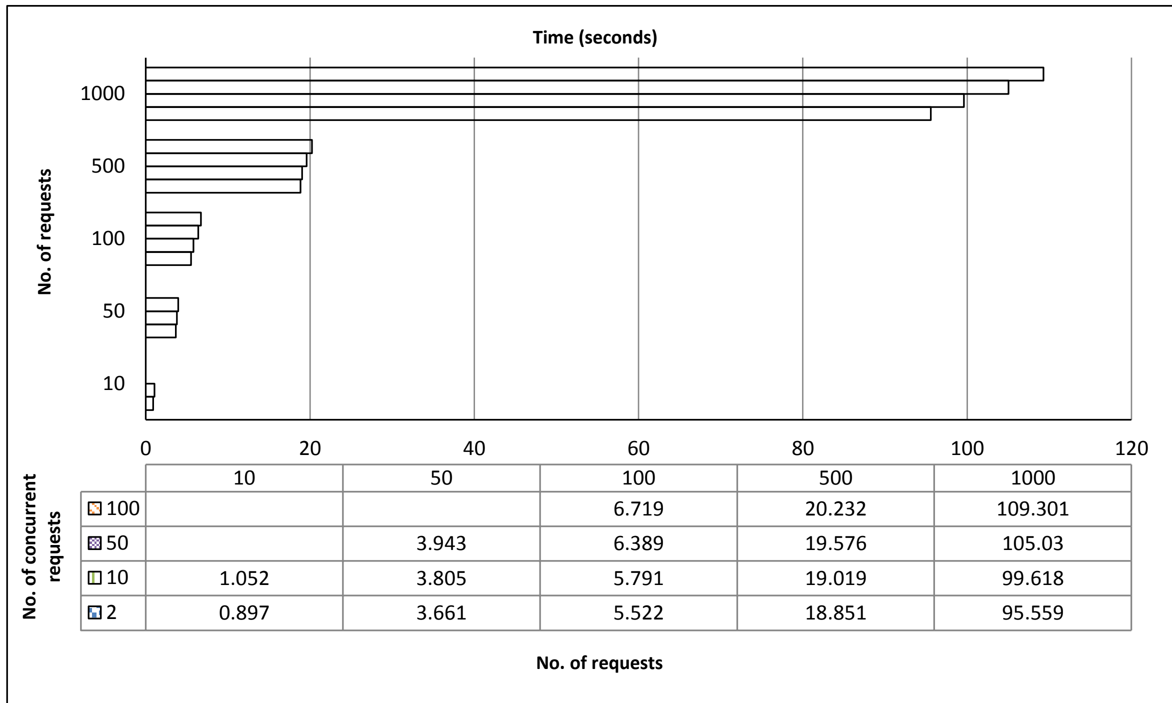
5.2.4 Communication analysis

Although both HTTP and CoAP protocols may be used in this gateway to communicate with the exterior, only CoAP communications will be analysed in this document. The justification for that decision is that CoAP is intended to be used to interact with the resource constrained devices present in the lower layers, and so, it's analysis is more critical than analysing communications made with higher layers (where HTTP is more likely to be used),

in the network domain, where resources are more abundant, and communications aren't so constrained.

To test the behaviour of the gateway under different loads of CoAP requests, the same setup of tests used in the previous chapters, to measure the notifications and requests times, was also used here.

The results obtained during the tests are depicted in Figure 5.6, and in Tables 5.1 and 5.2. More specifically, while Table 5.1 shows the normal average time per request, Table 5.2 shows the average time per request across all concurrent requests.



No. of concurrent requests	10	50	100	500	1000
100			6.719	20.232	109.301
50		3.943	6.389	19.576	105.03
10	1.052	3.805	5.791	19.019	99.618
2	0.897	3.661	5.522	18.851	95.559

Figure 5.6: Average time taken to address 10, 50, 100, 500 or 1000 CoAP requests (concurrently)

	10 req.	**50 req.**	**100 req.**	**500 req.**	**1000 req.**
100			0.06719	0.040464	0.109301
50		0.07886	0.06389	0.039152	0.10503
10	0.1052	0.0761	0.05791	0.038038	0.099618
2	0.0897	0.07322	0.05522	0.037702	0.095559

Table 5.1: Time (in seconds) per request (average)

	10 req.	**50 req.**	**100 req.**	**500 req.**	**1000 req.**
100			6.719	4.0464	10.9301
50		3.943	3.1945	1.9576	5.2515
10	1.052	0.761	0.5791	0.38038	0.99618
2	0.1794	0.14644	0.11044	0.075404	0.191118

Table 5.2: Time (in seconds) per request (average, across all concurrent requests)

By analyzing Figure 5.6 one can see that the results reflect the already expected behaviour from the gateway, i.e., raising the number of requests also raises the average time taken by

the gateway to complete that batch of requests. As such, while a batch of 10 CoAP requests, with a concurrency of 10, take approximately one second to complete, a batch of 1000 CoAP requests, also with a concurrency of 10, take one hundred seconds to compete.

Furthermore, the results in tables 5.1 and 5.2 reveal that the gateway also scales very well under these loads of requests, since raising the number of requests by ten times isn't reflected in the gateway average response time. Particularly, while responding to 100 requests, with a concurrency level of 100, takes an average time of 0.067 seconds, responding to 1000 requests, which is ten times greater, and also with a concurrency level of 100, takes only an average of 0.109 seconds.

Regarding the overhead introduced by CoAP, it obviously varies depending on the number of header options used, and on the size of each option.

In particular, for these tests, three headers were provided, one for the requesting entity, a second for the URI path, and a last one for specifying the content-type of the payload, which is the bare minimum needed to map the standard primitives to the CoAP requests (and vice-versa). Thus, each CoAP request added more ~141 bytes to the total amount of data transmitted.

The screenshot from Wireshark, in Figure 5.7, shows a capture of a CoAP request (UDP messages), and emphasized in red are the headers that are used to map the primitives in the CoAP request.

No.	Time	Source	Destination	Protocol	Length	Info
1843	15.151597	127.0.0.1	127.0.0.1	UDP	695	Source port: coap Destination port: coap
1844	15.154715	127.0.0.1	127.0.0.1	UDP	49	Source port: coap Destination port: coap
1845	15.156818	127.0.0.1	127.0.0.1	UDP	314	Source port: coap Destination port: coap
1846	15.166132	127.0.0.1	127.0.0.1	UDP	87	Source port: coap Destination port: coap

```
▷ Frame 1843: 695 bytes on wire (5560 bits), 695 bytes captured (5560 bits)
▷ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▷ Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
▷ User Datagram Protocol, Src Port: coap (5683), Dst Port: coap (5683)
▽ Data (653 bytes)
    Data: 49021d32396c6f63616c686f73748c6170706c6963617469...
    [Length: 653]
```


```
0020  00 01 16 33 16 33 02 95  00 a9 49 02 1d 32 39 6c   ...3.3.. ..I..29l
0030  6f 63 61 6c 68 6f 73 74  8c 61 70 70 6c 69 63 61   ocalhost .applica
0040  74 69 6f 6e 73 0a 77 65  61 74 68 65 72 41 70 70   tions.we atherApp
0050  0a 63 6f 6e 74 61 69 6e  65 72 73 0d 63 74 72 57   .contain ers.ctrW
0060  65 61 74 68 65 72 41 70  70 0f 01 63 6f 6e 74 65   eatherAp p..conte
0070  6e 74 49 6e 73 74 61 6e  63 65 73 1f 01 61 70 70   ntInstan ces..app
0080  6c 69 63 61 74 69 6f 6e  2f 6a 73 6f 6e 3f 17 52   lication /json?.R
0090  65 71 75 65 73 74 69 6e  67 45 6e 74 69 74 79 3d   equestin gEntity=
00a0  6d 32 6d 2e 72 65 71 75  65 73 74 69 6e 67 2e 65   m2m.requ esting.e
00b0  6e 74 69 74 79 c1 0d 3c  3f 78 6d 6c 20 76 65 72   ntity..< ?xml ver
00c0  73 69 6f 6e 3d 22 31 2e  30 22 20 65 6e 63 6f 64   sion="1. 0" encod
00d0  69 6e 67 3d 22 55 54 46  2d 38 22 3f 3e 0a 3c 6e   ing="UTF -8"?>.<n
00e0  73 30 3a 63 6f 6e 74 65  6e 74 49 6e 73 74 61 6e   s0:conte ntInstan
00f0  63 65 43 72 65 61 74 65  20 78 6d 6c 6e 73 3a 6e   ceCreate  xmlns:n
```


Figure 5.7: Wireshark screenshot with a CoAP request (CoAP headers emphasized)

However, much more overhead comes from the primitives, which during these tests, using XML representations, were accountable for an overhead of 414 bytes to each request transmitted. More specifically, while the payload transmitted was of only 141 bytes (shown in Snippet 2), when converted to the appropriate primitive (shown in Snippet 3), it raised to 555 bytes.

For the purposes of comparing which format, XML or JSON, would cause more overhead, a test was also done using a JSON primitive. The resultant primitive, present in Snippet 4, was of size 353 bytes, thus adding extra 212 bytes to the transmitted payload. Exactly half of the overhead given by the XML primitive.

Thereby, as expected, in overall one saves more space by using the JSON format for representing the transmitted data, than the XML format.


```
{
    "id": 1211,
    "temperature": 25.9,
    "humidity": 64,
    "wind_speed": 0,
    "gust_speed": 0,
    "unknown": 0,
    "rainfall": 0,
    "battery_low": "true",
    "wind_direction": "N"
}
```


Snippet 2: Payload used for the tests


```
<?xml version="1.0" encoding="UTF-8"?>
<ns0:contentInstanceCreate id="contentInstance1"
   xmlns:ns1="http://www.w3.org/2005/05/xmlmime"
   xmlns:ns0="http://uri.etsi.org/m2m">
  <ns0:content ns1:contentType="application/json">
    ZXlKcFpDSWdPaUF5TnpjM0xDSjBaVzF3WlhKaGRIVnlaU0lnT2lBeU5TNDVMQ0pv
    ZFcxcFpHbDBlU0lnT2lBMk5Dd2lkMmx1WkY5emNHVmxaQ0lnT2lBd0xqQXNJbWQx
    YzNSZmMzQmxaV1FpSURvZ01DNHdMQ0oxYm10dWIzZHVJaUE2SURBc0luSmhhVzVt
    WVd4c0lpQTZJREF1TUN3aVltRjBkR1Z5ZVY5c2IzY2lJRG9nSW5SeWRXVWlMQ0oz
    YVc1a1gyUnBjbVZqZEdsdmJpSWdPaUFpVGlKOQ==
  </ns0:content>
</ns0:contentInstanceCreate>
```


Snippet 3: XML Primitive sent during the tests


```
{
  "content": {
    "contentType": "application/json",
    "value":
        "ZXlKcFpDSWdPaUF5TnpjM0xDSjBaVzF3WlhKaGRIVnlaU0lnT2lBeU5TNDVM
        Q0pvZFcxcFpHbDBlU0lnT2lBMk5Dd2lkMmx1WkY5emNHVmxaQ0lnT2lBd0xq
        QXNJbWQxYzNSZmMzQmxaV1FpSURvZ01DNHdMQ0oxYm10dWIzZHVJaUE2SURB
```

```
        c0luSmhhVzVtWVd4c0lpQTZJREF1TUN3aVltRjBkR1Z5ZVY5c2IzY2lJRG9n
        SW5SeWRXVWlMQ0ozYVc1a1gyUnBjbVZqZEdsdmJpSWdPaUFpVGlKOQ=="
    }
}
```

Snippet 4: JSON Primitive

5.3 Comparison with Cocoon

Given that Cocoon, a project that provides an open source implementation of a GSCL, and already introduced in Section 3.2.6.1, is the project that provides the best base documentation, the objective of this section is to compare Cocoon with the GSCL implemented for this Dissertation. As far as possible, both of their use cases, requirements/requisites, and even runtime characteristics will be used to compare both of the implementations.

Starting by comparing both projects target use cases, Cocoon seems to be aiming at much more restrictive environments than our implementation, since the gateways used are devices with ARM9 processors, 60 MB of RAM and 46 MB of flash. From these limited characteristics, one may conclude that these gateways are not intended to act as a gateway for many devices (in fact, their Development Kit [50] comes only with 4 sensors), but rather a few devices only.

To achieve that, Cocoon's GSCL runs in a J2ME (Java Micro Edition) CDC (Connected Device Configuration) Virtual Machine (also known as CVM). Designed for very constrained devices, this VM targets devices with a 32-bit processor and a minimum of 2 MB of memory available.

However, despite of its advantages, one might argue that the approach used by Cocoon is more appropriate for the implementation of a DSCL (a subset of a GSCL), as according to the functionalities specified by ETSI, not every IoT device will be able to run a full DSCL compliant with the standard. As such, running a full DSCL implementation in resource constrained devices, as the ones shown in Table 2.1, will be an arduous task, even nearly impossible in some cases, given that the libraries used for communicating and the primitives mapping will consume more resources than the ones available. Thus, one can say that if a device is going to execute a DSCL, it will need to have a minimal set of requirements (in terms of CPU, RAM memory, among others), and therefore, using an implementation using J2ME wouldn't be a bad option.

On the other hand, our GSCL implementation, despite of targeting efficiency and a small footprint, doesn't aim at such devices, but rather less constrained devices, as the one used at the demo (Alix 3D2). However, as shown in memory usage Figure 5.3, nothing prevents this GSCL implementation from running on a device with only 60 MB of memory either.

Taking into account the documentation, one can see that Cocoon, despite being an older project (from 2011), doesn't yet provide a full implementation of a GSCL, being some of the functionalities specified by the standard still unimplemented, as the *notificationChannels*.

Another limitation is the fact that it only supports the XML format, and not JSON, which as seen before, introduces much more overhead in the transmitted messages.

Regarding the storage of data, just like our implementation, Cocoon also uses SQLite for persisting the information. However, contrary to our database model, Cocoon doesn't follow a hierarchical model, and uses instead a much simpler model, dividing the data between two

tables, according to their type. Thus, if it represents an attribute, the data is stored in the Attribute table, while if it is a resource, it is stored in the Document table. Figures 5.8 and 5.9 show the tables Attribute and Document, respectively, of Cocoon's database.

TABLE ATTRIBUTE Search Show All Add Duplicate Edit Delete

rowid	DOCUMENT_PATH	DOCUM...	DATE_V...	INT_VAL...	NAME	VALUE	ATTRIB...
5	/	containers	2013-02-14...	0	creationTime		8
6	/	containers	2013-02-14...	0	lastModifie...		8
9	/		2013-02-14...	0	creationTime		8
10	/		2013-02-14...	0	lastModifie...		8
13	/			0	searchString	hw/type/gat...	6
14	/			0	searchString	hw/vendor/...	6
15	/			0	searchString	hw/name/C...	6
16	/			0	searchString	hw/version/...	6
17	/			0	searchString	fwk/vendor/...	6
18	/			0	searchString	fwk/name/S...	6
19	/			0	searchString	fwk/version/...	6
20	/		2013-02-14...	0	creationTime		8
21	/		2013-02-14...	0	lastModifie...		8
26	/applications/SYSTEM/	accessRights	2013-02-14...	0	creationTime		8
27	/applications/SYSTEM/	accessRights	2013-02-14...	0	lastModifie...		8
28	/applications/SYSTEM/	notification...	2013-02-14...	0	creationTime		8
29	/applications/SYSTEM/	notification...	2013-02-14...	0	lastModifie...		8
30	/applications/	SYSTEM		0	type	application	6
31	/applications/	SYSTEM		0	searchString	ETSI.Object...	6
32	/applications/	SYSTEM	2013-02-14...	0	creationTime		8
33	/applications/	SYSTEM	2013-02-14...	0	lastModifie...		8
34	/	applications	2013-02-14...	0	creationTime		8
35	/	applications	2013-02-14...	0	lastModifie...		8
38	/accessRights/	Netadmin_AR		0	type	accessRight	6
39	/accessRights/	Netadmin_AR	2013-02-14...	0	creationTime		8

Figure 5.8: Table 'Attribute' of Cocoon's DB

TABLE DOCUMENT Search Show All Add Duplicate Edit Delete

rowid	PATH	NAME	XML_CONTENT
1	/scls/	subscriptions	BLOB (Size: 148)
2	/	scls	BLOB (Size: 316)
3	/applications/	subscriptions	BLOB (Size: 148)
4	/	applications	BLOB (Size: 332)
5	/containers/	subscriptions	BLOB (Size: 148)
6	/	containers	BLOB (Size: 268)
7	/accessRights/	subscriptions	BLOB (Size: 148)
8	/	accessRights	BLOB (Size: 332)
9	/	subscriptions	BLOB (Size: 148)
10	/		BLOB (Size: 600)
11	/applications/SYSTEM/contain...	subscriptions	BLOB (Size: 148)
12	/applications/SYSTEM/	containers	BLOB (Size: 328)
13	/applications/SYSTEM/access...	subscriptions	BLOB (Size: 148)
14	/applications/SYSTEM/	accessRights	BLOB (Size: 332)
15	/applications/SYSTEM/	subscriptions	BLOB (Size: 148)
16	/applications/SYSTEM/	notificationChannels	BLOB (Size: 292)
17	/applications/	SYSTEM	BLOB (Size: 748)
18	/accessRights/Netadmin_AR/	subscriptions	BLOB (Size: 148)
19	/accessRights/	Netadmin_AR	BLOB (Size: 1996)
20	/accessRights/Locadmin_AR/	subscriptions	BLOB (Size: 148)
21	/accessRights/	Locadmin_AR	BLOB (Size: 2068)
22	/accessRights/Locadmin_AR2/	subscriptions	BLOB (Size: 148)
23	/accessRights/	Locadmin_AR2	BLOB (Size: 2084)
24	/applications/SYSTEM/contain...	subscriptions	BLOB (Size: 148)
25	/applications/SYSTEM/contain...	contentInstances	BLOB (Size: 532)

Figure 5.9: Table 'Document' of Cocoon's DB

In their database, however, as can be seen in Figures 5.8 and 5.9, the fact that they are using strings for identifying the type of resource, or attribute, is an interesting fact, since that option was discarded from our database. Despite of needing less space, using strings in a field that is used constantly for doing search operations for a specific type of attribute/resource, can become costly in terms of resources. Even with the use of indexes.

Although Cocoon entitles itself has an open source project, despite the efforts done, their GSCL code wasn't found anywhere[1]. Therefore, the only way to withdraw knowledge of their implementation, for comparison, was by reading their documentation and using/testing a Cocoon installation.

Thus, for comparison, both implementations were installed on a Linux Fedora i686 virtual machine image. Running each application individually, one could see that both the implementations, on idle, consume similarly the same amount of memory, with Cocoon consuming 27 MB of memory, and this implementation sitting on 35 MB of memory.

Furthermore, despite confirming that Cocoon's GSCL was listening for both HTTP and CoAP traffic, no interactions were successfully made, either through CoAP[2] or through HTTP[3], which is unfortunate, since that would help to check how Cocoon behaves at runtime, even under higher loads of requests. Those results would then be used to compare Cocoon with this Dissertation GSCL.

[1]neither on their site, nor even after a fresh installation of the application (only compiled binaries are provided)

[2]During the attempts, no error message was returned back.

[3]Gave always the 'Resource not found' error, even though the resource provided being exactly equal to the one provided in their documentation [51]

Chapter 6

Conclusion

The recent efforts done both by the research community and SDOs to counteract the problems that pose a threat to the evolution of M2M systems, and consequently the achievement of a global IoT, have produced some decent solution proposals, and even more importantly, improved the knowledge of the area, which is essential when considering its perpetuation. This document not only provides a critical overview of some of the most relevant proposals, but, using the knowledge gained from evaluating the different proposals, also outlines some of the most important requirements for these middleware platforms.

However, the ultimate goal of this Dissertation was to produce a functional M2M gateway as mandated by the latest ETSI M2M standard [5]. The functionalities implemented for this gateway were successfully tested both in a demo testbed (a real use case), as well as concerning the performance. Regarding the former, it enabled to test the communication of the gateway with low profile devices, using the CoAP protocol and also a GIP module, as foreseen by the standard. The latter, on the other hand, enabled to check for performance issues, ensuring that the gateway was appropriate for more demanding scenarios.

The architecture conceptualized for this implementation allows that, not only new GAs may be added without much effort, and new GIPs may be implemented easily, to provide the gateway with new interactions capabilities, but also that the gateway may be used with more constrained devices, in opposition to real desktop computers/laptops or servers.

Besides explaining the implementation, which gives a good practical knowledge of ETSI's M2M standard, this document also provides an overview about how the standard works, and the visions and approaches taken by ETSI.

6.1 Future Work

Although the provided implementation is a working and tested version of a ETSI M2M gateway, there are still functionalities that weren't implemented, and some others that weren't tested due to the current lack of other entities's implementations to interact (NSCL, for example). Some of the most relevant and important functionalities that the gateway should have in the future are:

- **Notification channels:** Implement a way for non-server clients to receive asynchronous notifications;
- **Improve interoperability with NSCLs:** Implement *announces* to improve synchronization between GSCL and NSCL;
- **Service Bootstrap:** Implement the bootstrap of the GSCL;

- **Capabilities:** Implement GSec, to provide security mechanisms, GREM, to provide remote management, and GCS, to provide network communication functionalities;
- **Requirements assessment:** Assess which are the minimum requirements to execute this GSCL implementation.

Appendix A

Database selection

Although other models were considered, mainly the document store, the key-value store and the graph models, the relational model proved to be the most appropriate model for representing the hierarchical structured data defined by the ETSI M2M standard.

After deciding in favor of a relational model, in order to appropriately chose a database, some criteria were defined:

- Fast and lightweight;
- Minimal external configuration;
- Supports concurrency;
- Supports Binary Large Objects (BLOBs) of large sizes;
- Supports a JDBC driver;
- Public license.

Analysing the first two criteria defined above, it was a logical choice restraining the range of candidates to only embedded databases, in favor of the more resource consuming and hard to manage distributed databases. Thereby, in this implementation, the only process that is able to interact with the database is its owner, the GSCL.

Among the existent embedded databases, two of them, SQLite [52] and H2 [53], stood out, not only for matching the defined criteria, but also because of the benchmarks analysed from the official websites [54] [55]. (Note that besides of outdated, and despite the warning on the top of the website, the SQLite reference shows that it's an appropriate embedded candidate)

A.1 Tests

To decide between the two databases, 16 sets of tests were performed. Four tests for each type of operation (INSERTs, UPDATEs, SELECTs and DELETEs), where the first test had 1000 records (first case), the second 10000 records (second case), the third 100000 records (third case) and the forth 1000000 records (fourth case).

Before showing the results, it's important to note that to improve the accuracy, several measures where taken:

- To ensure that one test run doesn't affect the other, each test was executed in a new process with a new JVM;

- None of the results accounted for the initialization activities performed before the execution of the tests themselves. Thus, the results obtained are only relative to the execution of the operations under test;
- To minimize the effects of other OS processes that might be running in the background, each test was performed 5 times, and only the 4 best results were taken into account for the final average;
- To get more realistic results, the tests were performed in a database equal to the one used in the gateway. Also, the queries used in the tests simulated real operations.

A.1.1 Hardware and Software

All of the tests were performed under the following hardware:

CPU Intel Core 2 Duo Processor T9300 @ 2.50GHz;
RAM 4GB;
Hard Disk SATA 500GB @ 5400RPM.

And using the following software:

- Linux Fedora 17 (x86_64);
- Sun Java JDK 7 update 09.

A.1.2 Results

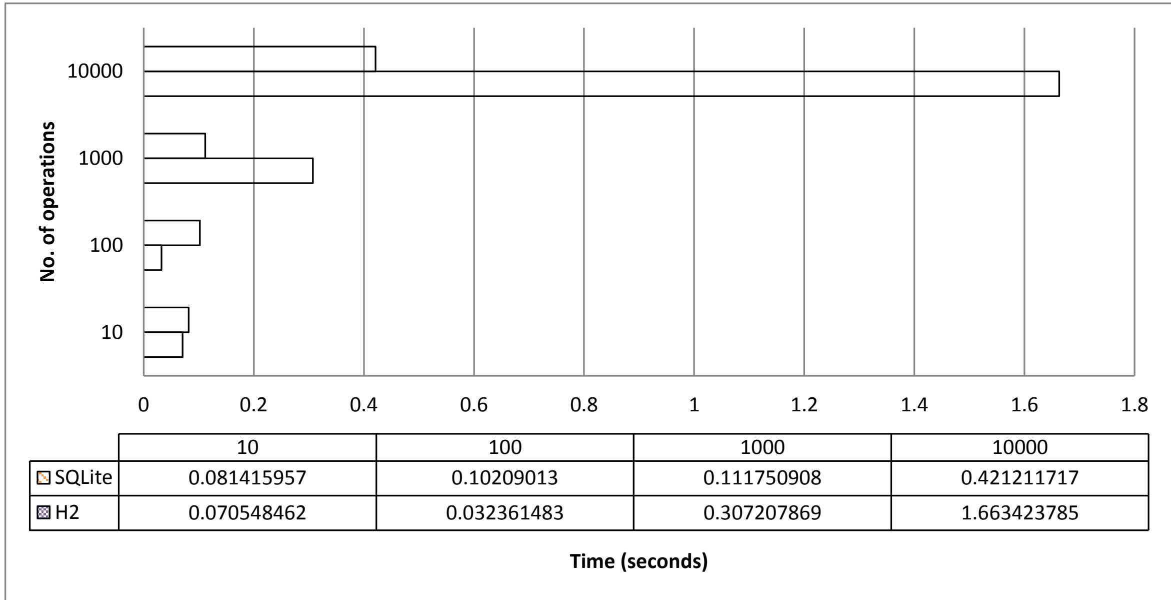
This section will show the obtained results, using tables for presenting the real numbers, and graphics for presenting a graphical comparison of performances between SQLite and H2 databases.

The performance stated here is relative to: i) time taken to perform the operations; ii) memory used by the JDBC libraries to perform the operations in the database.

At last, it's also important to note that the memory results shown below were obtained by analysing the memory used by the JVM. Therefore, when comparing those results, although they give a notion of which operations consume more memory, they can't be taken literally.

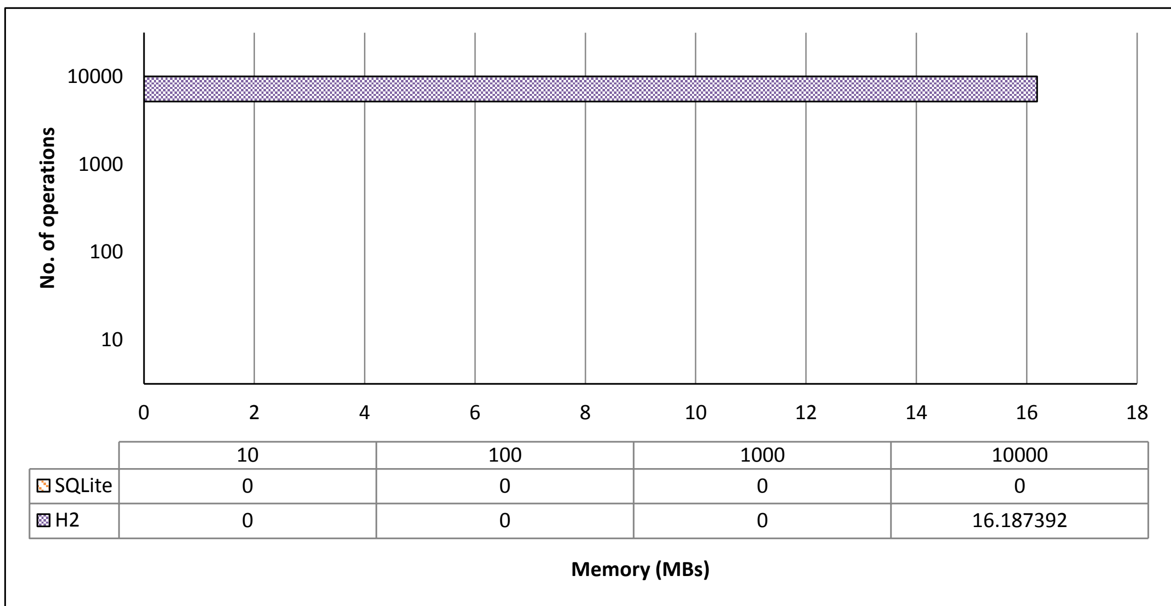
A.1.2.1 *INSERTs*

Regarding *INSERT* operations, one can see in Figure A.1 that H2 outruns SQLite in the first two cases, taking approximately three times less to complete a transaction of 100 *INSERT*s. However, in the last two cases SQLite outruns H2, the latter taking more than 1 second to complete a transaction of 10000 *INSERT*s. On the other hand, considering the memory used (Figure A.2), and only including the fourth case, since it is the only one that is relevant, SQLite consumes far less memory than H2, since the former doesn't consume any extra memory and the latter needs more 16 MB of memory for inserting 100000 records in the database.



	10	100	1000	10000
SQLite	0.081415957	0.10209013	0.111750908	0.421211717
H2	0.070548462	0.032361483	0.307207869	1.663423785

Figure A.1: Time performance of INSERT operations

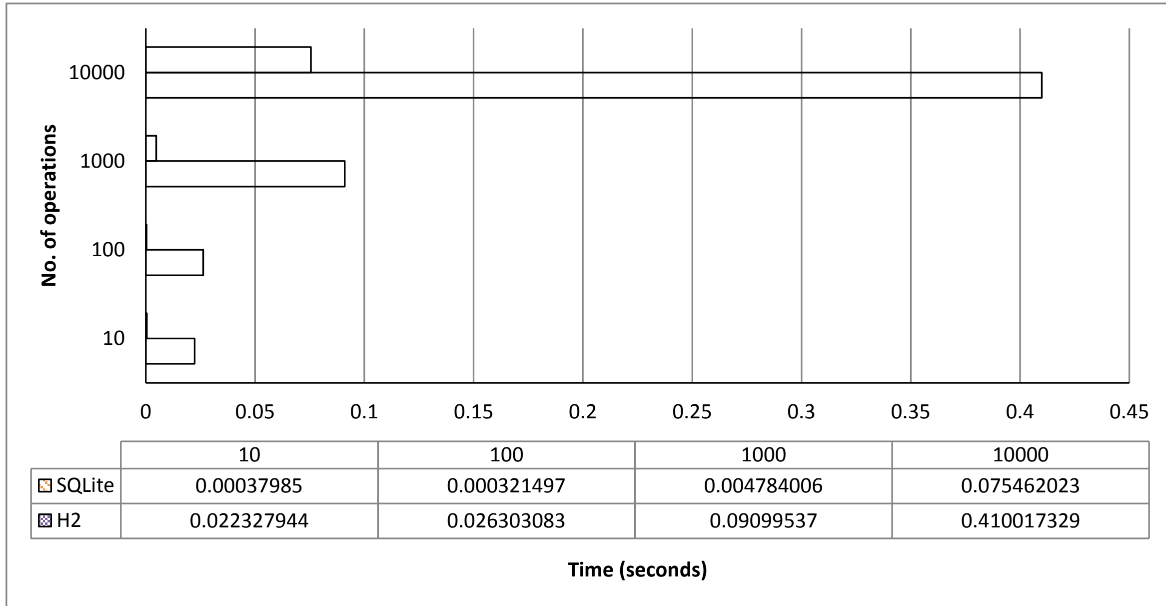


	10	100	1000	10000
SQLite	0	0	0	0
H2	0	0	0	16.187392

Figure A.2: Memory performance of INSERT operations

A.1.2.2 SELECTs

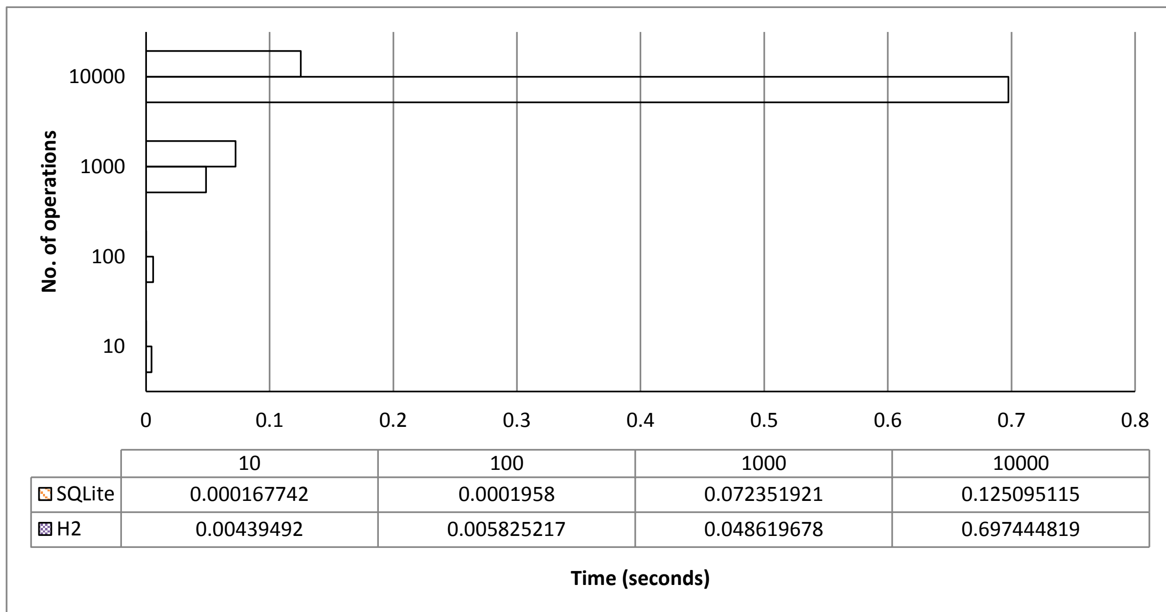
Figure A.3 shows that SQLite is much faster than H2 in all cases of *SELECT* operations, surpassing H2 by 80 times, in the second case. In this test, however, is clear that the H2 overall performance is far from ideal, since it falls short of SQLite's performance. Regarding memory usage, both SQLite and H2 don't consume any relevant memory during the execution of these operations.



	10	100	1000	10000
SQLite	0.00037985	0.000321497	0.004784006	0.075462023
H2	0.022327944	0.026303083	0.09099537	0.410017329

Figure A.3: Time performance of SELECT operations

A.1.2.3 UPDATEs



	10	100	1000	10000
SQLite	0.000167742	0.0001958	0.072351921	0.125095115
H2	0.00439492	0.005825217	0.048619678	0.697444819

Figure A.4: Time performance of UPDATE operations

UPDATE operations had a better performance in the SQLite database, since this database took much less time to complete the first, the second, and the fourth cases than the H2 database (in the first case, for example, SQLite was 45 times faster than H2, approximately). However, even if by a small margin, in the third case H2 outperformed SQLite, taking 0.024 seconds less to complete 1000 *SELECT*s. Regarding memory, as in the case of *INSERT*s, H2 uses much more memory than SQLite, needing more 8 MB of memory for performing 10000 *UPDATE*.

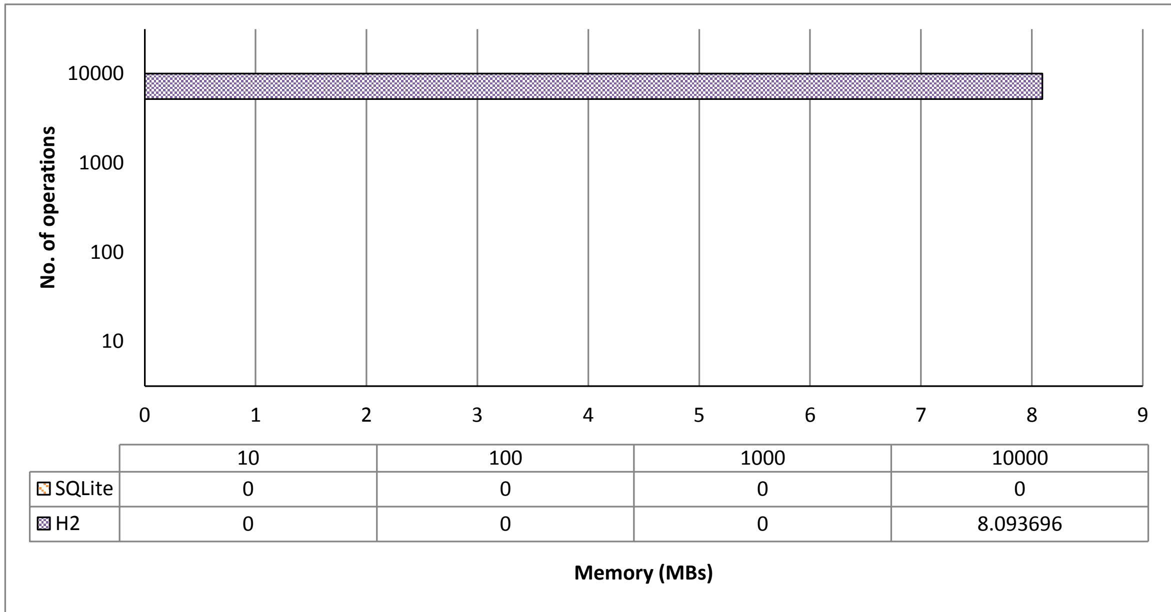


Figure A.5: Memory performance of UPDATE operations

A.1.2.4 DELETEs

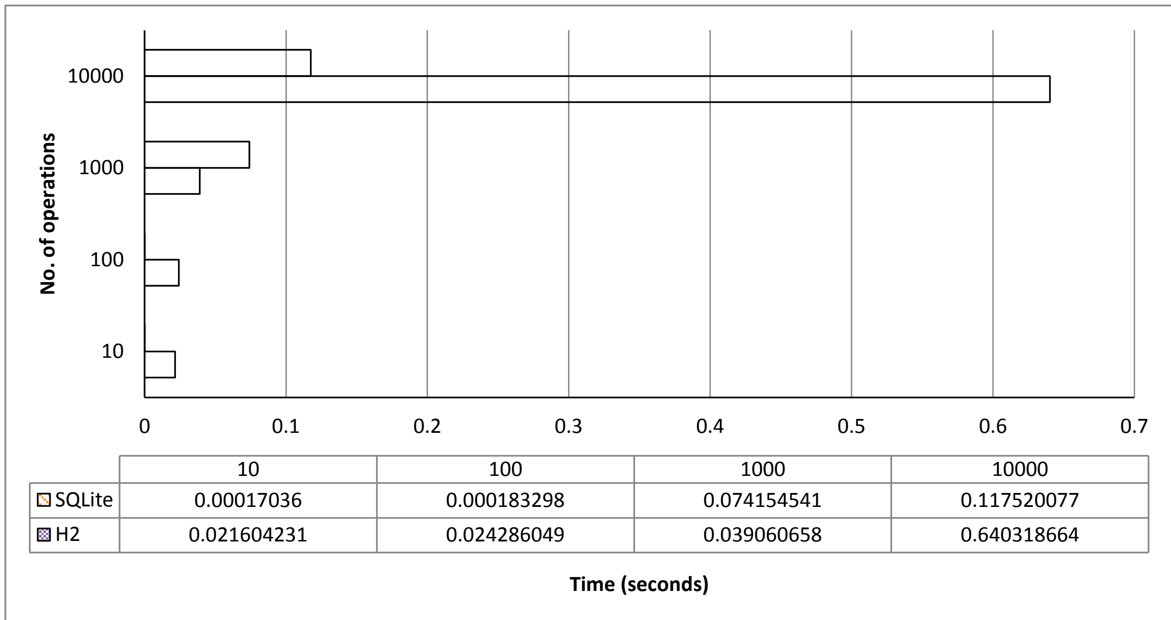


Figure A.6: Time performance of DELETE operations

Regarding time, *DELETE* operations and *UPDATE* operations have very similar results, and as such, the conclusions taken in the last section can be applied here. In memory, however, contrary to what happens in *UPDATE*s, the *DELETE* operations don't consume any relevant memory.

A.1.2.5 Summary

According to the results presented in the previous section, SQLite reveals itself as the most appropriate database for storing the GSCL and all the data received previously. As a first advantage, it stands out the fact that it didn't need any more memory than the one already provided by the JVM for the execution of the tests, while the H2 database, for the *SELECT* and the *UPDATE* operations demanded more memory. On the other hand, regarding the times obtained during the tests, SQLite achieved the best results during the *INSERT* operations, although H2 outperformed SQLite in the first two cases (10 and 100 records), the *SELECT* operations, and in almost every case of the *UPDATE* and *DELETE* operations, with the exception of the 1000 records case.

Thus, the SQLite database has the best ratio in terms of time and memory consumed during each operation. Considering it's worst cases, namely the third case in *UPDATE* and *DELETE* operations, one can argue that those are not the most common operations in M2M environments, but rather the *SELECT* and *INSERT* operations, since sensors/actuators and other entities ought to perform much less modifications to the database than searches and creations. Furthermore, the way the information is structured in the ETSI M2M standard dictates that smaller transactions (in the tens or hundreds) are likely to be more frequent.

To complete this section, despite of H2 outrunning SQLite in the first two cases of *INSERT*s, it's only by a small margin, and thus considered insufficient to cancel out the advantages observed in all of the other cases where SQLite outperforms H2, sometimes being 80 times faster.

Appendix B

Code snippets

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:m2m="http://uri.etsi.org/m2m"
   targetNamespace="http://uri.etsi.org/m2m">
    <xs:include schemaLocation="../complexTypes/searchStrings.xsd"/>
    <xs:include schemaLocation="../complexTypes/announceTo.xsd"/>
    <xs:include schemaLocation="../complexTypes/aPocPaths.xsd"/>

    <xs:element name="application">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="containers" type="xs:anyURI"/>
                <xs:element name="groups" type="xs:anyURI"/>
                <xs:element name="accessRights" type="xs:anyURI"/>
                <xs:element name="subscriptions" type="xs:anyURI"/>
                <xs:element name="notificationChannels"
                   type="xs:anyURI"/>
                <xs:element name="expirationTime"
                   type="xs:dateTime"/>
                <xs:element name="accessRightID" type="xs:anyURI"
                   minOccurs="0"/>
                <xs:element name="searchStrings"
                   type="m2m:searchStrings"/>
                <xs:element name="creationTime" type="xs:dateTime"/>
                <xs:element name="lastModifiedTime"
                   type="xs:dateTime"/>
                <xs:element name="announceTo" type="m2m:announceTo"/>
                <xs:element name="aPoc" type="xs:anyURI"
                   minOccurs="0"/>
                <xs:element name="aPocPaths" type="m2m:aPocPaths"
                   minOccurs="0"/>
                <xs:element name="locRequestor" type="xs:string"
                   minOccurs="0"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema
```

Snippet 5: XSD representation of the application resource

```java
package pt.it.av.apollogw.schemaClasses;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "containers",
    "groups",
    "accessRights",
    "subscriptions",
    "notificationChannels",
    "expirationTime",
    "accessRightID",
    "searchStrings",
    "creationTime",
    "lastModifiedTime",
    "announceTo",
    "aPoc",
    "aPocPaths",
    "locRequestor"
})
@XmlRootElement(name = "application")
public class Application {

    @XmlElement(required = true)
    @XmlSchemaType(name = "anyURI")
    protected String containers;
    @XmlElement(required = true)
    @XmlSchemaType(name = "anyURI")
    protected String groups;
    @XmlElement(required = true)
    @XmlSchemaType(name = "anyURI")
    protected String accessRights;
    @XmlElement(required = true)
    @XmlSchemaType(name = "anyURI")
    protected String subscriptions;
    @XmlElement(required = true)
    @XmlSchemaType(name = "anyURI")
    protected String notificationChannels;
    @XmlElement(required = true)
    @XmlSchemaType(name = "dateTime")
    protected XMLGregorianCalendar expirationTime;
    @XmlSchemaType(name = "anyURI")
    protected String accessRightID;
    @XmlElement(required = true)
    protected SearchStrings searchStrings;
    @XmlElement(required = true)
    @XmlSchemaType(name = "dateTime")
    protected XMLGregorianCalendar creationTime;
    @XmlElement(required = true)
    @XmlSchemaType(name = "dateTime")
    protected XMLGregorianCalendar lastModifiedTime;
    @XmlElement(required = true)
    protected AnnounceTo announceTo;
    @XmlSchemaType(name = "anyURI")
    protected String aPoc;
    protected APocPaths aPocPaths;
```

```
protected String locRequestor;

public String getContainers() {
    return containers;
}

public void setContainers(String value) {
    this.containers = value;
}

public String getGroups() {
    return groups;
}

public void setGroups(String value) {
    this.groups = value;
}

public String getAccessRights() {
    return accessRights;
}

public void setAccessRights(String value) {
    this.accessRights = value;
}

public String getSubscriptions() {
    return subscriptions;
}

public void setSubscriptions(String value) {
    this.subscriptions = value;
}

public String getNotificationChannels() {
    return notificationChannels;
}

public void setNotificationChannels(String value) {
    this.notificationChannels = value;
}

public XMLGregorianCalendar getExpirationTime() {
    return expirationTime;
}

public void setExpirationTime(XMLGregorianCalendar value) {
    this.expirationTime = value;
}

public String getAccessRightID() {
    return accessRightID;
}

public void setAccessRightID(String value) {
    this.accessRightID = value;
```

```
    }

    public SearchStrings getSearchStrings() {
        return searchStrings;
    }

    public void setSearchStrings(SearchStrings value) {
        this.searchStrings = value;
    }

    public XMLGregorianCalendar getCreationTime() {
        return creationTime;
    }

    public void setCreationTime(XMLGregorianCalendar value) {
        this.creationTime = value;
    }

  public XMLGregorianCalendar getLastModifiedTime() {
        return lastModifiedTime;
    }

    public void setLastModifiedTime(XMLGregorianCalendar value) {
        this.lastModifiedTime = value;
    }

    public AnnounceTo getAnnounceTo() {
        return announceTo;
    }

    public void setAnnounceTo(AnnounceTo value) {
        this.announceTo = value;
    }

    public String getAPoc() {
        return aPoc;
    }

    public void setAPoc(String value) {
        this.aPoc = value;
    }

    public APocPaths getAPocPaths() {
        return aPocPaths;
    }

    public void setAPocPaths(APocPaths value) {
        this.aPocPaths = value;
    }

    public String getLocRequestor() {
        return locRequestor;
    }

    public void setLocRequestor(String value) {
        this.locRequestor = value;
```

```
    }

}
```

Snippet 6: POJO representation of the application resource

Appendix C

Resources's structure

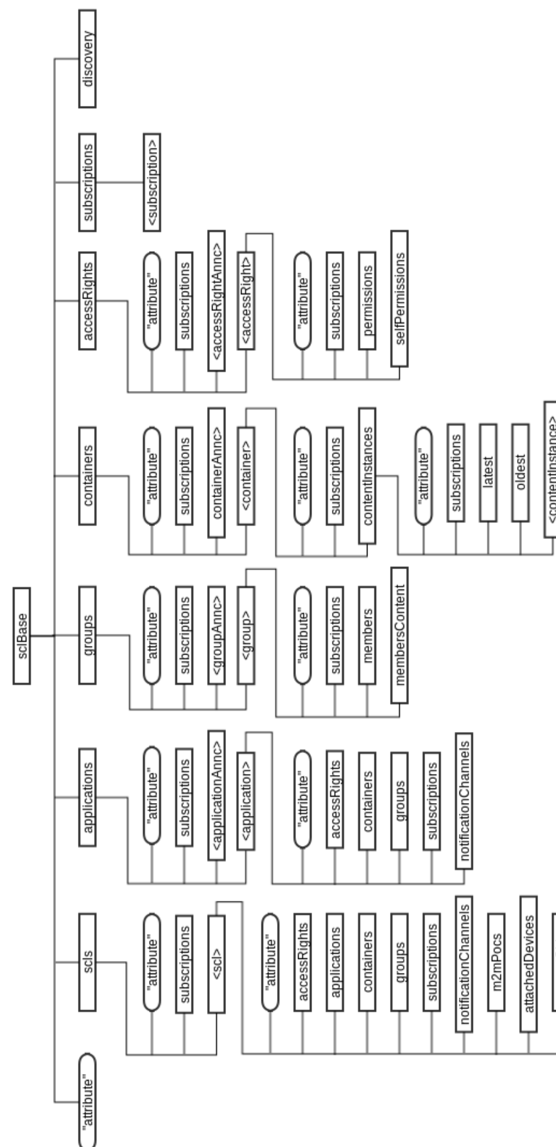


Figure C.1: Resources Structure

Glossary

Notation	Description
6LoWPAN	A set of standards that enable the efficient use of IPv6 over 802.15.4 based networks, thus enabling simple embedded devices to interact with IPv6 networks.
Bluetooth	An open standard for transmitting fixed and mobile electronic data over short distances.
OSGi	Formerly known as the Open Services Gateway initiative, now an obsolete name, OSGi is a set of specifications that define a dynamic component system for Java.
WiFi	Technology based on the 802.11 standards that allow electronic devices to connect and send data wirelessly, using radio waves, to the internet.
Z-Wave	A wireless communication standard designed for home automation. Based on the ITU G.9959 specification, Z-Wave uses low-power (Radio Frequency) RF to remotely control applications.
ZigBee	A set of specifications that define the OSI layers above the 802.15.4 standard. It's designed to be simpler and less expensive than other technologies, like Bluetooth, and uses low-power digital radio signals for allowing efficient communications between electronic devices.

References

[1] M. Strohbach, J. B. Vercher, and M. Bauer, "A case for ims: harvesting the power of ubiquitous sensor networks", *Vehicular Technology Magazine, IEEE*, vol. 4, pp. 57–64, Mar. 2009. DOI: 10.1109/MVT.2008.931627.

[2] R. H. Glitho, "Application architectures for machine to machine communications: research agenda vs. state-of-the art", in *Broadband and Biomedical Communications (IB2Com), 2011 6th International Conference on*, 2011, pp. 1–5. DOI: 10.1109/IB2Com.2011.6217900.

[3] D. G. Velev, "Internet of things – analysis and challenges", *Economic alterntives*, pp. 99–109, 2011.

[4] *Apollo project website*, http://atnog.av.it.pt/projects/apollo, Accessed at November 2013.

[5] *Machine-to-machine communications (m2m); functional architecture*, Technical Specification, ETSI, Oct. 2011.

[6] J. Swetina, "ETSI M2M - oneM2M – IoT (aspects of standardization)", NEC.

[7] "The evolution of wireless sensor networks", Silicon Labs, Tech. Rep., http://www.silabs.com/SupportDocuments/TechnicalDocs/evolution-of-wireless-sensor-networks.pdf Accessed atNovember 2013.

[8] *Data sensing lab i/o 2013*, http://data-sensing-lab.appspot.com/, Accessed at November 2013.

[9] M. Maurya and S. R. N. Shukla, "Current wireless sensor nodes (motes): performance metrics and constraints", *International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE)*, vol. 2, pp. 45–48, Jan. 2013.

[10] *Sun spot programmer's manual*, Document version: v6.0 (Yellow), Sun Microsystems and Oracle, Nov. 2010.

[11] *Waspmote technical guide*, Document version: v3.1, libelium, Jul. 2012.

[12] *Z1 datasheet*, Document version: v1.1, Zolertia, Mar. 2010.

[13] *Tinyos*, http://www.tinyos.net/, Accessed at November 2013.

[14] *Contiki*, http://www.contiki-os.org/, Accessed at November 2013.

[15] *Tinynode 584*, http://www.tinynode.com/?q=product/tinynode584/tn-584-868, Accessed at December 2013.

[16] *Wismote*, http://wismote.org/doku.php, Accessed at December 2013.

[17] *Cm5000 telosb*, http://www.advanticsys.com/shop/mtmcm5000msp-p-14.html, Accessed at December 2013.

[18] M. Hatton, *The global m2m market in 2013*, White Paper, Machina Research, Jan. 2013.

[19] K. Ashton, *That 'internet of things' thing*, `http://www.rfidjournal.com/articles/view?4986`, Accessed at November 2013, Jul. 2009.

[20] M. Weiser, "The computer for the 21st century", *Scientific American*, vol. 265, no. 3, pp. 66–75, 1991.

[21] "Next generation networks – frameworks and functional architecture models: overview of the internet of things", International Telecommunication Union, Recommendation Y.2006, Jun. 2012, Series Y: Global information infrastructure, internet protocol aspects and next-generation networks.

[22] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan, "Irisnet: an architecture for a worldwide sensor web", *Pervasive Computing, IEEE*, vol. 2, no. 4, pp. 22–33, Oct. 2003. DOI: `10.1109/MPRV.2003.1251166`.

[23] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh, "Hourglass: an infrastructure for connecting sensor networks and applications", Tech. Rep., 2004.

[24] R. Gold, A. Gluhak, N. Bui, A. Waller, C. Sorge, F. Montagut, V.Stirbu, H. Karvonen, V. Tsiatsis, S. Pennington, Z. Shelby, J. Vercher, M. Johansson, J.Bohli, T. Bauge, S. Esfandiyari, S. Haller, E. Kovacs, R. Egan, F. Carrez, and M. Presser, "Sensei d3.1: state of the art – sensor frameworks and future", in, Deliverable Report, Jan. 2008.

[25] J. B. Vercher, S. P. Marin, A. G. Lucas, R. S. Mollon, L. V. Grande, L. M. C. Cervera, and L. H. Gomez, "Ubiquitous sensor networks in ims: an ambient intelligence telco platform", in *Proceedings ICT-MobileSummit 2008 Conference*, IIMC International Information Management Corporation, 2008.

[26] K. Aberer, M. Hauswirth†, and A. Salehi, "Infrastructure for data processing in large-scale interconnected sensor networks", in *Mobile Data Management, 2007 International Conference on*, May 2007, pp. 198–205. DOI: `10.1109/MDM.2007.36`.

[27] I. Chatzigiannakis, C. Koninis, G. Mylonas, U. Colesanti, and A. Vitaletti, "A peer-to-peer framework for globally-available sensor networks and its application in building management", in *2nd International Workshop on Sensor Network Engineering (IWSNE'09)*, Jun. 2009.

[28] M. Isomura, H. Horiuchi, T. Riedel, C. Decker, and M. Beigl, "Sharing sensor networks", in *In ICDCSW '06: Proceedings of the 26th IEEE International ConferenceWorkshops on Distributed Computing Systems*, IEEE Computer Society, 2006, p. 61. DOI: `10.1109/ICDCSW.2006.98`.

[29] O. Akribopoulos, I. Chatzigiannakis, C. Koninis, and E. Theodoridis, "A web services-oriented architecture for integrating small programmable objects in the web of things", in *Developments in E-systems Engineering (DESE), 2010*, Sep. 2010, pp. 70–75. DOI: `10.1109/DeSE.2010.19`.

[30] A. P. Castellani, N. Bui, P. Casari, M. Rossi, Z. Shelby, and M. Zorzi, "Architecture and protocols for the internet of things: a case study", in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, Mar. 2010, pp. 678–683. DOI: `10.1109/PERCOMW.2010.5470520`.

[31] A. Gluhak and W. Schott, "A wsn system architecture to capture context information for beyond 3g communication systems", in *Intelligent Sensors, Sensor Networks and Information, 2007. ISSNIP 2007. 3rd International Conference on*, Dec. 2007, pp. 49–54. DOI: `10.1109/ISSNIP.2007.4496818`.

[32] *The sensei real world internet architecture*, White Paper, Sensei Consortium, FP7 Project Number 215923, Mar. 2010.

[33] *Etsi's organization chart*, `http://www.etsi.org/about/our-structure/organization-chart`, Official ETSI Website. Accessed at November 2013.

[34] J. Koss, “Etsi standardizes m2m communications”, *telit2market*, pp. 38–39, May 2010, Vice-Chairman of ETSI TC M2M.

[35] *Goals for and benefits of onem2m standardization*, `http://www.onem2m.org/whyjoin.cfm`, Official oneM2M Website. Accessed at November 2013.

[36] G. Wu, S. Talwar, K. Johnsson, N. Himayat, and K. D. Johnson, “M2m: from mobile to embedded internet”, *Communications Magazine, IEEE*, vol. 49, no. 4, pp. 36–43, Apr. 2011. DOI: `10.1109/MCOM.2011.5741144`.

[37] H. van der Veer and A. Wiles, *Etsi white paper no. 3 achieving technical interoperability - the etsi approach*, White Paper, Apr. 2008.

[38] *Machine-to-machine communications (m2m); mia, dia and mid interfaces*, Technical Specification, ETSI, Feb. 2012.

[39] Z. Shelby, K. Hartke, and C. Bormann, *Internet-draft, constrained application protocol (coap)*, `http://www.ietf.org/id/draft-ietf-core-coap-18.txt`, draft-ietf-core-coap-18, Jun. 2013.

[40] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *Rfc 2616, hypertext transfer protocol – http/1.1*, `http://www.rfc.net/rfc2616.html`, Jun. 1999.

[41] I. Cha, Y. Shah, A. U. Schmidt, A. Leicher, and M. V. (Meyerstein, “Trust in m2m communication”, *Vehicular Technology Magazine, IEEE*, vol. 4, no. 3, pp. 69–75, Sep. 2009. DOI: `10.1109/MVT.2009.933478`.

[42] *Cocoon overview*, `http://cocoon.actility.com/documentation/ongv2/overview`, Official Cocoon Website. Accessed at November 2013.

[43] *Ong tutorial*, Revision 6, Coccon Project, Actility, Nov. 2011.

[44] S. Wahle, T. Magedanz, and F. Schulze, “The openmtc framework - m2m solutions for smart cities and the internet of things”, in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2012 IEEE International Symposium on a*, Jun. 2012, pp. 1–3. DOI: `10.1109/WoWMoM.2012.6263737`.

[45] *Openmtc platform: a generic m2m communication platform*, White Paper, Fraunhofer FOKUS, Nov. 2012.

[46] Z. Shelby and J. Höller, “Embedded devices on the internet of things”, Sensinode and Ericsson, Jul. 2012.

[47] *Eclipselink moxy*, `http://www.eclipse.org/eclipselink/moxy.php`, Accessed at November 2013.

[48] *Hibernate*, `http://www.hibernate.org/`, Accessed at November 2013.

[49] *Toplink*, `http://www.oracle.com/technetwork/middleware/toplink/overview/index.html`, Accessed at November 2013.

[50] *Cocoon development kit*, `http://cocoon.actility.com/documentation/ongv2/development-kit`, Official Cocoon Website. Accessed at November 2013.

[51] *User's guide: etsi m2m scl rest interface*, Revision 3, Coccon Project, Actility, Aug. 2013.

[52] *Sqlite*, `http://www.sqlite.org/`, Official SQLite Website. Accessed at November 2013.

[53] *H2*, `http://www.h2database.com/`, Official H2 Website. Accessed at November 2013.

[54] *Sqlite database speed comparison*, `http://www.sqlite.org/speed.html`, Official SQLite Website. Accessed at November 2013.

[55] *H2 performance comparison*, http://www.h2database.com/html/performance.html#performance_comparison, Official H2 Website. Accessed at November 2013.