TÉCNICO LISBOA

Domotic Framework for the Internet of Things

The mHouse Framework

José Diogo Requeijo Dias

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Prof. Renato Jorge Caleira Nunes
Prof. Paulo Rogério Barreiros D'Almeida Pereira

Examination Committee

Chairperson: Prof. António Manuel Raminhos Cordeiro Grilo
Supervisor: Prof. Renato Jorge Caleira Nunes
Member of the Committee: Prof. João Nuno De Oliveira e Silva

November 2017

Acknowledgments

First of all I want to thank my supervisor teacher, Prof. Dr. Renato Jorge Caleira Nunes, for all the time and attention dedicated to me on all meetings that we had, for all the trust that he putted on me, and for all his ideas and contributions to this dissertation work.

Secondly, I want to thank to my university Instituto Superior Técnico and to all my remaining teachers, for all the knowledge and wisdom that they passed to me, and all life experiences that I had on this institution.

Finally I want to thank to all my closest friends that helped me throughout my course, and a very special thanks to my parents, to my brother and to my beloved girlfriend for all the comprehension, attention, motivation and encouragement, on every day of my life, because without them it would had been impossible to complete this master degree.

Abstract

Domotics and Internet of Things (IoT) are two technological fields that have had an exponential growth in recent years. Currently there are many technologies and solutions on both fields which can be used to implement and develop new services and applications. Nonetheless there are still some problems, because many solutions are proprietary and incompatible with each other, others only fulfill part of the problems and requirements of both contexts, and sometimes they are hard to use by anyone that do not have detailed technical know-how.

Therefore, a framework was developed to building a home automation system, which may possibly have hundreds of devices, in a simpler, robust and easier to understand way. This framework, based on the DomoBus solution, is called the ”mHouse Framework” and it allows the users/developers to access and control any home automation device present on their homes, from anywhere in the world and at any time. It is a totally open-source, expandable and generic solution, possible to integrate with the more recent and powerful technologies, services and applications of the Domotics and Internet of Things fields.

Keywords: Domotics, Home Automation, Internet of Things, Framework, DomoBus, Device Interoperation

Resumo

Os conceitos teóricos de Domótica e a Internet das Coisas são já antigos mas nos últimos anos estas áreas tecnológicas têm crescido exponencialmente. Hoje em dia, nestas duas áreas, existem muitas tecnologias e soluções que podem ser utilizadas para implementar e desenvolver novos serviços e aplicações. No entanto existem ainda alguns problemas, pois muitas destas tecnologias e soluções são proprietárias e não são compatíveis umas com as outras, outras apenas resolvem parte dos problemas ou cumprem parte dos requisitos inerentes às duas áreas, e por vezes essas soluções são difíceis de utilizar por alguém que não possua conhecimentos técnicos detalhados.

Assim, foi desenvolvida uma framework que ajuda a construir um sistema de automação para a casa, que possivelmente poderá conter centenas de dispositivos, de uma maneira mais simples, robusta e fácil de entender. Essa framework, baseada na solução DomoBus, tem o nome the "mHouse Framework" e permite aos utilizadores/desenvolvedores acederem e controlarem qualquer dispositivo domótico presente na sua casa, a partir de qualquer lugar do mundo e em qualquer altura. A solução é totalmente aberta (open-source), extensível e genérica, sendo assim possível integrá-la com as tecnologias, serviços e aplicações mais recentes e potencialmente mais potentes, que existem nas áreas da Domótica e da Internet das Coisas.

Palavras-chave: Domótica, Automação de casas, Internet das coisas, Framework, DomoBus, Interoperação de dispositivos

Contents

List of Figures

List of Abbreviations

3G Third Generation of Wireless Mobile Telecommunications Technology. 9

6LoWPAN IPv6 over Low-Power Wireless Personal Area Networks. 7, 8

API Application Programming Interface. vi, 10, 11, 19–22, 32, 33, 35, 40, 42–44, 49, 54–58, 60, 64, 75, 90

ARM Advanced RISC Machine. 17, 19

AWS Amazon Web Services. vi, vii, 4, 16–18, 56, 60, 63, 64, 66

BLE Bluetooth Low Energy. 8, 9

CEBus Consumer Electronic Bus. 25

CM Control Module. 26, 27

CoAP Constrained Application Protocol. vi, vii, 14–16, 38, 42, 43, 46, 54–56, 58, 66, 68, 85, 90–93, 95

CoRE Constrained RESTful Environment. 14, 15

DTLS Datagram Transport Layer Security. 68

EHS European Home Systems. 25

EIA Electronic Industries Alliance. 25

EIB European Installation Bus. 25

FTP File Transfer Protocol. 10

GSM Global System for Mobile Communications. 8

HTML Hypertext Markup Language. 49, 54

HTTP Hypertext Transfer Protocol. vi, 10, 14–17, 38, 40, 42, 48, 49, 54–56, 64, 67, 75, 85, 90, 94, 95

HTTPS Hypertext Transfer Protocol Secure. 67

IaaS Infrastructure as a Service. 12

ID Identifier. 44, 47, 57, 58, 76–90, 92–94

IDE Integrated Development Environment. 33, 58, 62

IEEE Institute of Electrical and Electronics Engineers. 7, 8

IETF Internet Engineering Task Force. 8, 15

IoT Internet of Things. iii, vi, vii, 1–21, 23, 24, 26, 37–39, 56, 60, 63, 64, 66–68

IP Internet Protocol. 8, 13–16, 25, 37, 46, 47, 58, 66, 68, 85, 87

IPsec IP Security. 14

ITU-T International Telecommunications Union Telecommunication Standardization Sector. 5, 6

JSON JavaScript Object Notation. 11, 12, 32, 54, 66, 75–94, 96

KNX KONNEX. 4, 25

LAN Local Area Network. v, 4, 7, 8, 25, 66

LPWAN Low Power Wide Area Network. v, 4, 8, 9, 66

LTE Long Term Evolution. 9

M2M Machine-to-Machine. 6, 9, 10, 15

MAC Media Access Control. 7

mHouse My House. vi, vii, 38–49, 54–56, 59–64, 66–68, 75, 95

MQTT Message Queuing Telemetry Transport. vii, 15–18, 21, 38

MQTT-SN MQTT Sensor Networks. 16

NAT Network Address Translation. 14

NB-IoT Narrowband IoT. 9

NX No Execute. 20

ORM Object Relational Mapping. 40

OS Operating System. 20

OSI Open System Interconnection. 25

PaaS Platform as a Service. 12

PAE Physical Address Extension. 20

PAN Personal Area Network. v, 4, 7, 66

PC Personal Computer. 25–27

Pub/Sub Publish/Subscribe. 20, 21

RAM Random Access Memory. 20

REST Representational State Transfer. vi, 10, 11, 14–16, 32, 38, 40, 42, 43, 49, 54–58, 60, 64, 66, 75, 90

RF Radio Frequency. 25

RPC Remote Procedure Call. 10

SaaS Software as a Service. 12

SDK Software Development Kit. 17, 33, 35, 58, 62, 63

SM Supervision Module. 26, 27

SMTP Simple Mail Transfer Protocol. 10

SOA Service Oriented Architecture. 10

SOAP Simple Object Access Protocol. 10, 11

SQL Structured Query Language. 41

SSE2 Streaming Simple Instruction, Multiple Data Extensions 2. 20

TCP Transmission Control Protocol. 10, 13–16, 38

UDP User Datagram Protocol. 10, 13–16, 38, 66

URI Uniform Resource Identifier. vi, 11, 54, 57, 75–89, 91–95

URL Uniform Resource Locator. 75

UWP Universal Windows Platform. 19

WLAN Wireless Local Area Network. 7

WPAN Wireless Personal Area Network. 8

WS Web Services. 10

XML Extensible Markup Language. 10–12, 30

1. Introduction

1.1 Context and Motivation

The devices needed to enable Smart Homes, such as sensors, actuators, and communication modules, have been available since the 1970s, and the Internet of Things (IoT) was first mentioned in 1999, making these concepts nothing new. However, nowadays several reports from reputable companies and analysts can be found, stating big numbers and promising forecasts for Home Automation and IoT future, making the study of these concepts even more important [1].

According to Cisco forecasts, the IoT will grow faster than any other category of connected devices, reaching 26.3 billion of devices connected to it by 2020 and possibly 500 billion by 2030, and Gartner forecasts that 20.8 billion connected things will be in use worldwide by 2020 [2][3]. The opportunities created by the IoT may have a global economic impact of $11 trillion per year by 2025, across multiple sectors, according to McKinsey Global Institute [4].

On the other hand, although Home Automation is not the only IoT application, according to a Business Insider's research, it is one of the biggest and main sectors of the IoT [5]. They expect that "the number of smart homes devices shipped will grow from 83 million in 2015 to 193 million in 2020" [6].

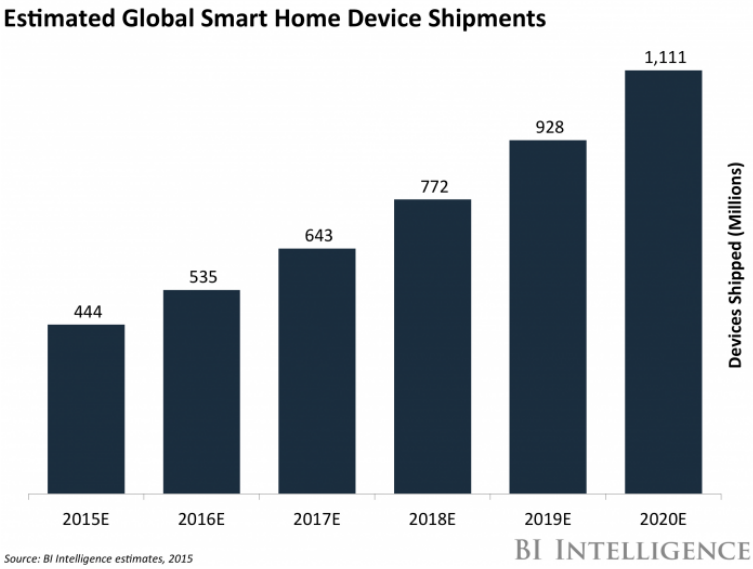


Figure 1: Estimated global smart homes devices shipments [6].

Based on this forecasts and numbers, it is clear that there is a huge market in Home Automation and IoT. One of the reasons why these concepts, although old, are now fastly growing, is the increasing reduction of the cost and size of embedded devices, such as microcontrollers, sensors, actuators, and communication modules, associated with the increasing functionality and power capabilities of such devices.

Throughout the years, research on home automation has been made to further develop it and to enable a broader public to adopt the technologies of smart homes, but several barriers were found in the process. Some of these obstacles are the complexity and the cost of the systems architectures, the inflexibility of the interfaces, the intrusiveness of the installation, the lack of network and device interoperability, and the difficulty achieving security and safety [1][7].

Currently, many technologies are available for home automation, developed by an increasing number of researchers and companies that recognize the field's value. Some of those technologies are standard or open source, and many of them are closed and proprietary. One big problem of these technologies is that they can be incompatible with each other, making interoperability impossible, and none of them dominates the market. Another problem is their cost, because powerful technologies, required on big and complex systems, tend to be very expensive, and most of the times the installation of the automation system is very intrusive, demanding structural work on the home building [1][7].

These issues make users lean to adopt proprietary solutions, which can be more economical and easily work well alone. This raises another problem that is the dispersion and confusion of the market, since each proprietary technology has its own vision and approach, which further decreases interoperability. Furthermore, the new technologies appearing, which promise new and better characteristics, do not often contribute to technology convergence, and that is one of the most important goals to achieve now [8].

Finally, on the other hand, there are current technologies and market solutions usually described as IoT applications, but most of the times they only address some specific set of objects/things and only provide some limited capabilities and functionalities, which have lead to the appearance of "*silos*" on the IoT field, i.e. each one of the subsystems that each technology/solution address, only work with its own specific objects/devices, and it is not aware of the remaining subsystems (interoperability is not correctly addressed), making the most basic idea of the IoT (*everything connected to everything*) impossible [9].

1.2 Objectives

The main goal of this dissertation is the development of a framework that allows the users and developers to easily interconnect smart home devices, taking into account an IoT context. Furthermore, a second goal is to implement, test and evaluate the developed framework with a real example prototype. Therefore, the framework must comply with the main characteristics and requirements of both the IoT and the home automation fields, giving support mainly to interoperability, heterogeneity, ease of deployment and usage, robustness, and global access and control. In order to provide some of these required characteristics, such as interoperability and heterogeneity, the DomoBus [10] approach will be used as a foundation to the work.

In essence, the proposed framework will produce two main results: the specification of an overall system architecture and its subcomponents, and the specification of all interconnection interfaces that the system and its subcomponents should provide. Both of these results should help every user/developer

on building a home automation system, allowing the interconnection of his/her devices and the access to them in a much easier, simpler and robust way, from everywhere in the world and at any time.

Moreover, the definition/specification of the interconnection interfaces is intended to allow the framework to be open, flexible, expandable and generic, enabling the possibility to further create and develop innovative ideas and applications that can easily work *on top of* it or with it. Therefore, the easy and seamless creation of more sophisticated home/IoT services and applications, using the framework, is promoted, e.g. a self-aware (intelligent/smart) home, where the system could make intelligent decisions towards home control, independently of the user direct command issuing.

Taking into account that the proposed solution is directed to home automation but with an IoT context, it doesn't make much sense that the framework's development aims a simply automated home, but instead it should target the concept of a *super-automated home*[1], along with the idea of accessing and controlling every object/thing that is present in the home environment and not only the devices that are usually targeted by current domotic systems. Thus, the "things" that are to be considered by the framework are all the home devices and appliances (which already are considered by current domotic systems), plus all the objects that usually are not taken into account by current domotic systems, such as tables, couches, doors, recipients, clothes, magazines, chairs, boxes, etc., basically every "thing"/object that belongs to or is in the home environment.

With this paradigm, the framework can basically support almost every idea/application that anyone wants to build *on top of* it, or with it, even the most complex, imaginative or futuristic ones, like the example of a self-aware home given before. On that example, with this paradigm, the framework can take advantage on getting and processing information, and acting upon a much more diversified and complete/bigger set of objects/things, thus possibly providing completely different and innovative functionalities, that are impossible to provide and/or implement with the current home automation systems, where the set of connected devices is much smaller and not as diversified. This paradigm also meets the proposed goal for the framework to have an IoT context/essence, as it can be seen as a reflection of the main idea of the IoT (*everything connected with everything*), and thus the applications/functionalities that can be implemented on systems with this paradigm are only limited by developer's/user's imagination.

Note also that, although the framework's main goal is to target the regular home building/space, its IoT essence allows its usage also on other situations/fields, only with minor adjustments, like for example on hotel buildings automation, where the number of control points can be even greater than in a *super-automated home*.

Finally it is important to notice that although security is one of the most critical aspects of IoT systems, this is a vast subject which cannot be fully investigated and developed on the scope of this thesis. Therefore, only the basics of security will be addressed on the development of the solution prototype, by adoption of only some basic security mechanisms already available.

[1] See section 2.2.1.

1.3 Thesis Outline

In this thesis, on Chapter 2, we present a background analysis, where definitions, characteristics, requirements, technologies and market solutions, about both the IoT and Domotics will be presented and discussed. On the IoT section it will be presented subjects like Personal and Local Area Network (PAN/LAN) Technologies, Low Power Wide Area Network (LPWAN) Technologies, Web Services and Cloud Computing Technologies, Application, Transport and Network Layers Technologies, and finally some examples of current market solutions like AWS Greengrass, Microsoft Azure IoT Suite and Google IoT Core. On the Domotics section it will be presented some Domotic enabling technologies, like X10 or KNX, the DomoBus solution, which is crucial for this thesis development, and also some examples of current market solutions like Belkin WEMO, Philips Hue, Samsung SmartThings, Apple HomeKit, Amazon Echo and Google Home.

On Chapter 3 we describe all the work performed and the framework's prototype developed on the course of this thesis. All the explanations and details about how every component works, both individually and together, its role on the overall system, as well as how all the utilized technologies are putted together to achieve the final solution, are presented and discussed.

On Chapter 4 we present and detail all the test cases that were performed to test and verify the built solution.

Finally, all the conclusions and results obtained from the achieved final solution are presented, along with some considerations to future work. All the relevant technical documentation about the achieved solution is also presented on appendices.

2. Background

In this chapter, information about the Internet of Things and Domotics will be addressed, as well as the key technologies and solutions currently available on each of the markets/fields. For each technology, comparisons and context interpretations will be done, taking into account that the main subjects of this work are the IoT and Domotics.

2.1 Internet of Things

2.1.1 Definitions, Characteristics and Requirements

According to the ITU-T Recommendation Y.2060 [11], from the International Telecommunications Union, the Internet of Things is "*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*". Therefore, the IoT can be defined as the interconnection between anything and everything in the world, as long as these connections can be enabled by embedding technology in the physical world [11].

Figure 2: Internet of Things [12].

First of all, it is essential to define the "*Thing*" concept. With regard to the Internet of things, a "*Thing*" is an object of the physical world (physical things) or the information world (virtual things), which is capable of being identified and integrated into communication networks like the IoT [11].

Physical things exist on the physical world and allow connections, sensing and actuation. Basically, it is everything in the world that is capable of being sensed, changed by actuation and/or connected [11]. Examples of this kind of things include the environment, the people and all the things that people can use (shoes, shirts, glasses, etc.), the infrastructures like the houses, the industry facilities, the streets, the buildings, the cities, all the electrical and electronic equipment, and a whole long list of other things.

Virtual things exist in the information world (virtual world) and are capable of being stored, accessed and processed. Examples of this kind of things include multimedia content, virtual data and software [11].

The IoT has a vision with several technological and social implications. In one hand it is expected to considerably integrate leading technologies, such as the ones associated with machine-to-machine (M2M) communication, automatic networking, data mining and decision-making, security, privacy and cloud computing, with leading ones for advanced sensing and actuation. In the other hand, through the exploration of identification, data capture, processing and communication methods, the IoT makes full use of "things" to offer services to all kinds of applications in a way that was unimaginable before, while it has to ensure the fulfillment of all today's security and privacy requirements, thus making a big difference in social terms [11].

The IoT fundamental characteristics, according to the ITU-T Recommendation, are [11]:

- **Interconnectivity:** Everything can be interconnected with anything, by accessing the global information and the communication infrastructure.
- **Things-related services:** The services provided must be *Thing*-related, taking into account the constraints of things, such as privacy protection and semantic consistency between physical things and their associated virtual things.
- **Heterogeneity:** Different devices are based on different hardware platforms and networks, and so they possibly interact with other devices or service platforms in different ways.
- **Dynamic changes:** Devices states and/or the number of devices change dynamically, e.g. when a device is sleeping and then wakes up.
- **Enormous scale:** The far-reaching vision of the IoT is that the number of devices connected with each other will be much bigger than the number of devices connected to the current Internet, and so the communication triggered by humans will noticeably be smaller compared to the communication triggered by the devices.

The relevant high-level requirements for the IoT are [11]:

- **Identification-based connectivity:** The connectivity between *Things* must be established based on identifiers, taking into account that possibly heterogeneous identifiers of different *Things* must be processed in a unified way.
- **Interoperability:** Interoperability needs to be ensured among heterogeneous and distributed systems, allowing the provision and consumption of a diversity of information and services.
- **Autonomic networking:** Processes like self-management, self-configuring, self-healing, self-optimizing and self-protecting techniques/mechanisms are fundamental in the networking control of the IoT, enabling the adaptation to different applications domains, different communication environments and the characteristic big scale of the IoT.
- **Autonomic services provisioning:** Provided by capturing, communicating and processing automatically the data of *Things*, based on rules configured by operators or customized by subscribers. This services may depend on automatic data fusion and data mining techniques.

- **Location-based capabilities:** Some *Things*-related communications and services may depend on the location information of *Things* and/or users.
- **Security:** Due to the idea of every *Thing* is connected, the integration of different security policies and techniques, associated with the variety of devices and user networks in the IoT, is needed.
- **Privacy protection:** Must be ensured during data transmission, aggregation, storage, mining and processing in the IoT, since many *Things* have their owners and users, and sensed data may contain private information concerning their owners and/or users. Although it should not set a barrier to data source authentication.
- **High quality and highly secure human body-related services:** Human body-related services security must be highly enforced and supported, as the data related to human static features and dynamic behavior is much more sensible and critical.
- **Plug and play:** To enable on-the-fly generation, composition or acquisition of semantic-based configurations, allowing the seamless integration and cooperation of interconnected *Things* with applications, and responsiveness to their requirements.
- **Manageability:** The operation process should be manageable by the relevant parties in order to ensure normal network operations.

2.1.2 Technologies

2.1.2.1 Personal and Local Area Network (PAN/LAN) Technologies

On the low level layers (below the network layer), the IoT context usually require technologies that offer good range, simplicity and low price, and at the same time low power consumption and strong security, as the expected number of devices is huge and so each device should last long, be inexpensive, simple and secure. Several technologies are currently available to be used with these requirements in mind, e.g. ZigBee, Z-Wave, Wi-Fi, Bluetooth and 6LoWPAN.

ZigBee is a open radio frequency communication specification, especially designed for Wireless Personal Area Networks (WPANs), based on IEEE 802.15.4 standard, and thus very similar to Bluetooth. It takes full advantage of the lower layers (MAC and physical radio layers) specified by 802.15.4 and adds higher layers such as logical network, security and application layers [5]. Although ZigBee is simpler, cheaper, less power consuming and more secure than Bluetooth, it has lower range, bandwidth [13], and presents some drawbacks, such as it's lack of native support in most consumer devices like smartphones, where Bluetooth or Wi-Fi are commonly supported.

Another solution is Z-Wave, which is a proprietary protocol specifically designed to residential and commercial buildings automation. It is a robust wireless protocol which target characteristics like low-power consumption, low protocol overhead, cheap device manufacturing cost, simple setup, security, low latency and decreased interference with other radio frequency network protocols, such as Wi-Fi, ZigBee or Bluetooth [5][13].

One of the currently most common wireless protocols worldwide is Wi-Fi, which is a protocol for Wireless Local Area Networks (WLANs), based on IEEE 802.11 standard, defined by the Wi-Fi Alliance.

It was firstly designed as the “Wireless Ethernet”, i.e. for high speed communication between devices in a range up to a few hundred meters [1][13]. The IEEE 802.11 standard is subdivided in several amendments (versions), each one representing an evolution of the main standard. The most common versions on home networking are 802.11b, 802.11g and 802.11n [13]. The latest versions of the standard offers data transfer rates up to the order of Gbit/s but, on the other hand, the power consumption of the Wi-Fi devices is much higher in comparison with other protocols such as Bluetooth or ZigBee [1][5].

Nonetheless, recent Wi-Fi modules have become available that can provide Wi-Fi communication with significantly lower power consumption, which can be a major key to Wi-Fi usage on IoT systems, given its popularity. Moreover, an ongoing amendment (802.11ah) aims to operate on the sub-1GHz band, providing greater range and, at the same time, lower power consumption, making it ideal for the small (constrained) devices usually encountered on the IoT [5][14].

Competing on popularity with Wi-Fi is Bluetooth, which is a standard for wireless short-range communications on Wireless Personal Area Networks (WPANs) based on IEEE 802.15.1 standard. Currently it is maintained by the Bluetooth Special Interest Group (SIG) and the most recent versions of the protocol ($>$4.0) were, and still are, being developed mainly towards the IoT, thus focusing on privacy, security and energy efficiency [5]. A subprotocol of Bluetooth v4.0, known as Bluetooth Low Energy (BLE), focus on low power consumption and low hardware development costs, and therefore is a key enabling technology for the IoT [1]. On June 2016, announced a new version of Bluetooth (the Bluetooth 5), which is even more IoT directed than the previous Bluetooth versions [14].

Finally, there is 6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks), which is a set of standards, defined by the IETF, that enable the efficient use of IPv6 on constrained wireless networks and devices [15]. LoWPANs are low cost communication networks that allow wireless connectivity between devices using the IEEE 802.15.4 protocol [16][17]. These networks are usually built with constrained devices, and thus are characterized by short range, low bit rate, low power and cost. Usually the devices employ 802.15.4 radios that are limited in computational power, memory, and energy [15].

As result of the Internet Protocol popularity, an effort was made to develop the capability to use IP (more precisely IPv6) on these constrained networks and devices, that usually use the IEEE 802.15.4 protocol. 6LoWPAN was built from this effort and it defines an adaption layer and the optimization of the related protocols [15]. It is seen as an important set of standards for the IoT, because IPv6 provides a seamless and extremely efficient way merge the new constrained/IoT devices and networks to the current Internet services, networks, and devices, taking into account all the new characteristics, requirements and paradigms [15].

2.1.2.2 Low Power Wide Area Network (LPWAN) Technologies

A Low Power Wide Area Network (LPWAN) is a wireless wide area network focused on the interconnection of constrained devices, within a range much greater than of a Personal Area Network (PAN) or of a Local Area Network (LAN). Thus, technologies for LPWAN are specialized to provide wider range, greater power efficiency and lower cost than the traditional mobile networks that work with GSM,

3G and LTE, and they are usually designed for Machine-to-Machine (M2M) networking environments [18][19][20].

On a LPWAN the data transfer rates and the power consumption of the connected devices are very low, reducing the cost and the energy wasting on each one of these devices which, in IoT scenarios, are usually located in remote areas where battery powering is imperative. This is a major difference comparing to the traditional mobile networks, which in turn are designed for a variety of services that can be unnecessary to IoT applications, like mobile voice and video streaming with high-speed data transmissions. LPWANs can also support more devices over a larger coverage area than the traditional mobile networks, as the connected devices need less bandwidth, transmission power and data rates [18][20].

The technologies that were previously discussed, such as ZigBee, Wi-Fi and BLE, are adequate for consumer-level IoT applications, i.e. where the number of connected devices is in the order of dozens/hundreds, and where each one of the connected devices can be "*less constrained*" and have relatively higher costs, than on industrial and commercial IoT applications, where the number of connected devices is in the order of thousands or even millions, and thus the cost and power efficiency of each device and of the overall network, is much more critical [18][20]. For these "larger" IoT applications, LPWAN technologies and networks are crucial for efficiency and possibility for deployment and proper work [19][20].

Currently there are many technologies/standards working and being developed to LPWANs. One of this technologies, provided by the 3rd Generation Partnership, is NarrowBand IoT (NB-IoT), which is a narrowband radio technology focused specifically on indoor coverage, low cost, long battery life, and enabling large numbers of connected devices, using the cellular telecommunications bands (licensed spectrum) [18][19].

There are also some other proprietary radio technologies, such as SIGFOX and LoRa, which operate on ISM frequency bands, and have been developed and designed specially for machine-type communication applications, addressing the ultra-low-end sensor segment, with very limited demands on throughput, reliability or quality of service [18][19][21].

One of the disadvantages of this kind of technologies for consumer-level IoT applications, is that sometimes, to achieve less power consumption, lower cost and greater coverage, some constraints on data rates, amount of messages exchanged per unit of time, and delay, are made, which may not be justifiable, or even proper, for the majority of the consumer-level IoT applications, like for example home automation.

2.1.2.3 Web Services and Cloud Computing

The IoT concept is widely ground-based on Internet, Web and M2M concepts and methodologies, and it can be seen as "a future evolution" of today's Internet. Two currently main key-concepts of today's Internet and Web technologies are Web Services and Cloud Computing, and consequently these concepts reveal themselves as key-ones on the IoT context as well.

Web Services (WS) are software systems/applications designed to support interoperable M2M interaction over a network, usually with a client-server architecture paradigm [22]. The purpose of WS is to allow server machines to provide some functionalities, or services, to the accessing entities (clients), and so each service has an interface (API) which describes its functionalities and can be used by clients to access and use its functionalities [13][23].

Currently there are many technologies/paradigms that allow the implementation of WS. Some of most well known ones are: Remote Procedure Call (RPC), SOAP (Simple Object Access Protocol) and REST (Representational State Transfer).

The first approach, RPC, enables inter-process communication between threads or processes on different hosts [24]. The client-server paradigm is used, as a client can "call" different procedures that will run on a remote host (the server). In the WS context, a RPC service makes available a service descriptor that is used on the client side. The procedures that become available by the service may contain a list of typed parameters, making the client tightly coupled to a specific server/interface [25][26]. This issue is one of the major problems of using a RPC approach on an IoT system, where the different components ("Things") should not rely or be tightly coupled with one another, due to the IoT inherent heterogeneous essence.

Another approach is SOAP. It is a protocol that also uses a WS Message API style and differs from RPC by avoiding the direct coupling to remote procedures as done in RPC [25]. SOAP usually uses Extensible Markup Language (XML) to represent its messages and HTTP to communicate them, although a variety of other protocols such as SMTP, FTP, TCP, UDP or even proprietary protocols can be used [1][13][24][26]. To support client-server communication SOAP specifies how to use the HTTP POST method for the request messages and how the response should be formatted, for the reply messages. This combined use of XML and HTTP is also a well-known standard "protocol" for client-server communication over the Internet [24]. Thus, as SOAP has less coupling between the involved parties, can be used with a large variety of protocols and uses a unified and generic way to represent the messages (XML), it is seen as a much better choice to work in the IoT context than RPC, as it offers interoperability through more uniform and generic interfaces.

Although SOAP proves to be, by essence, a much better choice than RPC to IoT, both of them are overcomed by REST, which is currently the most well-known and used paradigm/technology related to Web Services, as it presents many more advantages than the remaining ones. REST is an architectural style introduced by Roy T. Fielding in 2000 [27], and was developed along with HTTP [24][25]. It ensures most of the desirable properties for a Service Oriented Architecture (SOA) [23], like the generality of interface, device autonomy and seamless addition of devices. These characteristics can lead to systems much more scalable, but yet simpler to understand, making REST the best suited paradigm to use on the IoT and domotic systems, where the heterogeneity, dynamic change and enormous scale of systems are characteristic, and where there is a demand for interoperability, plug and play functionalities, and autonomic networking and services provisioning [1][13]. As it is based on a resource approach, services with a Resource API style usually adhere to REST principles, although not every Resource API can be considered RESTful.

In order to achieve the proposed goals, REST define a set of five constraints which must be fulfilled to obtain a RESTful API [13][25][27]:

- **Client-Server:** The division of concerns of the client and the server must be well defined. This improves the portability of the user interface across multiple platforms and the scalability of the overall system, by simplifying the server components.
- **Stateless:** The communication between the components must be stateless by nature and cannot rely on any stored context on the server, such that each request must contain all of the information necessary to be understandable. This improves the visibility, reliability and scalability of the overall system and interfaces.
- **Cache:** The data within a response to a request must be labeled as cacheable or non-cacheable. If a response is cacheable it can be reused by the client or an intermediate device to respond to an equivalent request. This improves efficiency, scalability and user-perceived performance as required on the IoT as well.
- **Uniform Interface:** All the components should use and present a generic and uniform interface to their resources. Applying the principle of generality to the component's interface simplifies the overall system architecture and improves the visibility of all interactions. This principle leads to four architectural sub-constraints that are needed to guide the behavior of the components:
  - each resource must be identifiable (ex: with an URI);
  - the manipulation of the resources should be done through representations of themselves;
  - the exchanged messages on the system should be self-descriptive;
  - hypermedia should be the engine of application state.
- **Layered System:** A layered (also known as multi-tiered) architecture should be used. Each component is constrained within a layer and cannot "see" beyond the immediate layer with which it is interacting. This places a bound to the overall system complexity, promotes substrate independence and enables intermediate-based load balancing, monitoring, and/or security checking.
- **Optional Code-On-Demand:** Client functionalities can be extended by downloading code or scripts from the server, and run them on the client side. This simplifies clients by reducing the number of features required to be pre-implemented, and improves the system extensibility.

As a direct result of the fourth constraint, like usually SOAP uses XML, REST messages are usually represented with both XML and/or Javascript Object Notation (JSON). Both of them are specifications to enable interoperation, storage, and data transfer between different systems.

XML was designed to be simple, generic, structured, human and machine readable, and capable of being used over the Internet. It uses a textual data format with Unicode encoding and has the ability to define formal structural and semantic definitions for metadata and information models [13].

JSON is a lightweight, language-independent, text format that facilitates structured data interchange. To describe and define the data that is to be interchanged, JSON definition is based on the

fact that almost every programming language have a small number of common data representations, such as lists, strings (or sequences of characters), numbers (or sequence of digits), and in some cases objects or structures [13].

Despite the XML's power, the simpler, but still powerful, definition of JSON make it a much more appealing format for the data interchanging in IoT systems. JSON is much simpler than XML because it has a much smaller grammar and maps more directly onto the data, it doesn't have much overhead as XML since it isn't a markup language, it is easier to read and write by both humans and machines, making it ideal to process by the usual IoT devices which have constrained capabilities, and finally it is data-oriented, rather than document-oriented like XML. Although, JSON simplicity may sacrifice some semantic context, which sometimes should be taken into account when deciding between one of them [25].

Another key technological concept related to the IoT is Cloud Computing. It is the concept of delivering on-demand computing resources over the Internet (e.g. anything from user applications to data centers). It is based on a pay-per-use ideology, so usually the resources are "*elastic*", i.e. the overall system scales up or down quickly and easily to meet demand, and cloud providers only charge accordingly with the amount of resources used by client companies/users on each instant [16]. Usually Cloud providers make available a huge set of tools and applications that can be used to build complex systems and services, making it easier for small companies/developers to abstract some infrastructural issues, such as load balancing, data replication, quality of service, among others.

Each offered Cloud service usually follows one of three paradigms/models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) [16].

IaaS refers to the concept of providing the computational infrastructure itself, i.e. the capability to use the cloud provider's machines and network (Infrastructure) by the client. This paradigm allows clients to freely use provider's resources like servers, networking, storage and data center space, and thus to deploy and run arbitrary operating systems and applications on his own cloud space (or virtual machine), making this model the most configurable among the three [16].

In PaaS the idea is to provide all the cloud-based hardware and software needed to support the complete lifecycle of building and delivering web-based (cloud) applications. With this paradigm, the client only needs to care about the functional logic of his/her applications, without worry about the cost and complexity of buying and managing the underlying hardware, software, machine or network provisioning, and hosting [16].

The last paradigm (SaaS) is based on the idea of providing applications that run on the cloud, to the client. The clients can access this applications, which are deployed and runned on the cloud infrastructure, from any device with connection to the internet and (usually) a web browser. Once again, with this paradigm the clients does not manage or control the underlying cloud infrastructure and, furthermore, now clients do not manage or control any of the cloud application capabilities, being only able to use them as provided [16].

Using the Cloud capabilities and tools can bring many advantages when someone wants to provide scalable, strong, and accessible from anywhere and at anytime, online applications for the IoT context.

The majority of the current main Cloud Providers like Amazon, Google and Microsoft, have generic IoT applications and services built upon their Cloud Infrastructure, which are totally integrated with their remaining Cloud services, such as services to simply build web and mobile applications and services, to backup, archive and analyze big amounts of data, to simply build and manage databases, to recover data and systems from disasters, to compute with high performance, etc.

Moreover, due to the generic and growing open-source mindset behind the Cloud concept and these big companies Cloud services, IoT applications that are built for a specific scenario with open-source tools, like the proposed framework developed on this thesis, can and should take all the advantages of both the Cloud and Web Services concepts and their technologies, as they simplify the building process, giving the developer time to focus on functionality and application logic, rather than on, for example, deployment, interoperability and scalability details.

2.1.2.4 Application, Transport and Network Layers Technologies

As the IoT is intrinsically and directly related to today's Internet, given its interconnectivity and networking characteristics, the technologies on the upper layers of the protocol stack (Application, Transport and Network layers) should be the ones that offer the better solution to the IoT requirements on each scenario and that, at the same time, provide backwards compatibility, in order to allow current and past legacy services and applications to work with the new ones which are being developed and built with the IoT innovative mindset.

On the Transport and Network layers, the current most well known protocols provide good solutions to the new challenges brought by the IoT. These protocols are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) for the Transport layer and the Internet Protocol (IP) for the Network layer. Both TCP and UDP provide end-to-end message transport from an application running on one machine to an application running on the same or on other machine, and each application is identifiable by an application's port number. The IP enables the addressing of every machine that is connected to the network, providing them a unique number known as the machine's IP address, and giving the appearance of a large and seamless network [24][28].

TCP is connection-oriented, i.e. a connection is established before the actual exchanging of data between the machines, and control messages are used to establish and shutdown connections, control the data congestion and flux, and acknowledge the reception of messages, providing a reliable, flow-controlled, and full-duplex stream transport service [13].

UDP and IP are connectionless protocols, i.e. all messages are exchanged without any previously established connection, making these protocols very simple and lightweight, because only messages with the actual data are exchanged and no other control messages are needed [13]. This simplicity is counter balanced with a *best-effort* delivery paradigm, which means that messages can be lost, duplicated, or delivered out-of-order.

Currently two versions of IP are in use - IP version 4 (IPv4) and version 6 (IPv6). Although today's most widely used version is yet the fourth (IPv4), IPv6 is the more actualized and better suited one on

the IoT context. That is because IPv4 uses addresses only 32-bit long, and so, currently, even with NAT techniques to extend the address space, it is already fully occupied by the machines of today's Internet. In IPv6 the addresses are 128-bit long, 7.9 octillion times bigger than on IPv4, providing a sufficient address space to address every single device of the expected huge IoT, in addition to offer other improvements that can also be key functionalities on IoT systems (ex.: integrated support for IP security - IPsec) [13][28].

Therefore, in the IoT context, UDP is a much better suited solution than TCP, given its simplicity and lightness, which come from being connectionless, as the overhead of messages needed to exchange the actual data is null compared to TCP, where most part of the exchanged messages are not data related, but are although required to the connection establishment and termination, and data congestion and flux control. Thus, UDP is best suited to systems where the exchanged messages are small (ex.: IoT systems), in contrast with TCP that is better suited to systems where the exchanged messages are bigger (ex.: pure Web applications). Furthermore, TCP is best suited to communication between only two machines at the same time (one sender/one receiver), whereas UDP allows efficient multi-point communications, i.e. one to one, one to many and many to one, which can be an advantage, as in IoT systems this kind of interactions can be useful or even required.

Moving up to the Application layer, the current most well known protocol for data transfer on the Web, which can be seen as a possible solution for some IoT applications, given is popularity and strength, is the Hipertext Tranfer Protocol (HTTP). It allows to exchange contents, such as hypertext, with a request-response pattern, typically using the Transport layer (TCP) port number 80 to send and receive the messages [13]. In HTTP, each request has a header and an optional body. The header of the request always contains: a request method specifying the action to be made on the server, which can be GET, POST, PUT and DELETE (following the CRUD[1] paradigm); an URI identifying the resource on which the action is to be made; and also other relevant metadata [13]. The optional body is dependent of the request method and, when present, should have some information about the requested entity/action (ex: if REST is being used, the request body can have the representation of the resource being transferred). HTTP responses have also a header and a body. The header of a response has always a status line and a code, which identify the response's type, i.e. success of the requested action or type of error when occurred [13].

As HTTP was mainly built for today's Internet and Web, and as the IoT has some new (specific) characteristics which need to be taken into account, like the regular usage of constrained devices on huge constrained environments (i.e. enormous networks and systems, constituted by small and simple devices), some new Application layer protocols were and are still being developed towards this new requirements and characteristics that emerged from the IoT concept.

One of these "new" protocols is the Constrained Application Protocol (CoAP), which is an application layer protocol especially designed for Constrained RESTful Environments (CoRE). These environments are those where constrained resources are used, i.e. where the connected devices are small, work with low-power, low processing and memory capabilities, and where the networks can be very

[1] Acronym for the four basic functions of persistent storage: Create, Read, Update and/or Delete.

lossy (e.g. the IoT), but, at the same time, where Web technologies and REST paradigms provide good advantages in order to be used [24][29]. This protocol was proposed on 2013 and is still being maintained by the Internet Engineering Task Force (IETF) CoRE Working Group [30][31].

CoAP is especially relevant as it is a RESTful protocol which can be easily transformed to HTTP, allowing enhanced interoperability between small devices and Web-based services [24]. However, it is important to notice that CoAP is not a mere compression of HTTP, but a whole new protocol which only takes advantage on some specific parts of the HTTP specification. Thus, CoAP has the basic request methods defined by HTTP (GET, PUT, POST and DELETE), and these have the same semantics as on HTTP. Furthermore, CoAP defines the new request method OBSERVE, that a client can use to receive automatic notifications from the server when some resources are modified, without having to "poll" the server to notice those changes [16][29].

On the Transport and Network layers CoAP work over UDP/IP because, in constrained environments, UDP offers better performance than TCP. However, as some applications require a higher quality of service (reliability) than the one offered by UDP, CoAP implements a thin control layer below its request-response model that deals with this issue. This model is depicted in Figure 3 [29]. By using UDP, CoAP also provides multicast capabilities, which further improves its usability on constrained environments, by reducing bandwidth utilization for certain operations.

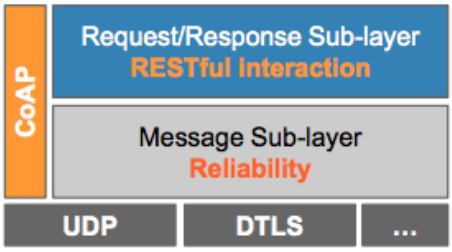


Figure 3: CoAP protocol stack [29].

A second protocol which is relevant on the IoT context is Message Queuing Telemetry Transport (MQTT), which is a lightweight, open, simple and easy to be implemented publish/subscribe messaging transport protocol, making it also a good solution for constrained environments such as Machine to Machine (M2M) communications and the IoT. MQTT runs over TCP/IP [24][32], or over other similar protocols as long as ordered, lossless and bidirectional connections are provided, and this can be a problem because, in some IoT scenarios, the devices may be too constrained to even be capable of establishing this complex connections. On the other hand, the use of a publish/subscribe message pattern provides one-to-many message distribution and decoupling of applications. With this pattern, clients register at a central server for one or more topics in which they are interested, and if one client publishes information for a specific topic, all the clients that subscribed that topic will receive the new message [16].

Both CoAP and MQTT are useful protocols in the context of the IoT, as they are open standards, better suited to constrained environments than HTTP, provide mechanisms for asynchronous communication and run over IP. The main difference between CoAP and MQTT is their essence. CoAP was

designed to keep a RESTful and HTTP-like approach suited to constrained environments, giving CoAP full interoperability with today's Internet/Web. On the other hand MQTT gives flexibility in communication patterns and acts purely as a pipe for binary data, being agnostic to the content of the messages exchanged. MQTT has a centralized architecture as the exchange of messages is always made through a centralized server that handles the topic's registry and the routing of the messages, whereas CoAP has a pure client/server approach, i.e. although in some applications the architecture is a centralized one, a more distributed architecture is possible as well. Finally, MQTT only works with Transport and Network layer protocols which provide ordered, lossless and bidirectional connections - usually TCP/IP. This can be problematic because, in order to provide this complex connections, a greater bandwidth and more computational resources are required from both the network and the constituent devices/components [29]. CoAP, on the other hand, doesn't have this problem because it runs over UDP (or other connectionless protocol), which is in essence better suited to constrained environments. A possible solution to this problem can be also the use of MQTT-SN, which is a version of MQTT protocol that defines a UDP mapping of MQTT, and adds broker support for indexing topic names, allowing also the practical use of MQTT over the 802.15.4 physical layer protocol [32].

2.1.3 Commercial Solutions

Several solutions from different companies can be found currently on the market, directed to the IoT. These solutions are usually provided by some of the biggest companies on the computing and technological field such as Amazon, Microsoft and Google, among others. Usually these IoT solutions are built upon each company's cloud platform, given all the advantages previously discussed about using Web Services and Cloud Computing to support IoT applications.

Amazon Web Services (AWS)

Amazon has some IoT specific services, which are part of the AWS solutions suite, such as AWS Greengrass [33] and AWS IoT Platform [34]. These services help the users/developers to collect and send data to the AWS cloud, making it easy to load and analyze the collected information, and providing the ability to manage all the connected devices, so that users/developers can focus on developing applications that fit their needs.

AWS Greengrass is a software service that allows to run local computations, messaging, data caching, and synchronization capabilities for connected devices in a secure way. With this service, connected devices can run AWS Lambda functions, keep data synchronized, and communicate with other devices securely – even when not connected to the Internet. Using AWS Lambda, Greengrass ensures that the IoT devices can respond quickly to local events, operate with intermittent connections, and minimize the cost of transmitting IoT data to the cloud [33]. It seamlessly extends AWS to devices, so they can act locally on the data they generate, while still using the cloud platform for management, analytics, and durable storage. With Greengrass, developers can use familiar languages and programming models to create and test their device software in the cloud, and then deploy it to their devices. The service can

be programmed to filter device data and only transmit necessary information back to the cloud, while it authenticates and encrypts device data at all points of connection using AWS IoT's security and access management capabilities. This way, data is never exchanged between devices when they communicate with each other and the cloud without proven identity [33].

AWS Greengrass allows to build IoT solutions that connect different types of devices with the cloud and each other. Devices that run Linux and support ARM or x86 architectures can host the Greengrass Core. The Greengrass Core enables the local execution of AWS Lambda code, messaging, data caching, and security [33]. Devices running AWS Greengrass Core act as a hub that can communicate with other devices that have the AWS IoT Device SDK installed, such as microcontroller based devices or large appliances. AWS Greengrass Core devices and the AWS IoT Device SDK-enabled devices can be configured to communicate with one another in a Greengrass Group. If the Greengrass Core device loses connection to the cloud, devices in the Greengrass Group can continue to communicate with each other over the local network. A Greengrass Group may represent one floor of a building, one truck, or one home [33].

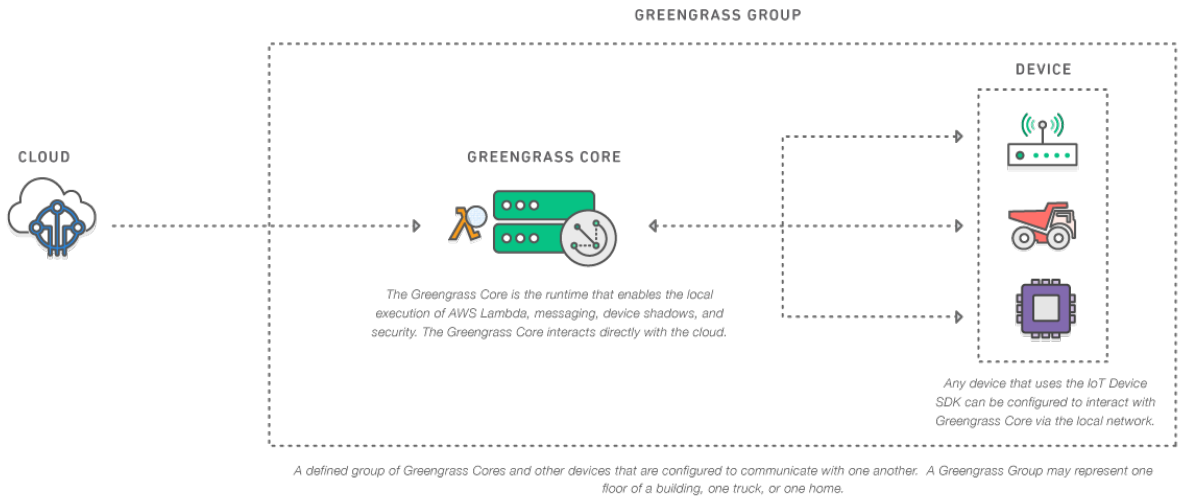


Figure 4: Greengrass Group [33].

A second service, AWS IoT, is a managed cloud platform that allows users/developers to easily connect their devices and securely interact with cloud applications and other devices, supporting several protocols, such as HTTP, WebSockets, and MQTT [34].

It can support billions of devices and trillions of messages, and can process and route those messages to AWS endpoints and to other devices reliably and securely, as it provides authentication and end-to-end encryption throughout all points of connection, so that data is never exchanged between devices and AWS IoT Platform without proven identity. In addition, developers can secure access to their devices and applications by applying policies with granular permissions [34].

It also allows to filter, transform, and act upon device data on the fly, based on business rules defined by the users/developers. Thus, users/developers can update their rules to implement new device and application features at any time, and they can keep track of and communicate with all devices, all the time, even when they are not connected. This is possible because AWS Iot Platform stores the latest

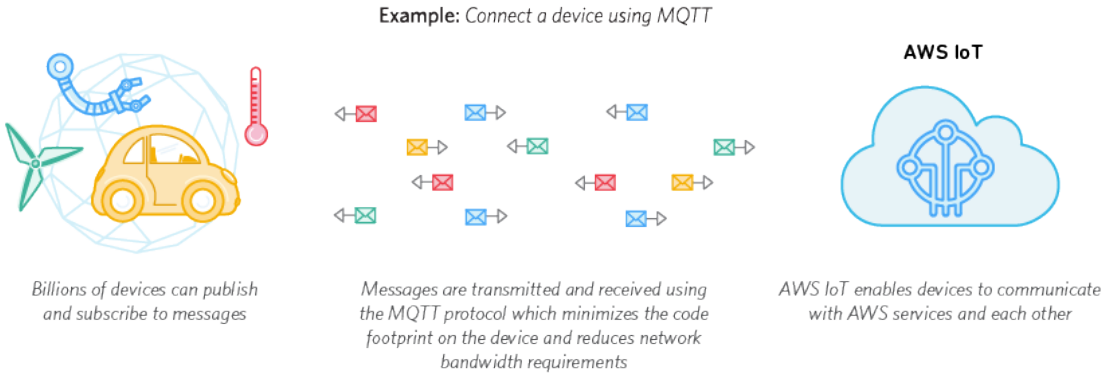


Figure 5: Example of connecting a device to AWS IoT Platform using MQTT [34].

state of every device so that it can be read or set at anytime, making the device appear to developer applications as if it is online all the time. This means that those applications can read a device's state even when it is disconnected, and also allows to set a device state and have it automatically implemented when the device reconnects [34].

The AWS IoT Platform can be easily used along with other AWS services like AWS Lambda, Amazon Kinesis, Amazon S3, Amazon Machine Learning, Amazon DynamoDB, Amazon CloudWatch, AWS CloudTrail, and Amazon Elasticsearch Service, to build big and possibly complex IoT applications that gather, process, analyze and act on data generated by connected devices, without having to manage any infrastructure [34].

Microsoft

Microsoft also provides a large set of IoT services that help users/developers to create and build new applications and solutions. They already provide some examples and predefined customizable solutions, applicable to several industries, such as healthcare, manufacturing, natural resources, retail, smart cities and transportations. Some of the most relevant IoT services provided by Microsoft on this large set of solutions are the Azure IoT Suite [35] and the Windows 10 IoT Core [36].

The Microsoft Azure IoT Suite [35] is an enterprise-grade solution that enables the users/developers to collect data from devices, analyze data streams in-motion, store and query large data sets, visualize both real-time and historical data, integrate with back-office systems, and manage all connected devices [37]. To deliver these capabilities, Azure IoT Suite packages together multiple Azure services with custom extensions as preconfigured solutions. These preconfigured solutions are base implementations of common IoT solution patterns, that help to reduce the time taken by the user/developer to deliver IoT solutions. They are, for example: the remote monitoring solution which enables to monitor the status of devices such as vending machines; the predictive maintenance solution that helps to anticipate maintenance needs of devices such as pumps in remote pumping stations and to avoid unscheduled downtime; and finally the connected factory solution that helps to connect and monitor all user's industrial devices.

By using the IoT software development kits, anyone can customize and extend these solutions to meet his/her own requirements. The users/developers can also use these solutions as examples or templates when developing new and different IoT solutions [37].

Each service that is packaged in the Azure IoT Suite provides specific functionalities. Therefore, the core to Azure IoT Suite is the Azure IoT Hub service [37][38]. This service provides bidirectional (device-to-cloud and cloud-to-device) messaging capabilities, and acts as the gateway to the cloud and the other key IoT Suite services. The service enables the user/developer to receive messages from his/her devices at scale, and to send commands to them. The service also enables the user/developer to manage his/her devices, i.e. for example, the user can configure, reboot, or perform a factory reset on one or more devices connected to the hub [37][38].

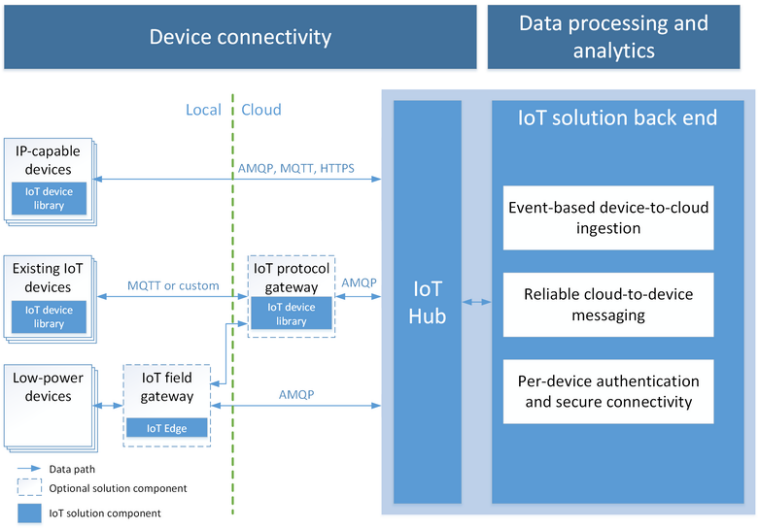


Figure 6: Hub Architecture [38].

The Azure Stream Analytics service [39] provides in-motion data analysis. IoT Suite uses this service to process incoming telemetry, perform aggregation, and detect events. The preconfigured solutions also use this service to process informational messages that contain data, such as metadata or command responses from devices. The solutions use Stream Analytics to process the messages from devices and deliver those messages to other services [37][39].

The Azure Storage and Azure Cosmos DB services provide the data storage capabilities. The preconfigured solutions use blob storage to store telemetry and to make it available for analysis. The solutions use Cosmos DB to store device metadata and enable the device management capabilities of the solutions [37].

Finally, the Azure Web Apps and Microsoft Power BI services provide the data visualization capabilities. The flexibility of Power BI enables the user/developer to quickly build his/her own interactive dashboards that use IoT Suite data [37].

Outside the Azure scope, Microsoft also provides another IoT service which is the Windows 10 IoT Core [36]. It is a version of Windows 10 that is optimized for smaller/constrained devices with or without a display, and that runs on both ARM and x86/x64 devices. It utilizes the rich, extensible Universal Windows Platform (UWP) API to build the solutions, and it also supports the easy-to-use

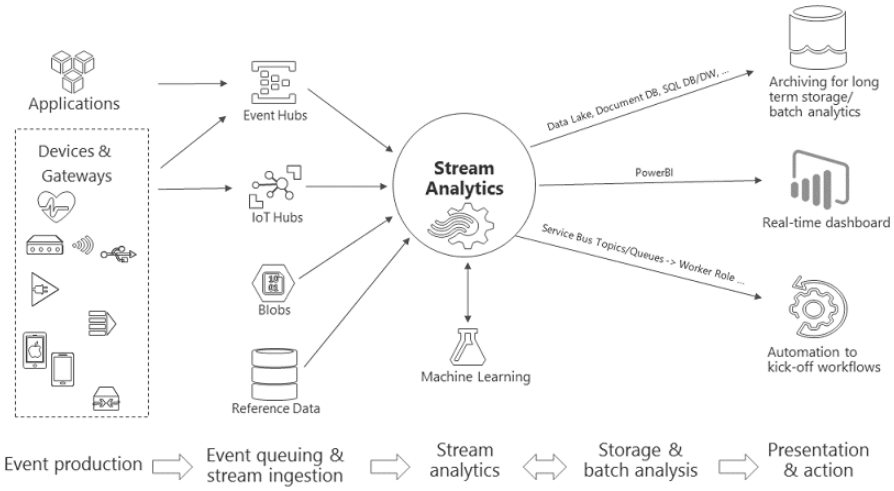


Figure 7: Stream Analytics [39].

Arduino Wiring API used in Arduino sketches and libraries for direct hardware access. To develop applications, users/developers can use the latest free Visual Studio Community Edition which includes universal application templates, a code editor, a powerful debugger, rich language support, and more [40].

This way, developing applications for the small/constrained IoT devices is much easier and fast because developers can take advantage on all the Windows strength. However, not all “small” devices can run the Windows 10 IoT Core because it requires a 400 MHz or faster processor (x86 requires PAE, NX and SSE2 support) and, depending on whether it is run in a headed or headless mode, i.e. if the devices have a video display and use Windows video subsystem to address it, or if they have no display, the memory requirements are 512 MB RAM (256 MB free to OS) and 2 GB Storage, or 256 MB RAM (128 MB free to OS) and 2 GB Storage for memory, respectively to the headless and headed modes [40].

Google

Based on its cloud platform, Google also provides IoT services to help users, developers and companies to build IoT applications and solutions, such as Google Cloud IoT Core [41] and Android Things [42].

The Google Cloud IoT Core [41][43] is a fully managed service that allows users/developers to easily and securely connect, manage, and ingest data from millions of globally dispersed devices. In combination with other services on Google Cloud IoT platform [43], it provides a complete solution for collecting, processing, analyzing, and visualizing IoT data in real-time to support improved operational efficiency.

Device data captured by Cloud IoT Core gets published to Cloud Pub/Sub for downstream analytics. Users/developers can do ad-hoc analysis using Google BigQuery, easily run advanced analytics and apply machine learning with Cloud Machine Learning Engine, or visualize IoT data results with rich reports and dashboards in Google Data Studio, which can help users/developers to improve opera-

tional efficiency, anticipate problems, and build rich models that better describe and optimize businesses [41][43].

The Cloud IoT Core allows to securely connect a few or millions of user globally dispersed devices through protocol endpoints that use automatic load balancing and horizontal scaling to ensure smooth data ingestion under any condition. It supports the standard MQTT protocol, so users/developers can use their existing devices with minimal firmware changes, and it runs on Google's serverless infrastructure, which scales automatically in response to real-time changes and adheres to stringent industry-standard security protocols that protect business data [41][43]. Furthermore it brings devices data into user's everyday business through a secure, intelligent, and responsive IoT data pipeline. Support is also provided to a wide variety of embedded operating systems, and Cloud IoT Core works seamlessly with Android Things, which can be used to push out automatic firmware updates, or easily push user's own device updates. It also provides out-of-box support for devices from leading manufacturers like Intel and Microchip, and automatic changes can be also triggered to devices based on real-time events using Cloud Functions workflows. Finally it allows to centrally manage the user's entire IoT data network from a single pane of glass [41][43].

The Cloud IoT Core has two main components: a device manager and a protocol bridge [41].

The device manager allows individual devices to be configured and managed securely in a coarse-grained way, and management can be done through a console or programmatically. The device manager establishes the identity of a device, and provides the mechanism for authenticating a device when connecting. It also maintains a logical configuration of each device and can be used to remotely control the device from the cloud [41].

The protocol bridge provides connection endpoints for protocols with automatic load balancing for all device connections. It has native support for secure connection over MQTT, and it publishes all device telemetry to Cloud Pub/Sub, which can then be consumed by downstream analytic systems [41].

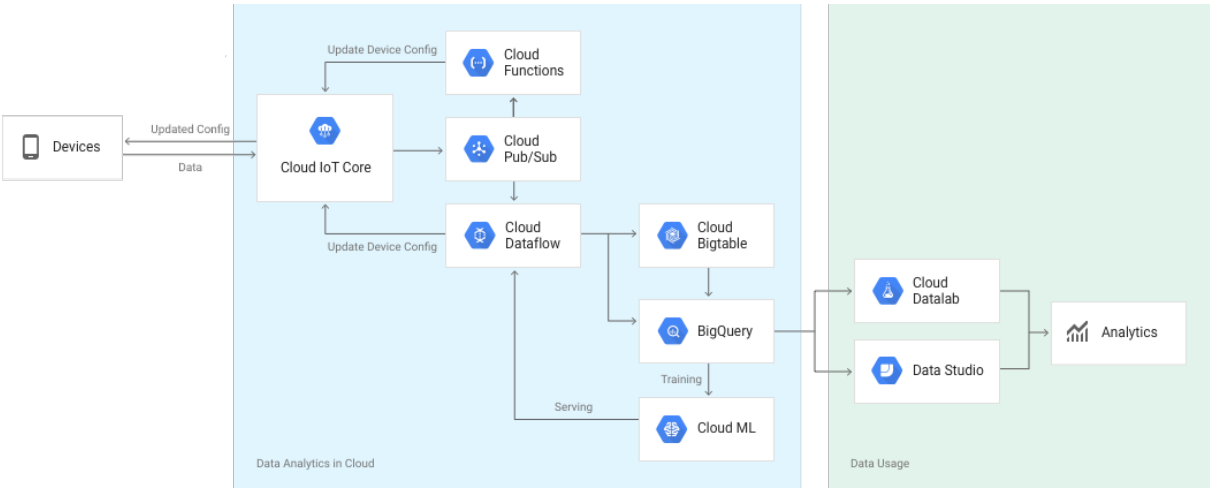


Figure 8: Example overall data flow diagram [41][43].

Google also provides the Android Things service [42], which makes the development of connected embedded devices easier, by providing the same Android development tools, best-in-class Android framework, and Google APIs, that make developers successful on mobile development [42][44].

Applications for embedded devices bring developers closer to hardware peripherals and drivers than smartphones and tablets. In addition, embedded devices typically present a single app experience to users. Thus, Android Things extends the core Android framework with additional APIs provided by the Things Support Library, which allow applications to be integrated with new types of hardware, not typically found on mobile devices. Furthermore, the Android Things platform is also streamlined for single application use. System applications are not present, and the developed application is launched automatically on startup to immerse users in the application experience [42][44].

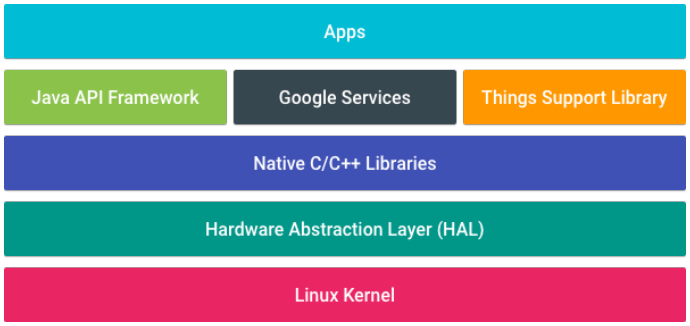


Figure 9: General Architecture [44].

Regarding all these types of solutions, one possible big disadvantage is that, sometimes, they may require some detailed technical know-how from the users/developers and some time to fully study and understand their way of work. Thus, they are perfect for industrial solution's development, which usually is performed by technological companies with high technical knowledge, during a long period of time and with many technological resources, but they may be very hard to use on simpler user-case solutions, where the user/developer may work alone, not having any technical knowledge, time to spend on developing it, and may have fewer and/or cheaper resources.

2.2 Domotics

2.2.1 Definitions and Comparison with the Internet of Things

Domotics, also known as Home Automation, is an extension of building automation which can be defined as the ability to control the several systems that compose the home building [13][16]. An Automated Home can be referred as a Smart Home, where an automation system enables the user to control every device/appliance present on the home environment, such as doors, windows, lights, electricity, gas and water meters, thermostats, air conditioners, sprinklers, washers and dryers, TVs, etc. To achieve this automation, the smart homes have to be equipped with devices that enable connectivity, sensing, and control, so there can be an interconnection network that allows the user to take control of the home.

The propagation of home automation has been growing due to the benefits that it offers. It can ease most of the daily tasks, such as automatically control the garden watering, turn on the exterior lights when the night falls and turn them off when the sun rises, establish and control the correct wanted

Figure 10: Domotics.

temperature, humidity and lighting levels, or turn off lights in rooms that are empty, among others. This way, a smart home can increase the user's comfort and, at the same time, improve energy, gas and water savings [8]. Furthermore, home automation can enhance both home's and user's security, allowing the detection and warning on emergency situations, like gas leakages, fires, flooding or intrusion. It has even greater added value to older people, or to people with reduced mobility or other physical limitations, by improving their autonomy and quality of life [8].

The home automation concept, as well as other types of automation, like building and city automation (smart cities), has some similar characteristics and crucial differences with the IoT concept. Both of them are based on the capability of interconnecting simple devices, which are somehow connected to the physical world, and thus to build a network that provides convenient and easy access to the physical things/resources.

A first difference between IoT and domotics is that usually the majority of the actions taken on a domotic system is towards actuate on some resource, like turning a light off or open a blind - see Figure 11. On IoT systems the opposite is verified, as the majority of the actions is towards retrieving data from the physical environment, like getting the temperature or the humidity of the air, in order to get insights and conclusions about that data, and so the most relevant communications are from the physical devices/things to the user controller. However, both ways of communication are present on both concepts, depending on each specific scenario, and that can also be seen as a similarity between them. On the other hand, while domotics is a much more restricted concept, as it address only home automations, the IoT concept is a much broader one by essence, as it means the *network/internet of everything*, and if domotics is only regarded as the network of the home automated devices, it can also be seen as part of the IoT concept.

A second difference between "*classic*" domotics and the IoT is the range of action. While domotic systems usually actuate only within the house space/location and its surrounding areas, IoT systems are typically bigger ones, with a broad actuation range distributed throughout a large area. A third difference is the number of devices involved, because domotic systems usually only have dozens of devices, where the IoT ones possibly have thousands or even millions of devices - see Figure 11. Although, when

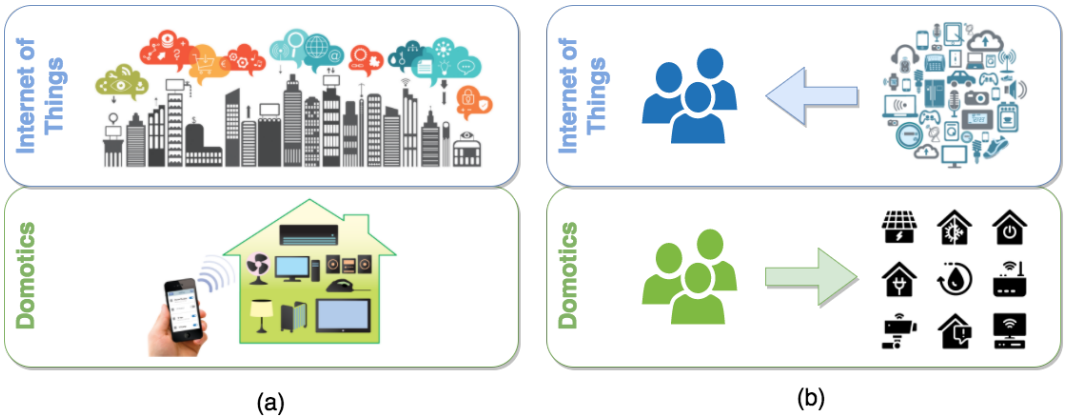


Figure 11: Comparison between Domotics and IoT: (a) - Range and number of devices comparison; (b) - Direction of the majority of communications and number/type of devices comparison.

thinking about *super-automated homes*, i.e. smart homes where the number of control points (devices) is huge comparing with the common basic "*classic*" automated homes, and where the overall system is much more complex and provides a richer set of integrated services, the merge between the IoT and domotics concepts is clear, because now within each house will possibly exist hundreds of connected devices, and by adding the several devices from all the houses, an IoT subsystem can be created (e.g. the set of all smart houses on a city).

A fourth difference can be found on the devices used on each field. IoT devices are usually very small, computationally constrained, cheap and "*dumb*" devices, and they usually work as a "*cloud*" of thousands or millions of devices, where each one only performs a simple function, like reading the temperature and reporting it. On the other hand, domotic devices can be small, computationally constrained and cheap devices too, but can also be smarter devices, with stronger computationally capabilities and sometimes expensier and bigger in size, like for example smart TVs or fridges. Therefore, regarding the type, size and price of the devices used, domotics is a much broader concept than IoT, as the number of devices used on domotics is usually smaller and so the price of the overall system tend to be smaller too, even though the devices used could be bigger and expensive.

Finally, an effort is being made on both worlds to develop technologies and devices that are simpler and easier to use, deploy and run, in order to the interested user, which may not have detailed technical know-how or experience, be able to use, develop, install or configure them conveniently and straight away, adapting them to his/her specific wishes, needs or requirements.

2.2.2 Technologies

As home automation systems are usually localized only within the house space/location and its surrounding areas, the technologies that were developed and that are usually pointed as the specialized one's for home automation, are mostly low-level technologies to work and communicate in relatively low ranges, with low bandwidth, and with simple devices.

One of the first technologies that appeared towards home automation is X10, which was designed and proposed in 1975 by Pico Electronics of Glenrothes (Scotland) for communication among home

devices and appliances. It is a low-level technology which uses power line wiring for signaling and control, and the message signals involve brief radio frequency bursts, representing digital information [45]. A wireless radio based protocol transport for X10 was also defined later. Despite its simplicity, it has very limited capabilities and, over the years, it has experienced some challenges such as a history of poor performance, distance limitations, power phase limitations, and sporadic reliability [46][47][48]. Nonetheless, due to its age, it is still a technology widely available to use on current home automation systems [49].

As the popularity of domotics increased over the years, other technologies were designed as an improvement to X10. One of those is Consumer Electronic Bus (CEBus), which is a set of standards and communication protocols, defined by the american Electronic Industries Alliance (EIA) in 1992, for electronic devices to transmit commands and data. It also specifies communication through power line wire, and now through low voltage twisted pair (copper cable), coaxial cable, infrared, radio frequency (RF), and fiber optics too [45][48]. Another one of those technologies is Insteon, which was developed in 2004 by SmartLabs, and was designed to be compatible with X10 systems. The innovation here was to expand the home automation capabilities to include new features like wireless communication. Due to its compatibility with X10, it is capable of transmitting X10 messages over the power line wire, which can be regarded as a disadvantage due to the legacy X10 issues, adding the fact that it introduces RF communication, which has some problems by itself as well [49][46][47].

Today, despite all the others, one of the most used technologies on home automation is KNX (Konnex), which is an open standardised OSI-based network communication protocol for building automation [50]. It is a successor to, and a convergence of three older standards: the European Installation Bus (EIB) [48], the European Home Systems control (EHS) [48] and BatiBus. KNX defines several physical communication medias, such as twisted pair wiring (inherited from BatiBUS [48] and EIB Instabus standards), power line wire networking (inherited from EIB and EHS, similar to that used by X-10), radio frequency, infrared and Ethernet (also known as EIBnet/IP or KNXnet/IP). Its main advantages are the independence of any particular hardware platform, which means that a device can be controlled by anything from an 8-bit controller to a PC, and the focus on safety and energy savings [45][49][48].

ZigBee and Z-Wave, being LAN technologies, are often described also as domotic specific technologies. Both of them use wireless mesh networking, which utilize switches as repeaters, and thus the reliability, range, as well as cost of the network, is directly proportional to the number of units installed [49]. When using Z-Wave in a domotic context, some careful considerations has to be made, because there are reports of trouble with Z-Wave networks containing more than 30-40 connected nodes (devices), despite theoretically it could contain up to 232 nodes, and that some of the products from certain manufacturers are incompatible with other manufacturer products, increasing the difficulties with the usage, configuration and installation of a Z-Wave network, due to the increased complexity of this technology compared to X10 or KNX [49][48]. ZigBee as also the disadvantage of being the most expensive technology comparing to the other ones, due to its complexity [47][48].

All of the above technologies are currently pointed out as smart home specific and can work well on such environments. Nonetheless, one problem of these technologies is that, usually, only the

automation devices support them, and the other user devices, such as smartphones and PCs, do not support them by essence, and thus, the control of the home environment cannot be done straight away with the typical user devices. Another problem is that these technologies usually work on a centralized and localized way, i.e. they cannot simply be accessed from anywhere and at anytime by the user or by any other service through the Internet, and allowing that could bring advantages like providing smarter and better applications/services to the home user, which can be based on the data collected from the automation devices, and can have typical user-friendly interfaces, like smartphone apps or websites. Of course, on the other hand, the majority of the smart home devices available on the market work mainly with these technologies, because they usually solve some traditional home automation problems, like the installation intrusion, simplicity to configure, deploy and replace, among others [49][46][47].

When putting "*classic*" domotics and IoT side to side, technologically speaking, the conclusion is that, if the use of the above technologies is a must for some reason, unless some gateways are used/developed, the devices working with these technologies cannot be simply/directly integrated on an IoT context, due to all technological incompatibilities, as these technologies do not overcome some IoT requirements because they were not built with the IoT in mind. On the other hand, technologies like Wi-Fi or Bluetooth, which are mainly pointed as Internet/IoT enabling one's, can also be easily used on a smart home context, making the automation devices straight away compatible with the remaining IoT technologies, as they have a much more growing popularity and availability than the legacy smart home technologies, and they can also solve some of the those technologies problems (e.g. the compatibility with the user typical devices, as almost all current smartphones and PCs support Wi-Fi, Bluetooth or some kind of Internet access).

2.2.3 The DomoBus Solution

DomoBus [10][51] is a home automation system that was developed as an academic project by Prof. Dr. Renato Jorge Caleira Nunes, on Instituto Superior Técnico, Lisbon. "*Its development was carried in order to overcome many difficulties felt while trying to access standard products for testing and experimentation purposes*" [51]. It is focused on the ability to interconnect the devices with other networks, to promote interoperation and to accomplish an integrated solution.

The general architecture of the DomoBus is illustrated in Figures 12 and 13. It is a distributed system composed, essentially, by Control Modules (CM), which operate on the "Control Level" (Figure 12), and Supervision Modules (SM), which operate on the "Supervision Level" (Figure 13), and they are interconnected by a communication network that allows them to interact and cooperate with each other [51][52].

The Control Modules are small microprocessor-based boards (e.g. Arduino) that directly connect to sensors and actuators, which they can use to sense and/or act over the physical environment/things, such as, switches, lights, motors, pumps, electric heaters, air conditioners, among others. This modules are quite flexible as they are able to run different applications and perform different functions. For example, a CM may read 8 switches and control the intensity of 4 light channels, while another CM may read and emit infrared signals and control 8 relays. The DomoBus approach can thus be an economical

solution as the hardware resources of a CM allow to control multiple devices (up to 20), making it feasible and cost-effective to address *super-automated homes* [51][52].

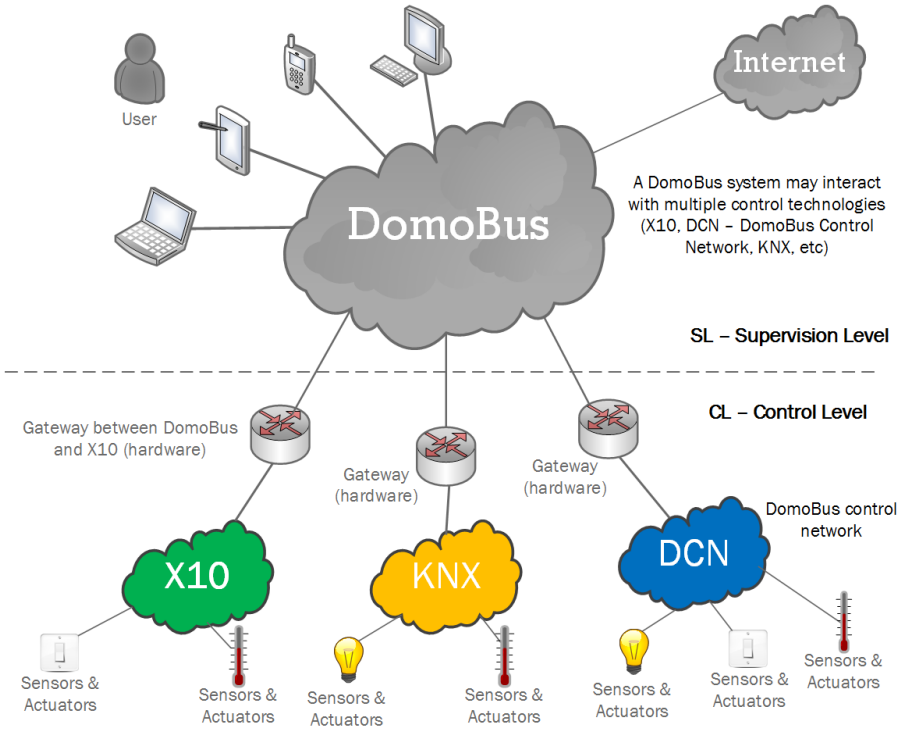


Figure 12: DomoBus general architecture - Control Level.

Each CM can be connected to one or more Supervision Modules (SM) through *gateways* - see Figures 13 and 13. The Supervision Modules (SM) are responsible for the system management and supervision, and they can be implemented with regular computers or even smaller computerized devices such as Raspberry Pis. They receive information from the CMs, process it accordingly to the programmed rules and required behavior, and issue the appropriate commands to the CMs. A system may have as many SMs as needed, and so in small systems there may be just one SM, while on a big and complex system there may be, for example, one SM for each network/house segment. This particularity allows a distributed supervision, offering benefits regarding response time and reliability (no single points of failure). Furthermore, a SM can control any CM in the system and the SMs can interact with each other in order to share information or coordinate actions. Thus, the interaction with other systems becomes easier, allowing interoperation and the achievement of integrated solutions as proposed [51][52].

Finally, the system architecture also considers the existence of a PC - *Home Server* (see Figure 13), that offers a powerful graphical interface for system monitoring and programming, and allows the access to the Internet and other systems if required [51].

The DomoBus architecture offers benefits over other solutions, as a small number of modules can be easily installed on each room of a house, simplifying cabling to sensors and actuators, and making installations more modular.

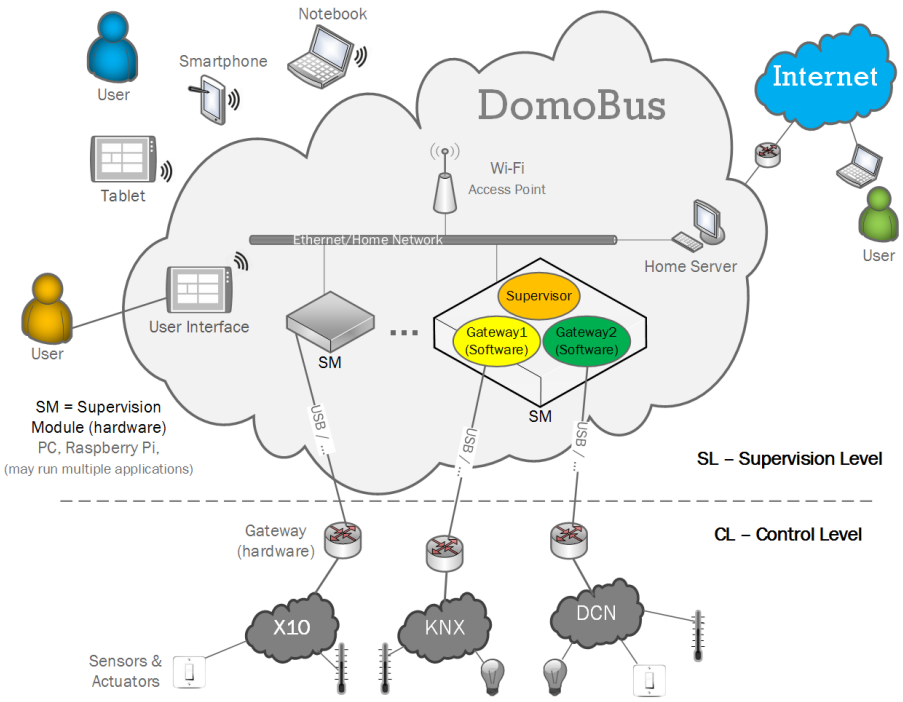


Figure 13: DomoBus general architecture - Supervision Level.

Another big contribution given by DomoBus is its abstraction model specification. This model allows for the abstracted description, organization and manipulation of the automation devices and the home building. It provides a generic way to establish, programatically, the physical structure of the home building and each device present on it, defining each one as a set of properties and services, as well as a way to define scenarios [52][53][54]. This way, the DomoBus system (specially this abstraction model) can be applied to any home automation system.

The first part of the DomoBus abstraction model, present on Figure 14, addresses the home/house description, and it allows to define each house as a group of floors, each one further composed by a group of divisions, and each division can contain a group of devices. This provides a simple, flexible and organized way to represent each specific home structure/environment [52][53].

Figure 14: DomoBus abstraction model for a house's structure and location of devices[53].

The second part of the DomoBus abstraction model, present on Figure 15, addresses the description of domotic devices. It is further divided into two main sections. The first one defines generic device types based on their properties, and the second one refers to each real domotic device existent on a specific automation system, which is associated with a specific device type [52][53].

Each **Device Type** (e.g. *Light Bulb* type), is defined by a set of properties, each one with a specific value. For the *Light Bulb* example type, we can have two properties: On-Off, which states if the light is turned on or off, and *Intensity*, which states the light bulb intensity. Another example can be an *Air*

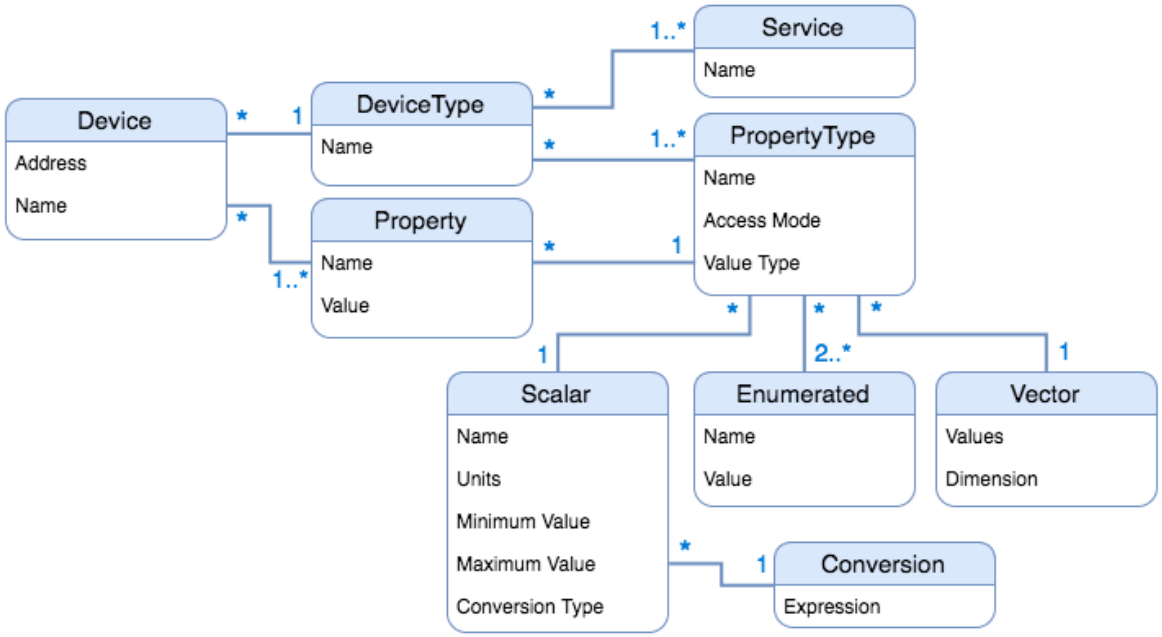


Figure 15: DomoBus device description abstraction model [53].

Conditioner type, where the properties can be *On-Off*, *Temperature*, *State* (Cooling or Heating), and *Ventilation Speed* [52][53][54].

Each **Property** has only one **Property Type**, which is device independent (i.e. can be shared among/used by several devices), and it is used to "describe" each device property. Thus, each **Property Type** defines a name, an access mode (read-only, write-only or read-write), and a *Value Type*, for its correspondent **Property** [52][53][54].

Thus, each **Property Type**, and consequently each device property, has only one *Value Type*, which can be one of three possibilities: **Scalar Value Type**, **Enumerated Value Type** or **Vector Value Type**. The **Scalar Value Type** is used to define scalar values and it states the maximum and minimum possible values, the units for that value type, and an optional conversion type, which can be used, for example, to convert temperature values from Fahrenheit to Celsius degrees if a **Scalar Value Type** refers to temperature. The **Enumerated Value Type** defines a name-value pair for each possible value, and it is used to define possible specific value choices when a property can only have such limited and specific set of values. On the air conditioner example given above, it can be used on the *state* property, because such property can only have a limited set of values - *0 (Cooling)* or *1 (Heating)*. Finally, the **Vector Value Type** allows to define other types of values that cannot be represented by the **Scalar** or **Enumerated** types. An example can be a property named *AlarmCode* where the alarm code must be a number sequence [52][53][54].

Each domotic device existent on the home environment is modulated as a **Device**, and on each **Device** it must be specified/used a **Device Type**. For example, if we have a *Kitchen Light* which we want to integrate on the system, this device's type could be *Light Bulb*, and thus, automatically, its properties and correspondent value types, are defined and set, and the same procedure/reasoning can be applied

to integrate all the lights on the automation system. Furthermore, for each **Device** a **Service** set may be specified, which should state the device's functionality, i.e. for the *Kitchen Light* example, this device functionality is to illuminate, and so it should belong to, for example, the *Lighting Service*. Each **Service** can have several devices, and thus the devices can also be organized by their functionalities/services on the system [52][53][54].

This abstraction model provides a generic and flexible way to represent all the domotic devices independently of each one's structure/technology, because it is applies a reasoning simply based on the description of each device properties, allowing also an easy addition of new devices to the home automation system [52][53][54].

Finally, DomoBus presents also a way to specify the system's behavior, by defining behavioral scenarios (IF...THEN...ELSE formulations), which allows the configuration and adaptation of the system behavior according to the user's need and preference. The scenario's abstraction model, which is presented on Figure 16, allows the user to program the automation system's behaviour [52][53].

Figure 16: DomoBus scenario's abstraction model [53].

On a scenario, for example "Turn on/off all lights", a set of conditions must be verified to trigger the wanted actions. This scenario conditions can be as complex as needed, and they are defined by a set of logical operators, which in turn are done through expressions built from sequences of the "OR" and/or the "AND" operations, providing a flexible way to build the conditions. The list of actions to be performed when the conditions are verified, is a sequence of any number of activation actions, where each one of them performs a value attribution to a specific domotic device property. Deactivation actions are optional and they are executed when one or more conditions are no longer verified. A scenario can be deactivated according to the user's need, and it can be reactivated later. On the example scenario given above ("Turn on/off all lights"), the conditions could be IF the *Intensity* property value of a light sensor is below a certain threshold value, THEN the executed actions will set the value "On" to the On-Off property of all Light Bulb devices present on the automation system, or ELSE the value "Off" is set [52][53][54].

With this effective and simple way to specify an action scenario, it becomes easy to shape the home system's behaviour to the user's need and preference, such as to program pre-defined actions to specific events like parties, special dinners, etc. [52][53].

In order to define and represent all these specifications and abstraction models on a both machine and human readable format, DomoBus uses a XML file to store all the relevant description informations about the automation system, according to the specified abstraction models [54]. By using XML, it is thus possible to expand the system specifications, including more relevant details when needed, or using

only part of it, by simply ignoring some of the specified elements, for example by applications that do not require them to work [52][53][54].

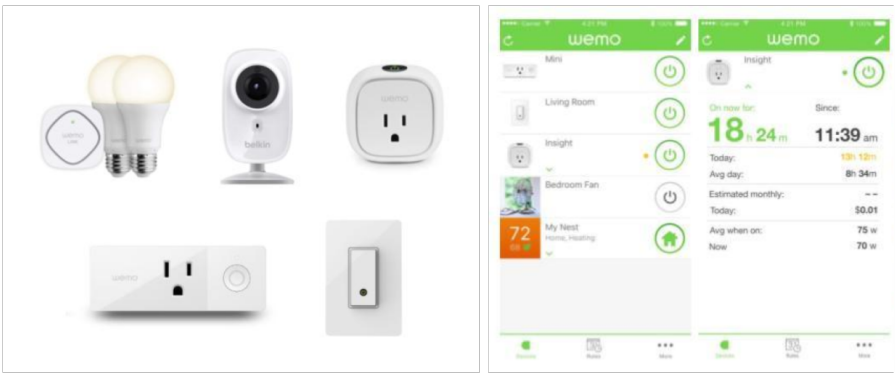
2.2.4 Commercial Solutions

Currently there are many domotic solutions provided by several well known companies on the the domotics market, as well as some others provided by companies new on this domain. Some examples are companies like Belkin, Philips, Samsung, Apple, Google or Amazon.

The solutions provided by these companies can be organized in different sub-areas, because some of them are specialized on the interconnection of the automation devices (some of them focus on the own company's proprietary devices and others provide solutions to work with all kind of devices), others are specialized only on a subset of home devices (only lights for example), and finally others focus on trying to agile and provide better and new ways to the user to control the home devices, for example with voice recognition and command issuing.

Belkin WEMO

Belkin WEMO [55] is the first example of home automation solution which focus on the interconnection of the automation devices. Belkin, known by its products, provides several home devices that work out-of-the-box with WEMO, such as smart plugs, light switches, lights, or smart cameras. With WEMO, users can control their devices from a smartphone application and it also supports integration with Amazon Echo/Alexa or Google Home/Assistant [55]. It enables users to schedule automatic actions to be performed by their devices, to receive notifications about their devices actions and to monitor and take insights about energy usage. WEMO works with WiFi, so a WiFi network must exist on the user's home, and it has a totally distributed architecture, as each device appears on the home WiFi as an isolated device, not recurring to a specific "*server*"/hub. The main disadvantage of this kind of solutions is that WEMO is a totally closed proprietary solution, i.e. it only works with Belkin devices built to work with WEMOS, so users are always restricted on the kind/type of devices and automations they can have on their homes [55].

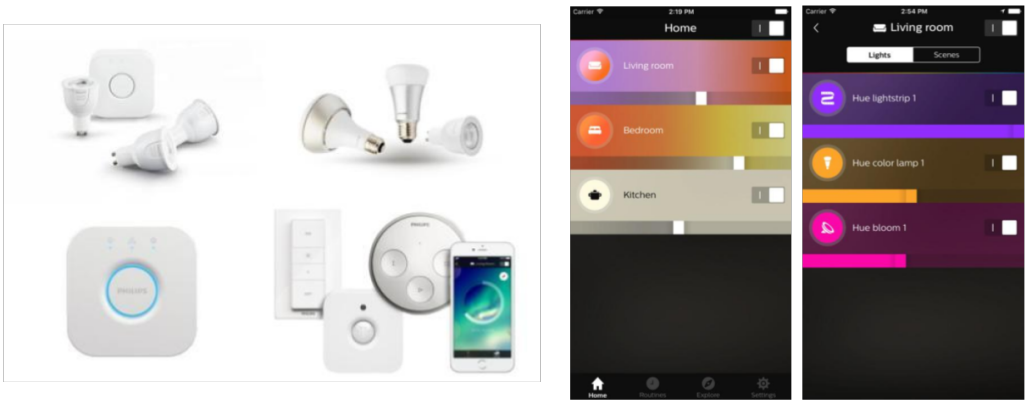


(a) (b)

Figure 17: (a) - Belkin WEMO example devices [55]; (b) - application screens [56].

Philips Hue

Philips presents another home automation solution specialized on the interconnection of the automation devices, but in this case focusing only on lights, as they are one of the best well-known products made by Philips. The provided solution is called Hue [57], and it is composed by a large set of different smart light bulbs produced by Philips, a hub/bridge which is the heart of the whole system, a set of controls such as motion sensors or dimmer switches, a smartphone application, a web portal and a set of user applications that are built by the users and that can control the devices and perform different actions on the system [58]. The Hue system is based on ZigBee and the Hue bridge, via WiFi pairing with a home WiFi router, allows the users to control their devices from anywhere. Hue enables users to control their home light environment by creating rooms, schedule actions, creating scenes, and the smart light bulbs also have many capabilities such as the possibility to present different colors, syncing to music, TV and games, for even more immersive effects [58][59]. It is also compatible with solutions like Amazon Echo/Alexa, Google Home/Assistant and Apple HomeKit. Despite working only with lights, one of the main advantages of the Hue system is that it presents a set of RESTful APIs where the data is exchanged with JSON, that users/developers can easily use to develop new functionalities and applications, and, as it works over ZigBee, modular developments (without the Hue bridge for example) are also possible [59].



(a) (b)

Figure 18: (a) - Philips Hue example devices [58]; (b) - application screens [60].

Samsung SmartThings

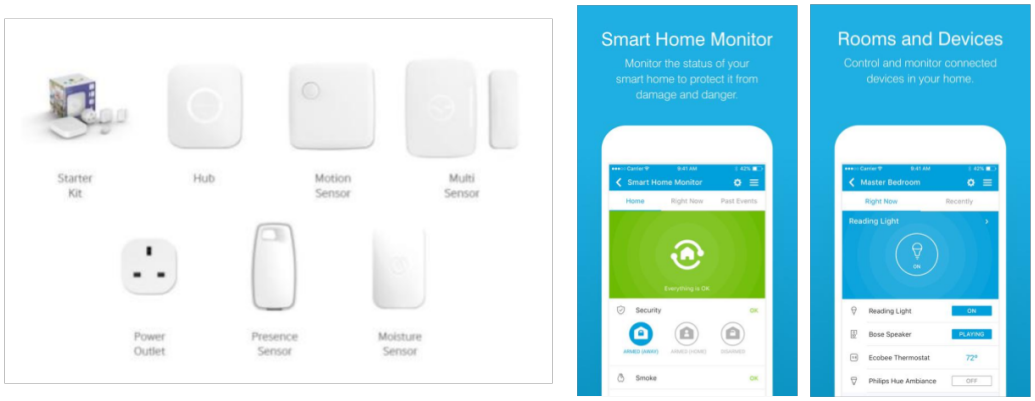
A broader and more general domotic solution is presented by Samsung - the Samsung SmartThings [61]. It is a solution that integrates hundreds of products from dozens of different brands on several areas such as lightning, dimmers, switches, outlets, sensors, door locks, thermostats, etc. With the Samsung SmartThings solution users can: lock and unlock their house doors, control and automate their lights from anywhere, protect valuable items and secure off-limit areas of their homes, get instant alerts if there's unexpected movement or entry in their homes, trigger an alarm to sound if there's unwanted motion or entry, be notified if doors, windows, and cabinets open unexpectedly, among many

others. To do that users must only to acquire the SmartThings Hub and the free SmartThings mobile application, and then they can add as many other popular smart devices as they want to create a fully connected home [61][62].

The heart of the SmartThings system is the SmartThings Hub which is a device that connects with the house WiFi network router via Ethernet and enables the communication between all connected devices, the SmartThings cloud and the SmartThings mobile application. It works with devices supporting the ZigBee and/or Z-Wave protocols [62]. Samsung SmartThings is a totally open system and also provides APIs, SDKs and an IDE so that developers can easily build new applications and domotic services to work with SmartThings. The main advantage of SmartThings comparing with solutions like the Philips Hue, is that it provides an open and simple way/system to access and control a big range of home automation devices from several different brands (for example, it can control the Philips Hue Light Bulbs as well), where the Philips Hue only controls the lighting system [61][62].

The SmartThings mobile application allows users to access and control their devices from their smartphones, tablets or smartwatches, and a cloud portal/application is also provided with the same purpose [61][62].

Finally, the SmartThings system is totally compatible with Android and iPhone systems, Apple Watch, Amazon Echo/Alexa and Google Home/Assistant, making it even easier and convenient for users to access and control their home devices [61][62].



(a) (b)

Figure 19: (a) -Samsung SmartThings example devices; (b) - application screens [63].

Apple HomeKit

Apple HomeKit is a domotic solution provided by Apple which has a working behaviour very similar to the Belkin WEMO and the Samsung SmartThings solutions [64]. It is mainly based on a mobile application and it works with dozens of automation devices from many manufacturers, such as lights, switches, outlets, thermostats, windows, fans, sensors, air conditioners, locks, cameras, etc. Architecturally, each device appears on the application to the user as a isolated device (like on Belkin WEMO) but an architecture where a hub is used to support communications and remote access (like Samsung

SmartThings), is also possible [64][65]. The mobile application enables users to access and control their devices and it comes already built in on the newer iOS versions. It allows users to organize their devices by room, manage multiple device at the same time, control their home with Siri, and more. In order to access the home devices remotely on iOS devices, Apple TV can be used as a home hub device to do things like shut the garage door, view live camera video of the doorway, or even ask Siri to lower the thermostat, from anywhere [64][65]. Although Apple provides a framework to develop devices, applications and services to the HomeKit solution, several disadvantages can be pointed to this solution. The first one is that, where a solution like SmartThings is built always to try to work and be compatible with the majority of the devices from the most well-known devices manufacturers, the Apple HomeKit solution paradigm is that manufacturers must build their devices to be compatible with the Apple HomeKit solution and must be firstly certified by Apple if they want to be supported by Apple HomeKit. A second disadvantage is that it is a very closed solution where a proprietary protocol is used for communication, which does not help the development of new applications, functionalities and services for it. A third disadvantage is that the user controlling application only works with Apple devices (iPhone, iPad, iPod and Apple Watch), and so users that do not own any of Apple's devices cannot use the HomeKit solution. Finally, although it is compatible with voice controlled solutions like Siri, it is not compatible with well-known solutions from other companies like Amazon Echo/Alexa and Google Home/Assistant.

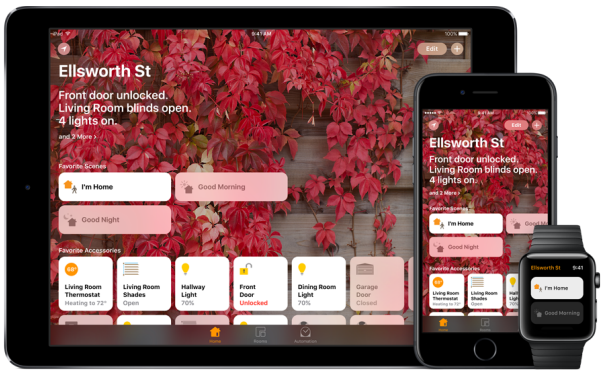


Figure 20: Apple HomeKit example controlling devices [64].

Amazon Echo/Alexa

In order to provide simpler and more convenient ways to users when controlling their devices, Amazon developed a solution that enables users to control their devices with voice. That solution is built upon a voice recognition service, that "lives" on the cloud, called Alexa [66], and a device that users can buy and place in their homes, which connects to the online Alexa service and enables users to use Alexa's capabilities, called Echo [67]. Thus, Echo is a hands-free speaker that users control with their voices. Echo connects to Alexa to play music, make calls, send and receive messages, provide information, news, sports scores, weather, and more, instantly. All users have to do is ask. As Alexa is built in the cloud, it is always getting smarter. The more customers use Alexa, the more she adapts to speech patterns, vocabulary, and personal preferences, and because Echo is always connected, updates are delivered automatically [67][66]. In order to always improve and give better experiences to

the users, a great advantage of this solution is that Amazon provides the Alexa Skills Kit, which is a free collection of self-service APIs, tools, documentation and code samples that make it fast and easy for developers to add skills to Alexa. All code runs on the cloud — nothing is installed on any user device. With the Alexa Skills Kit, developers can create compelling, hands-free voice experiences by adding their own skills to Alexa [66]. Customers can access these new skills on any Alexa-enabled device simply by asking Alexa a question or making a command. Another great advantage of Echo/Alexa is that the majority of the more interesting automation solutions/devices (e.g. Samsung SmartThings) are totally out-of-the-box compatible with Echo/Alexa, and so users can simply use their voice to control such devices, besides the mobile applications built and offered to do that.

(a) (b)

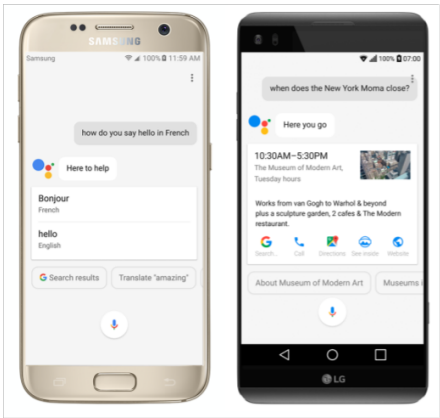
Figure 21: (a) - Amazon Echo device; (b) - Alexa example commands [67].

Google Home/Assistant

Just like Amazon Echo and Alexa, Google also developed its solution that enables users to “talk” with their devices in a simpler and more convenient way. The provided solution is also built upon an online service called Google Assistant [68] and a device that users can buy and place on their homes, which connects to Google Assistant and enables users to use its capabilities, called Google Home [69]. Just like Amazon’s solution, Google Home allows users to perform all sort of control operations with their devices, like play music, make calls, send and receive messages, provide all sort of informations, news, traffic, all in real-time. Also, just like Amazon’s solution, Google provides a set of APIs, SDKs and all relevant documentation so that developers can easily build new applications, functionalities and services to work with Google’s Assistant, always helping to improve its overall “intelligence” and capabilities [68][69]. Besides providing all the advantages also provided by Amazon’s solution, like the out-of-the-box compatibility with the majority of the greatest automation devices/solutions, Google’s Assistant provides a even more immersive experience to the users, because its goal is to work on the largest set of user devices possible [68]. It can work on the user’s smartphone, the user’s Google Home device, smartwatch, and new versions are being built to even work on the user’s smart TVs, cars, etc. Therefore, the Assistant can learn, over time, all the user’s habits, favourite services and actions, among others, on all user de-

vices, providing better helping informations and suggestions [68]. On the other hand, although Google assures security on all interactions with the service, to some users such an immersive experience like the one provided by Google's Assistant, can be a bit scareful, because if the user allows it, it will learn almost everything about the user's life habits, favourite actions, services, etc.

(a)



(b)

Figure 22: (a) - Google Home Device; (b) - Assistant mobile application example screens [69]

3. Solution Implementation

On this chapter the proposed solution is presented. Firstly, some initial considerations are done regarding the approach taken to achieve the final solution, the main requirements, and the main functionalities developed to the framework, taking into account the objectives and goals proposed on section 1.2. Next, the overall architecture and the main components of the framework are presented, followed by the description of the framework's implementation on its prototype. Finally all the technology choices that were made to develop the framework's prototype and to define the framework are properly explained.

3.1 Initial Considerations

The main objective of this thesis was the development of a framework that helps to build domotic systems with an IoT essence. Such framework is purely the definition of a system architecture and a set of interfaces that allow the overall system, and its components, to be interconnected with each other and/or with other systems or services. Moreover, a framework's prototype was also developed to prove the framework's usability and capabilities. Nonetheless, that prototype should not be seen as the main result of this thesis. The main contribution is done through all the framework definitions, which can be applied to almost any domotic/IoT scenario with only minor adjustments, independently of the prototype development that was done using some specific technologies/languages.

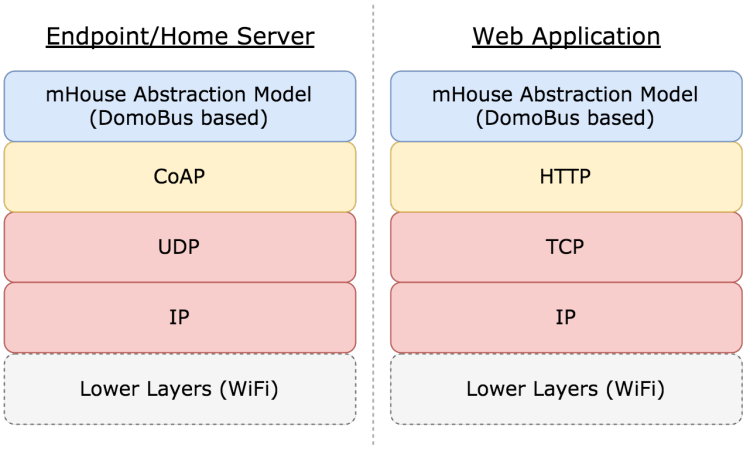


Figure 23: Framework General Technology Stack.

The developed framework is a high-level one, i.e. it only makes restrictions (specifications) from the network layer up. This implies that, theoretically, the framework works correctly independently of the protocol or technology used on the layers below the network layer, as long as such technology can be integrated with the IP protocol. On the network layer, taking into account the considerations made on section 2.1.2.4, the framework uses the IP technology[1] - see Figure 23.

[1] It can work with IPv4 or IPv6, but the developed prototype is working with IPv4.

The framework is also RESTful, i.e. the technologies used, the developed interfaces and the overall system architecture, all follow the REST paradigm. This choice was done taking into account the analysis made on section 2.1.2.3. Therefore, on the application and transport layers, taking into account the study performed on sections 2.1.2.4 and 2.2.2, the framework uses the CoAP and the UDP protocols for communication on more constrained environments, such as on home networks, and the HTTP and TCP protocols for communication on non-constrained environments, such as on the Web/cloud - see Figure 23. This choices state a big difference from the solutions presented on section 2.1.3, as they usually rely on the MQTT protocol, with a publish-subscribe communication pattern, and so they can not be considered 100% RESTful.

The framework system architecture is inspired on the system defined by the DomoBus solution and, to abstract all the framework's concepts, an abstraction model was also developed based on the DomoBus one - see Figure 23. In order to provide a solution with an IoT essence, the framework targets mainly *super-automated homes*, where basically every "thing"/object that belongs to or is in the home environment can be considered to be interconnected.

Finally, to ease references to the developed framework, a name was given to it. It is called "mHouse Framework", where mHouse is spelled "mouse" and it means "my house".

3.2 Main Requirements and Functionalities

After performing the background study for this thesis (chapter 2), and also taking into account the objectives stated on section 1.2, some main requirements and functionalities were defined to be essential on the framework's development.

First, despite all the characteristics and requirements from both the IoT and the Domotic fields, which are important and, of course, were taken into account, the development was focused on providing a way to build Domotic systems capable of being accessed and controlled from everywhere and at anytime, and that have four main characteristics: **openess**, **simplicity**, **expandability**, and **generality**.

Therefore, the framework should be an **open** solution, in order to allow anyone to use it in the most convenient and "free" way. The framework should also be **simple** to use and to understand, to ease its usage on developing big and complex Domotics and/or IoT systems. It should be **expandable** as well, to allow adding and/or removing parts or components to or from it in a simple and easy way. And finally it should also be completely **generic** in order to abstract and be capable of integrate many technologies and devices in a generic way.

These four characteristics can be seen as the four pillars of all the developed work, and they were firstly targeted to allow, on a second step, to pursuit other wanted characteristics such as interoperability, heterogeneity, security, etc., which are also relevant on this thesis context.

Now, the specific functionalities that the user should be capable to do when using the framework are:

- To access, control and manage all the system resources (devices, houses, etc.) from anywhere and at anytime, through generic and simple interfaces, not only by the end user but also by other systems, services and/or applications.

- To store all the collected data and actions on histories, to allow their posterior analysis taking insights and possibly issuing reactions based on that analysis.

- To always detect device failures and/or misbehaviour, allowing to build alarmistic services that can alert the user about those failures or on emergency situations.

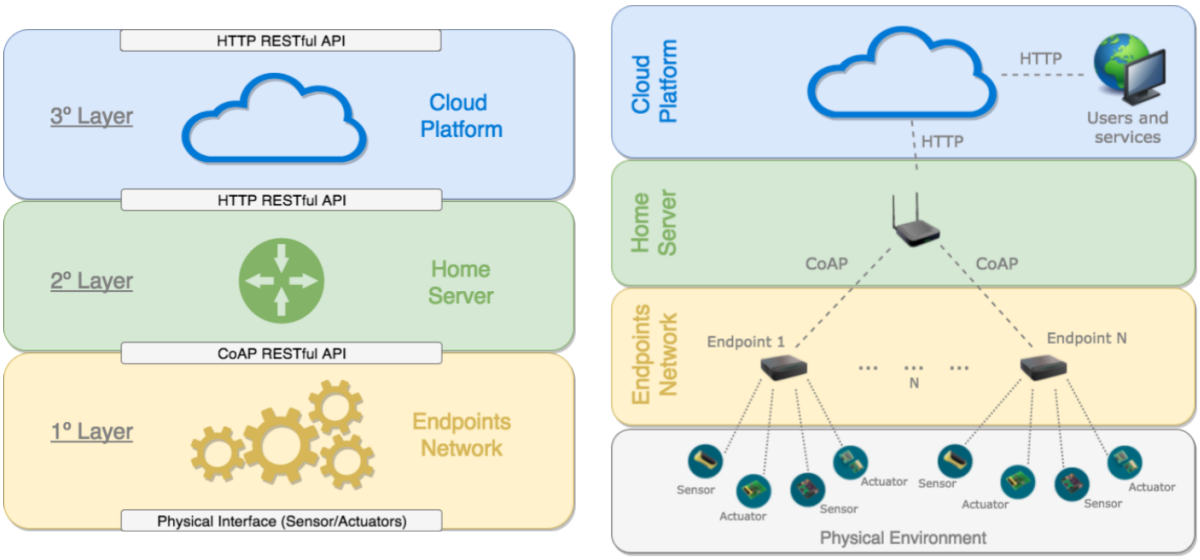
- To organize all the devices by their location on the houses and/or by their functionality, to help the user on their management and to allow issuing commands for groups of devices, e.g. turn off all lights.

- To integrate a huge diversity of devices, with the possibility that each one can be based on different technologies and can have different sensors and/or actuators attached to it.

- To use the framework's system as a "whole" or just parts of it, integrating other systems or services on it, or integrating it with others. This two last functionalities are very important in order to give an IoT essence to the Domotic system.

Finally it is specially relevant to note that, although security is a fundamental requirement of any technological system, specially on an IoT and/or Domotic context, the developed prototype was not built focusing it, and only some basic security mechanisms were used on the prototype development. Thus, the framework prototype can not be seen/used straight forward as a totally secure one, but given its open, expandable and generic essence, security mechanisms or components can be easily added to it, in order to provide a stronger solution.

3.3 System's Architecture



(a) (b)

Figure 24: (a) - mHouse Framework General Three Abstraction Layered Architecture;
(b) - mHouse Framework General Physical Architecture.

As presented on Figure 24, the mHouse Framework has a three layered architecture with three main components: the Cloud Platform, the Home Server and the Endpoints Network. Each one of these components is crucial in order to use all framework's functionalities, because each one of them has a specific purpose and generally represents an abstraction layer.

Cloud Platform

The Cloud Platform represents the top layer which is the one with more abstraction regarding the physical/low level details. It takes the form of a cloud-based web application, allowing the whole system to be accessed by the user from anywhere and at anytime, with top-level friendly interfaces, such as an web responsive graphical interface, accessible from any common web browser, and a RESTful API, accessible by any device/machine with the capability to make HTTP requests. This platform abstracts all the details of the remaining system such as devices and spaces modeling. It also allows the user to see and store all the data collected by the devices, and to send commands for each one of those devices. Another advantage and goal of this platform is to provide a place where every action (data change) is stored in an action history. This data can be accessed and analyzed later by other services, which can further provide insights, conclusions and statistics about the collected data. Finally, the Cloud Platform allows to always monitor all the connected devices over time, even when the user is not using (accessing) the Cloud Platform, thus supporting the development of alarmistic applications or services, which can notify the user in case of device failure or emergency situations.

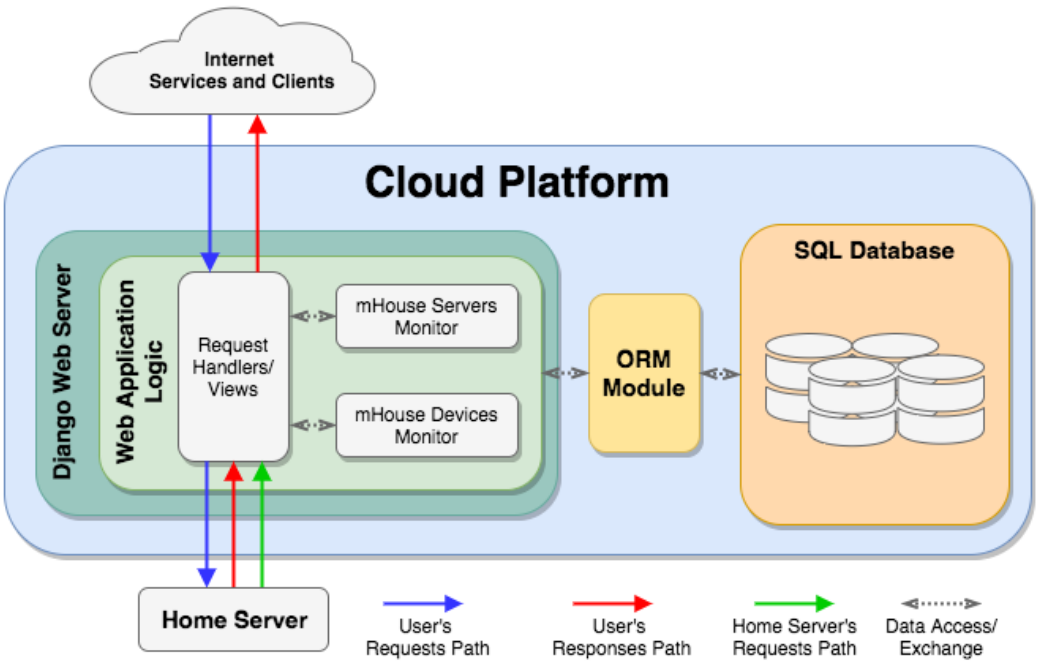


Figure 25: mHouse Framework Cloud Platform Architecture.

As presented on Figure 25, the Cloud Platform component is composed by a web server, which holds all the application logic, an Object-Relational Mapping (ORM) Module that provides an Object-like

interface to a database which, in turn, is the final subcomponent of the Cloud Platform and it is where the overall framework's data is stored.

The web server subcomponent is constituted by a module that handles all incoming requests, and also by two modules that monitor, respectively, all the Home Servers and all **Devices** registered on the Cloud Platform over time.

The database subcomponent stores all the relevant information about the mHouse Framework and it is always synchronized with all the registered Home Servers. It stores all the data representations for each **User**,**House**, **Area**, **Division**, Home **Server**, **Device**, Configuration, **Service**, User Action and Device State Update/Change, that is created/registered on the Cloud Platform. All this information is modulated as explained on section 3.4 to allow its correct storage and management on a common relational SQL database.

Home Server

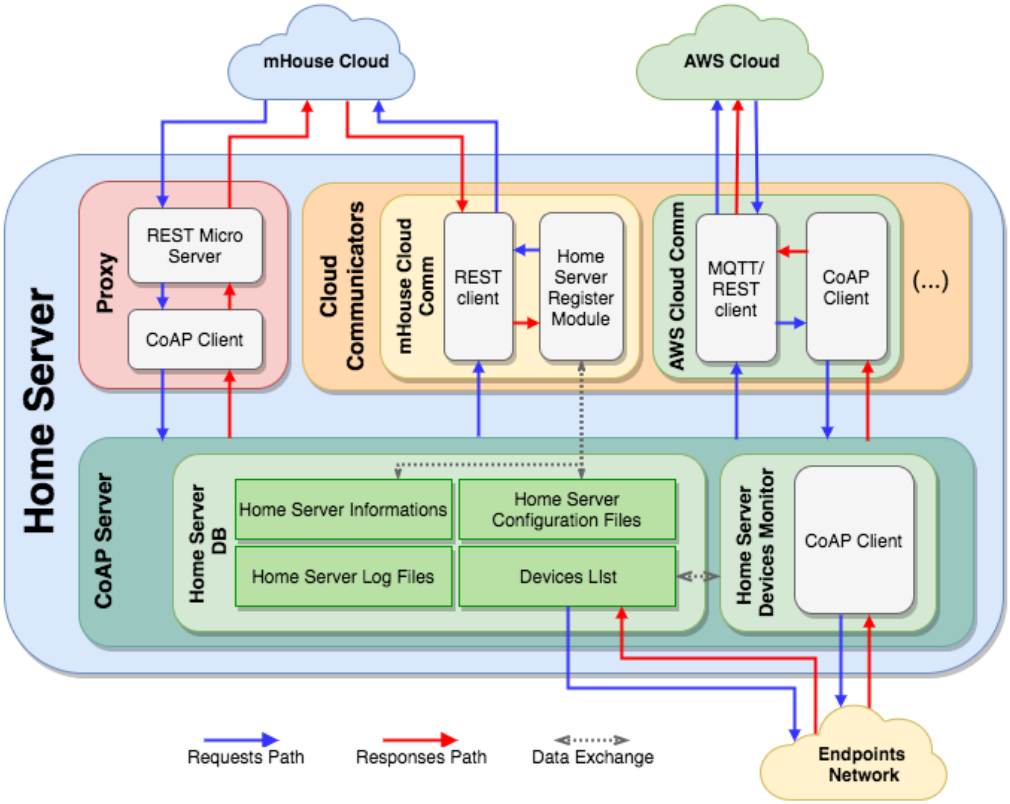


Figure 26: mHouse Framework Home Server Architecture.

The Home Server represents the middle abstraction layer and it acts as a middleware between the Cloud Platform and each one of the connected devices. It takes the form of a software application which runs on a user machine, placed inside his/her house, and it is directly connected to the home internet network. This Home Server is where all devices are represented as resources, making it crucial to the proper work of the framework as a full system. It allows communications on both directions, from the devices to the Cloud Platform and from the Cloud Platform to the devices, by respectively using

the RESTful API provided by the Cloud Platform and providing its own RESTful APIs as well, namely a HTTP based one and a CoAP based one - see Figure 24. Furthermore, this component can also be seen as a “proxy” or “gateway” between the the top and the bottom layers, because without it the communication between those components is not possible, as each device only communicates directly with the Home Server, and never directly with the Cloud Platform.

As presented on Figure 26, the Home Server component is constituted by three main subcomponents: the Proxy, the Cloud Communicators and the CoAP Server.

The Proxy subcomponent is a ”translator” component, composed by a HTTP micro server which exposes, in the top end, a HTTP/RESTful interface, and by a CoAP client that, on the other end, forwards all the requests to the CoAP Server subcomponent. For every incoming HTTP request, the Proxy has the functions to “translate” the relevant request information to the CoAP protocol, maintaining all the request semantics, and to forward that ”translated” request to the CoAP Server, already as a pure CoAP request. During this translation, the Proxy also tries to validate the most part of the incoming requests, so that bad formatted requests never reach and overload the CoAP Server subcomponent.

The Cloud Communicators subcomponent is an easily expandable subcomponent, composed of several “communicators”. A “communicator” is a client that has its specific protocols and behaviour, and that allows the Home Server to interact with a specific Cloud Platform. When working online (with some kind of Cloud support), the Home Server uses these ”communicators” to send device registration and state update requests to each one of the Cloud Platforms integrated with it. By default, the Cloud Communicators subcomponent has a ”communicator” to integrate the Home Server with the mHouse Cloud Platform. By being expandable, other ”communicators” can be easily added to the Cloud Communicators subcomponent. This way, all the Home Server capabilities can be easily integrated with other Cloud Platforms, abstracting that each one of those platforms can have different service characteristics, requirements and APIs.

The CoAP Server is the last subcomponent of the Home Server and it can receive and handle all CoAP requests. It is composed by two main modules: a database, where all the Home Server information, configuration and resources are stored, and a devices monitoring module, which is used to monitor every device connected to the Home Server. The CoAP Server exposes a CoAP interface that is used by the Proxy subcomponent and also by all Endpoints connected to the Home Server through the Endpoints network. The CoAP Server can be seen as the main subcomponent of the Home Server, because it is were all the relevant Home Server informations and resources are stored, and where all the Home Server’s logic is implemented. The devices monitoring module present on the CoAP Server works similarly to the monitoring modules present on the mHouse Cloud Platform. It provides a simple way to always detect device failures, even when there is no internet connection on the Home Server.

Endpoints Network and Application

The Endpoints Network is the network between the Home Server and all the Endpoints, as presented no Figure 24. This network has a star topology and a client-server architecture, where each

Endpoint is a client of the Home Server component. On the mHouse Framework's context, an Endpoint is a low-level microcontroller/embedded software application, which can run on each device present on the home environment. Those devices can have sensors and actuators attached to them, to actually sense and act upon the physical environment. The Endpoint application works, on each on of them, as a "module" that provides the correct behaviour and communication pattern with the remaining Framework components, being, at the same time, totally configurable to handle every specific sensor and/or actuator. In order to establish network connections and communicate with the Home Server, the Endpoints use the CoAP based RESTful API provided by the Home Server - see Figure 24. Thus, this last component represents the low-level/bottom abstraction layer of the mHouse Framework, because it addresses all the details and characteristics of different sensors, actuators, and physical environments.

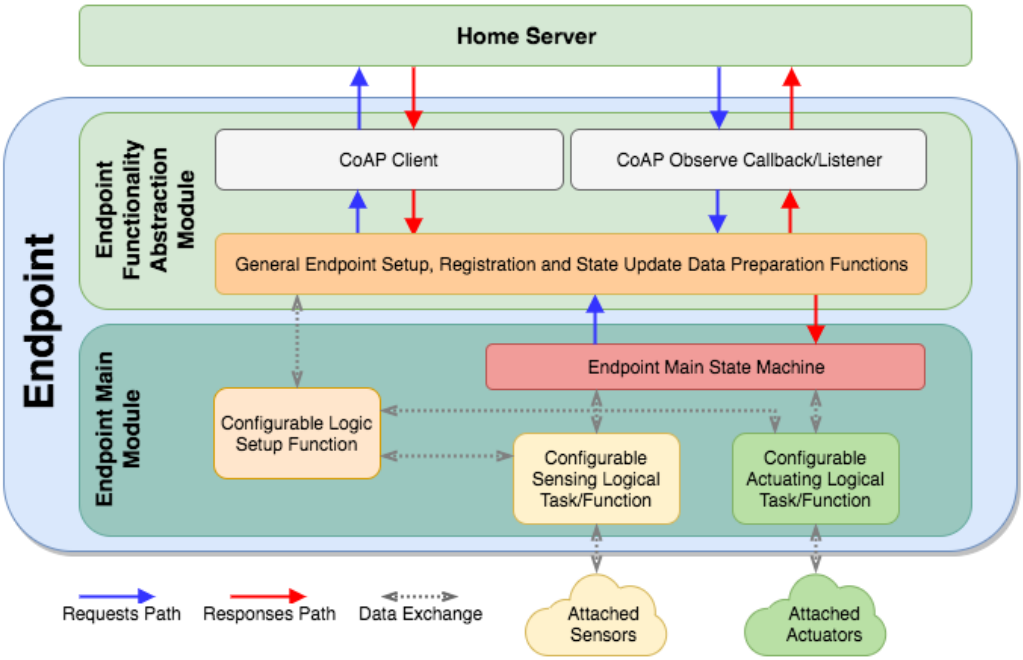


Figure 27: mHouse Framework Endpoint Application Architecture.

The main architecture of the Endpoint application is presented on Figure 27. It is generally composed by a Functionality Abstraction Module, that abstracts all the interaction details and requirements with the remaining framework's components, and a Main Module, that can be configured to work on every specific device with its specific sensors and actuators.

The Endpoint Functionality Abstraction Module is composed by a simple CoAP client (used to communicate with the Home Server's CoAP Server), and by a set of abstraction functions that abstract some behavioral functionality details, e.g. the proper preparation of each request to register the Endpoint or to update the Endpoint's state on the Home Server.

The Endpoint Main Module is composed by an *Endpoint State Machine* and a set of fully configurable settings and functions. The *Endpoint State Machine*, runs in loop infinitely, and ensures the correct behaviour of the Endpoint application according to what is expected by the remaining framework components.

On the set of configurable functions, firstly there is the *Configurable Logic Setup Function*, which is where the user should initialize and setup all the sensors and/or actuators, or whatever he/she needs to run only once at the beginning of the Endpoint Application. Secondly there is the *Configurable Sensing Logical Function*, which is where the user can implement all the sensors logic and can link the sensors outputs to their correspondent mHouse **Device** properties, which are then used on the Endpoint state update requests that are sent to the Home Server. Thirdly there is the *Configurable Actuating Logical Function*, which is where the user can implement all the actuators logic and can link the actuators inputs and/or outputs to their correspondent mHouse *Device* properties, which are then used on the Endpoint state update requests that are received from the Home Server.

Note, once again, that the greatest contribute given by the mHouse Framework is all the APIs and application/component behavioural and architectural specifications, and not only the example prototype software that was developed in the course of this thesis, i.e. the software/solution developed can be used working as it is but, more important, is that the whole mindset and the specifications provided can be easily applied to other technologies, languages or scenarios, with only minor adjustments.

3.4 Abstraction Model

To efficiently handle and manage all the different framework's entities and concepts, an abstraction model was developed to the mHouse Framework. This model is based on the DomoBus one, introducing only a few adjustments to simplify the combination of all data entities with the remaining mHouse system concepts.

The mHouse Abstraction Model is presented on Figure 28. At first sight, notice that its overall structure is mainly the combination of some parts of the DomoBus Abstraction model. It joins the "spaces modeling" part (Houses, Floors/Areas, Divisions) with the "devices modeling" part (Devices, Device Types, Properties, Property Types, Services, etc.).

The model's "root" (main) entity, from which all the others derive, is the **User**. Every other entity belongs to a specific **User**[2]. A **User**, like all the other modeled entities, is always identified by a universally unique "*ID*" on the framework's system. It also has a unique "*email address*" and a "*password*", which are the credentials that a specific user must provide to login on the mHouse Cloud Platform. This **User** concept is only slightly present on the DomoBus System Specification, and it doesn't appear on its abstraction model schemes (Figure 14 and 15), so this can be seen as a first difference between the mHouse Abstraction Model and the DomoBus one.

Each **User** has a set of **Houses** and a set of **Servers**. Each **House** can have only one **Server** and a **Server** can only belong to one **House**. Along with its universally unique "*ID*", each **House** also has a unique "*name*" on the **User**'s account, which helps him/her to easily identify that **House**.

[2]With the exception of the **Core Configurations** entities that are already present on the mHouse Framework by default, and do not belong to any specific user.

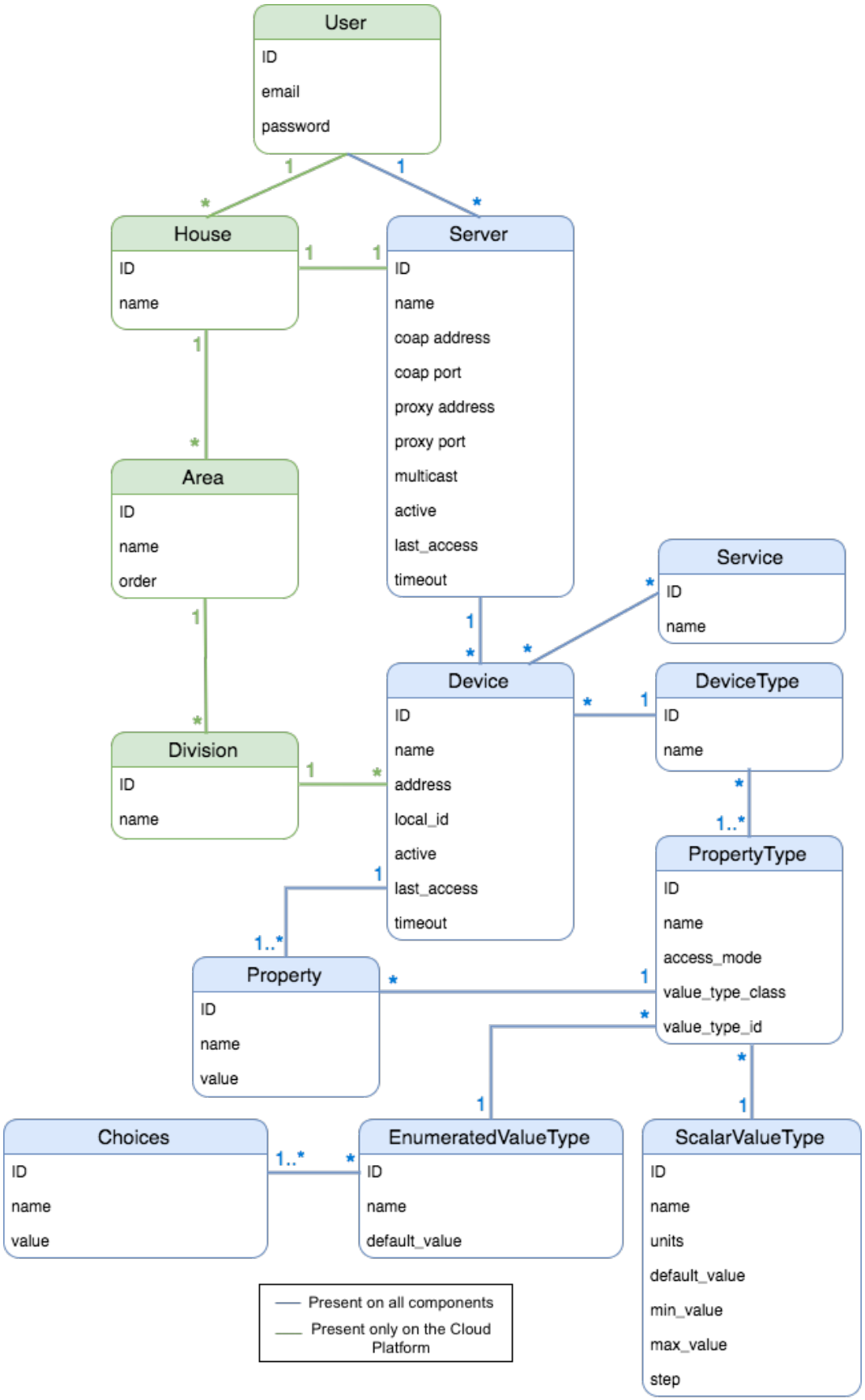


Figure 28: mHouse Framework Abstraction Model.

In the DomoBus Abstraction Model each **House** could have a set of **Floors**. Here, in the mHouse Abstraction Model, this relationship was slightly changed so that each **House** can have a set of **Areas**, being this the second difference between the mHouse and the DomoBus models. The difference between the **Area** and **Floor** concepts is just their essential semantics/connotation. With the **Floor** concept, by essence, the user should divide his/her house as a set of house floors, and when the house has only one floor, this division remains unuseful or could bring some confusion to the user when dividing his/her house. On the other hand, the **Area** concept is a much broader one, because it represents, by essence, a generic area of the house. Thus, each **House** can be subdivided on several **Areas**, and the **User** can have, for example, the Bedrooms area, the Garden area, the Hall area, etc. Notice that, if the user really wants to divide its house by its floors (like just when using the **Floor** concept), each **Floor** can also generally be seen as an **Area**, so that division is also possible and never brings confusion to the user.

Despite this "naming"/connotation difference between **Floors** and **Areas**, just like the DomoBus **Floor** concept, a mHouse **Area** has also a "*name*", which is unique for each user's **House**, and an "*order*" number. On the DomoBus Abstraction Model, the "*order*" number was used to directly indicate the **Floor**'s number (0 for the ground floor, 1 for the first floor, and so on). Given the introduced difference, now, on the mHouse Abstraction Model, the **Area**'s "*order*" number represents some kind of "order" that the **User** can given to his/her **Areas**. This "*order*" number can be used to group **Areas**, to order them in some wished way, or even to represent a floor number in the case where each **Area** corresponds directly to a house floor.

Next, each **Area** can be further divided into a set of **Divisions**. Each **Division** has a "*name*" (unique for each **Area**) and can have a set of **Devices**.

Until now it was detailed the first part of the mHouse Abstraction Model, which is directly correspondent to the first part of the DomoBus Abstraction Model, where the spatial structure (division of the user's house) is modulated. The concepts modulated on this part of the mHouse Abstraction Model (green part of Figure 28), are only present and managed on the higher abstraction layer of the mHouse Framework - the Cloud Platform.

The second part of the Abstraction Model modulates the Home **Servers**, **Devices**, **Services** and "Configurations" concepts (i.e. **Device Types**, **Property Types**, **Properties** and **Property Value Types**). These concepts are present across all the abstraction layers of the mHouse System - from the Cloud Platform to the Endpoints Network (blue part of Figure 28).

The **Server** entity generally represents a Home Server. Each **Server** has:

- A "name".
- A "CoAP IP address" (unique on the **User**'s account context) and "CoAP port" - which are the local home network/private IPv4 address and port of that specific **Server**, and they are used by the Endpoints to connect to that **Server**.
- A "Proxy IP address" and "Proxy port" - which are the public IPv4 address and port of that specific **Server**, and are used by the Cloud Platform to connect to that **Server**.
- A "multicast option" - which states if the **Server** is using multicast addresses to communicate.

- An “active“ flag - states if the **Server** is currently running.
- A “last access” value - which is a timestamp record for the last access/request made by the **Server** on the Cloud Platform.
- A “timeout” value - which states the timeout interval, in seconds, that must be used when monitoring the **Server** from the Cloud Platform.

Besides that, each **Server** can belong to a **House** and can also have a set of **Devices**.

The **Device** entity represents each Endpoint/device connected to the mHouse system. Each **Device** must belong/”be connected” to a **Server**, and it has:

- A “name”.
- An “address” - which is the IPv4 address used by the **Server** to interact with that **Device**.
- A “local ID” - which is the ID provided by the **Server** to that **Device**, and it is unique on the Endpoints Network that is supported by that **Server**[3].
- An “*active*“ flag - states if the **Device** is currently running or not.
- A “*last access*” value - which is a timestamp record for the last access/request made by the **Device** on the Cloud Platform.
- A “*timeout*” value - which states the timeout interval, in seconds, that must be used when monitoring the **Device**.

Besides that, each **Device** can belong to a **Division**, can have a set of **Services**, and it has always its specific “Configurations” - a **Device Type** and a set of **Properties**.

The mHouse “Configurations” concepts are inherited from the DomoBus Abstraction Model, with the exception of the **Vector Value Type**, that is dropped here because it brings no advantages to the mHouse Framework, as all the values represented by that value type can easily be represented with one of the other value types. All the other inherited entities have the same meaning as in the DomoBus Abstraction Model.

Therefore, each **Device Type** has a “*name*” and a set of **Property Types**. Like on the DomoBus Abstration Model, each **Property Type** has a “*name*” and an “*access mode*”, and now it also has a “*value type ID*” and a *”value type class*”. These “*value type ID*” and ”*value type class*” have the same goal (meaning) of the “*value type*” option present on the DomoBus Abstraction Model, which is to identify the specific **Scalar Value Type** or **Enumerated Value Type** on which the **Property Type** is based. The “*value type class*” option can be ”SCALAR” (if the **Property Type** is a **Scalar Value Type**) or ”ENUM” (if the **Property Type** is an **Enumerated Value Type**), and the “*value type ID*” option uniquely identifies the value type being referenced, among the set specified by the “*value type class*” option.

The **Scalar Value Type** generally represents scalar/number values, just like on the DomoBus Abstraction Model. Each **Scalar Value Type** has a “*name*”, a “*unit*” and the “*default*”, “*maximum*”, “*minimum*” and ”*step*” values, which are concepts directly inherited from DomoBus. Here the DomoBus ”conversion” concept is dropped because it brings on added value to the mHouse Framework goals but, if on future work, if anyone needs to have it, it can be easily introduced on the system, working just like it does on the DomoBus system.

[3]The ”*local ID*” differs from the ”*universal ID*” because the first one uniqueness is restricted to the providing **Server** scope, where the second one is unique everywhere on the mHouse system, i.e. it is unique to all the different **Servers** and **Devices** present on the framework.

The **Enumerated Value Type** represents the enumerated values, just like on the DomoBus Abstraction Model. Each **Enumerated Value Type** has a "*name*", a set of **Choices**, and a "*default value*" that must be one of the values present on the specified set of **Choices**. The **Choice** concept is introduced here to represent the possible choices that an **Enumerated Value Type** can have. Each **Choice** has a "*name*" and a "*value*". The "*name*" of a **Choice** is used as the virtual value of that **Choice**, and the "*value*" of the **Choice** is a semantic/physical value that can be specifically and semantically related to the **Choice** itself. For example, consider the **Enumerated Value Type** called "Air Conditioning Commands". A **Choice** set for this **Enumerated Value Type** could be, for example, "On", "Off", "Cooling" and/or "Heating". Now, considering that the option "On" represents the physical value 1, the option "Off" represents the physical value 0, and the "Cooling" and "Heating" options represent the physical values 2 and 3, the **Choices** needed are: choice "On" with the "*value*" 1, choice "Off" with the "*value*" 0, choice "Cooling" with the "*value* 2, and choice "Heating" with the "*value*" 3. Notice that the virtual values/options that the **Enumerated Value Type** can take, and that are used by the mHouse system components, are the ones on the "*name*s" set of those **Choices**, and not the ones on the "*value*s" set. So, for the given example, the "Air Conditioning Commands" **Enumerated Value Type** can have the "*name*s" "On", "Off", "Cooling" and/or "Heating", which have a physical/semantics mapping to the physical "*values*" "1", "0", "2" and/or "3".

Next, each **Device** has a set of **Properties** that are specific for that **Device**, and that is automatically generated from the **Device Type** reference. Thus, each **Property** has a **Property Type**, a "*name*" and "*value*". The **Property**'s "*name*" and "*value*" are automatically set and regulated by the **Property Type** reference, e.g. a **Property** that has a **Property Type** with the name "Humidity" (which, for example, references a **Scalar Value Type** with the name "Percentage", and maximum value of 100, minimum value of 0, step value of 1, "%" as units and default value of 50), will have the "*name*" "Humidity" (inherited from the **Property Type**'s "*name*"), will start with the default value of 50 (inherited from the "Percentage" **Scalar Value Type**'s "*default value*"), and the possible range for its values varies from 0 to 100 with steps of 1 (all these informations also inherited from the "Percentage" **Scalar Value Type** ones).

Finally, each **Device** can belong to a set of **Services**. Each **Service** as a "*name*" and its purpose is to group the **Devices** by functionality, in order to give a different organizational/management perspective to the user. Overall, the **Devices** can be organized and/or grouped by their **Server**, by space (**Division**) or by functionality (**Service**).

3.5 Prototype's Implementation

3.5.1 Cloud Platform Prototype

The Cloud Platform component prototype was developed in Python with the Django Framework. It was deployed to the Elastic Beanstalk AWS PaaS cloud service accessible on `http://mhouseframework.eu-west-1.elasticbeanstalk.com/`. To interact with the Home Server component prototype by sending HTTP requests, the Requests HTTP python library [70] was used on the development.

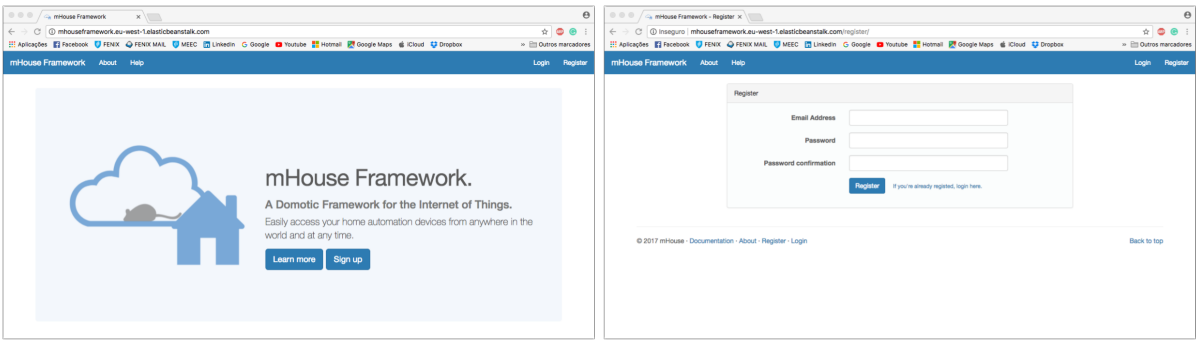
On this prototype, the Cloud Platform monitoring modules, that were explained on section 3.3, are always running in parallel with the remaining Django application, each one in its specific thread. On each one of them, the monitoring function works by periodically comparing the current timestamp with the timestamp of the last access done by a Home Server or by a Device (depending on the monitor module), plus a specific ”timeout”, characteristic of each Home Server and/or Device. Therefore, when a Home Server or a Device does not send any request to the Cloud Platform during a time interval greater than its characteristic “timeout” parameter, these monitoring modules consider that the correspondent Home Server or Device is “down”, and they accordingly change the correspondent state of the Home Server or Device.

Graphical Interface

The developed web server provides a HTML graphical interface, accessible by the regular user from any common web browser, and a RESTful API accessible by any machine with capability to make HTTP requests.

The HTML graphical interface has two main areas: the Public Area and the Personal Area.

The Public Area holds a home page with general informations about the mHouse framework, and a subset of pages where users can login and/or register themselves on the mHouse Cloud Platform. It also provides help links to all mHouse Framework documentation and prototype application source codes - Figure 39 (b).

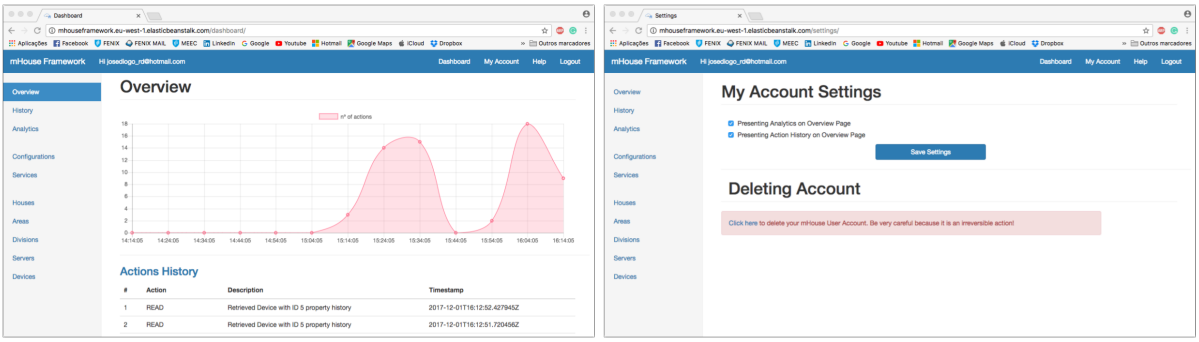


(a) (b)

Figure 29: (a) - Cloud Platform prototype home page. (b) - Cloud Platform prototype ”Register” page.

The Personal Area - Figure 29 - is only accessible by registered/logged in users, which also have access to the Public Area as well. The opposite is not true. Thus, in order to use the Cloud Platform the user must first register on it. To do it he/she can go the register page and provide a valid email and password to complete the registration.

When the user enters the Personal Area, an “Overview” page is presented where the user can see the overall/summary information about any of his/her resources - Figure 30 (a). This page is configurable as the information presented here can be selected on the “My account” page- Figure 30 (b).

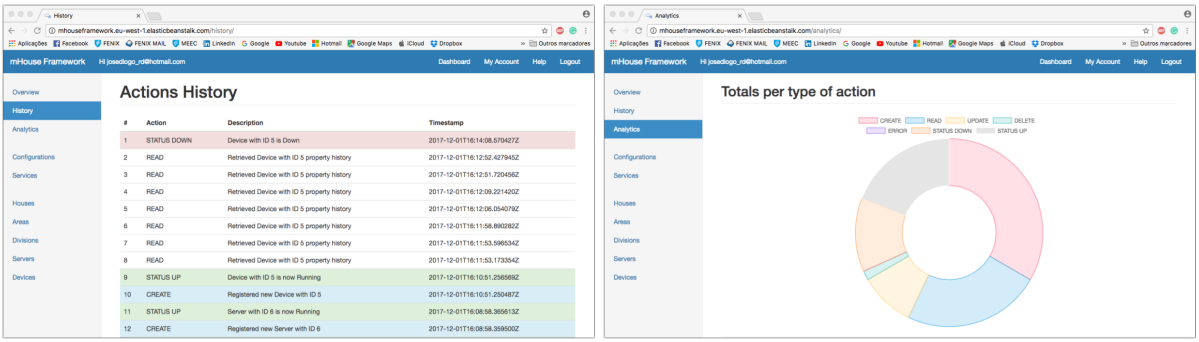


(a) (b)

Figure 30: (a) - Cloud Platform prototype ”Overview” page. (b) - Cloud Platform prototype ”My Account” page.

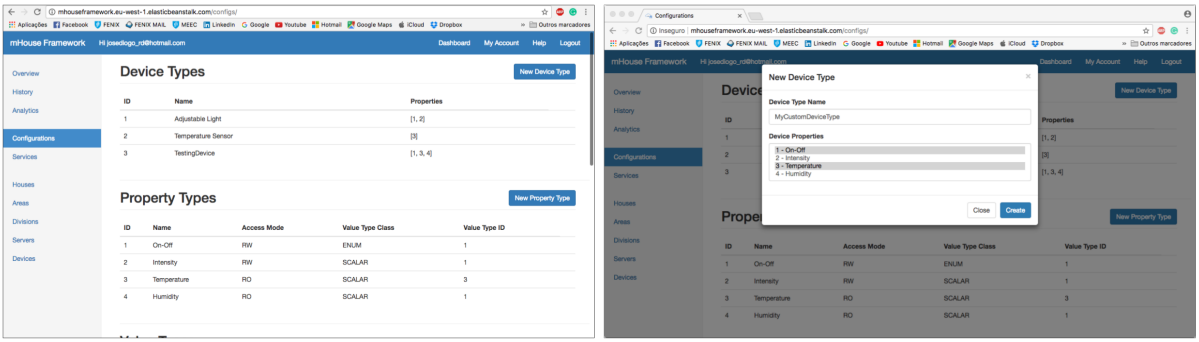
The second page is the “History” page - Figure 31 (a). There the user can see all actions that was and are performed on any of his/her resources over time, since the creation of his/her Cloud Platform account.

The third page of the area is the “Analytics” page, where the user can see some statistics about his/her actions and resources - Figure 31 (b).



(a) (b)

Figure 31: (a) - Cloud Platform prototype ”History” page. (b) - Cloud Platform prototype ”Analytics” page.

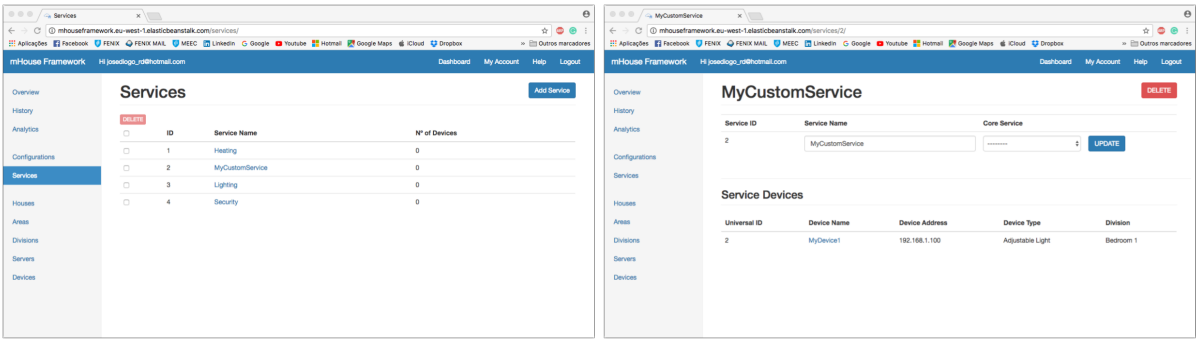


(a) (b)

Figure 32: (a) - Cloud Platform prototype ”Configurations” page. (b) - Adding a Custom Device Type example.

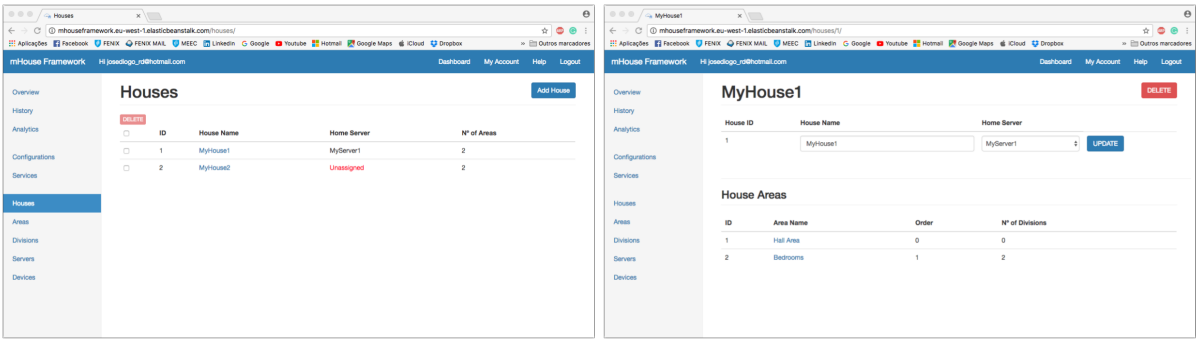
The next page is the “Configurations” page - Figure 32. Here the user can see all the **Core Configurations** and **Services**, and he/she can here create new **Custom Configurations** - Figure 32 (b) -, which will be accessible only by him/her. The purpose of the **Core Configurations** and **Services** is to give examples of possible Configurations, and to save some trouble to the user if a Configuration suitable to his/her needs already exists among the Core ones.

The next page is the “Services” page - Figure 33 (a). Here the user can see his/her **Services**, i.e. the **Services** that will be available on his/her Home **Servers**. On this page the user can also create new **Services** or delete old ones. When the user creates a new **Service** it is asked to provide a name to that **Service**, or he/she can simply choose one of the **Core Services** in which case the new **Service**'s name will be the name of the selected **Core Service**. When the user already has **Services**, he/she can select one and a page with that specific **Service** information will be presented, where the user can also update it - Figure 33 (b). In this **Service** specific page, a list of **Devices** that belong to that **Service** is also presented.



(a) (b)

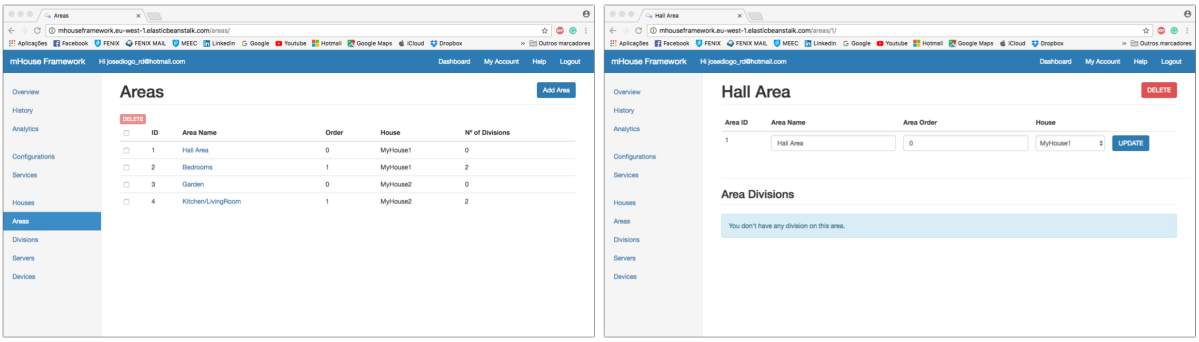
Figure 33: (a) - Cloud Platform prototype ”Services” page. (b) - A specific Service page.



(a) (b)

Figure 34: (a) - Cloud Platform prototype ”Houses” page. (b) - A specific House page.

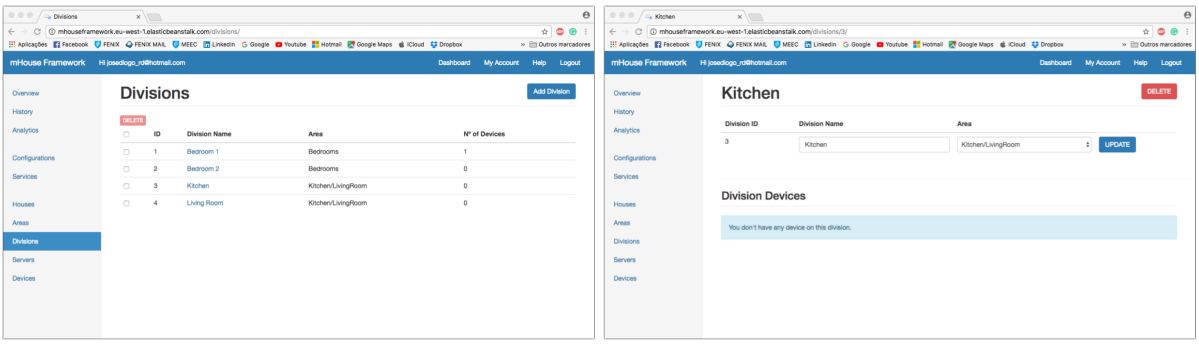
The next page is the “Houses” page - Figure 34 (a). Here the user can see all of his/her **Houses**, create new ones and/or delete them. When creating a new **House** the user must provide all the **House** required informations as explained on section 3.4. When the user has some **House**, he/she can select it and a new page is presented where he/she can see and/or update that **House's** information - Figure 34 (b). In this **House** specific page, a list of **Areas** that belong to that **House** is also presented.



(a) (b)

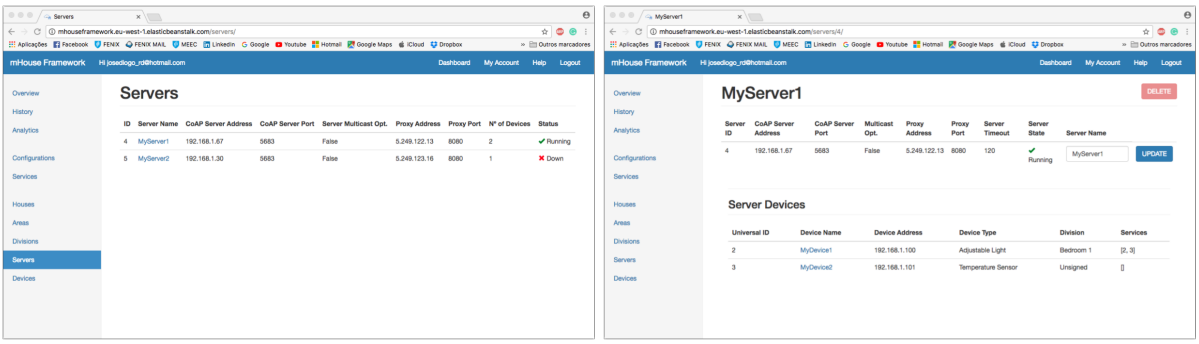
Figure 35: (a) - Cloud Platform prototype ”Areas” page. (b) - A specific Area page.

The next page is the “Areas” page and it is where the user can see all of his/her **Areas**, create new ones and/or delete them - Figure 35 (a). When creating a new **Area** the user must provide all the **Area** required informations as explained on section 3.4. When the user has some **Area**, he/she can select it and a new page is presented where he/she see and/or update the **Area's** information - Figure 35 (b). In this **Area** specific page, a list of **Divisions** that belong to that **Area** is also presented.



(a) (b)

Figure 36: (a) - Cloud Platform prototype ”Divisions” page. (b) - A specific Division page.



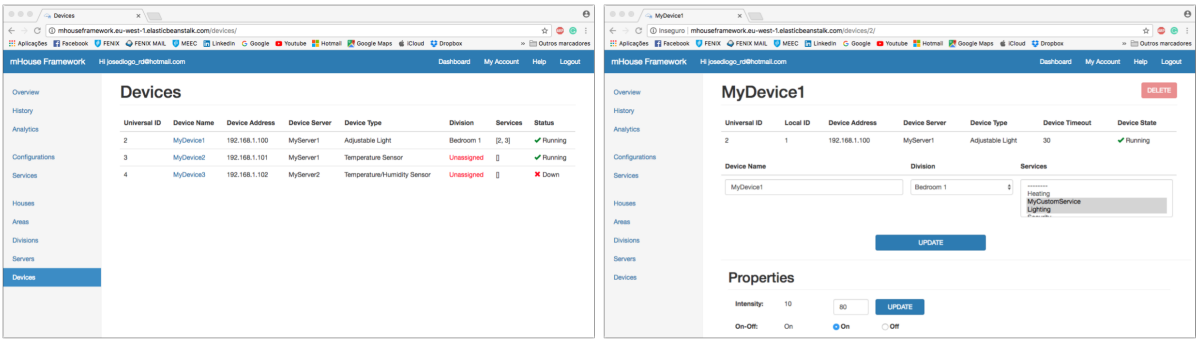
(a) (b)

Figure 37: (a) - Cloud Platform prototype ”Servers” page. (b) - A specific Server page.

The next page is the “Divisions” page - Figure 36 (a). Here the user can see all of his/her **Divisions**, create new ones and/or delete them. When creating a new **Division** the user must provide all the **Division** required informations as explained on section 3.4. When the user have some **Division**,

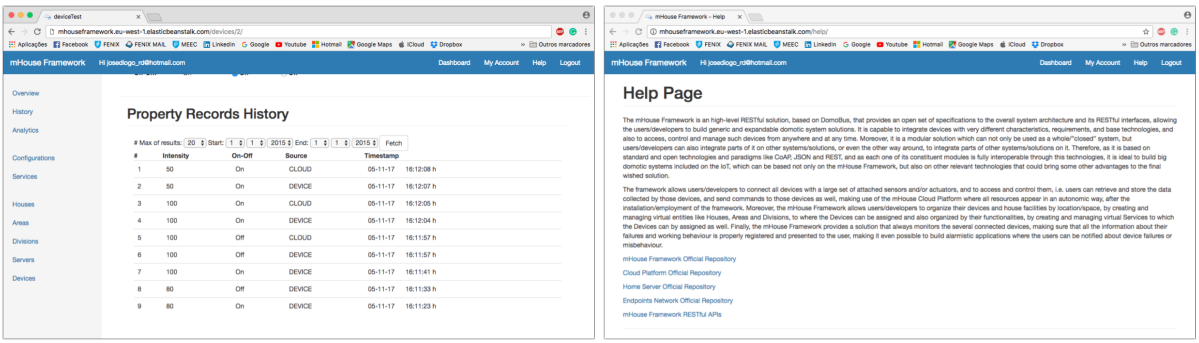
he/she can select it and a new page is presented where he/she can see and/or update the **Division's** information - Figure 36 (b). In this **Division** specific page, a list of **Devices** that belong to that **Division** is also presented.

The next page is the "Servers" page - Figure 37 (a). Here the user can see all of his/her Home **Servers**. The user here cannot create new **Servers** because each **Server** will automatically appear here when the user starts the Home **Server** application on the machine hosting it. When the user has some **Server**, he/she can select it and a new page is presented where he/she can update the **Server's** name and, when the Home **Server** appear to be "down" (stating that the Home **Server** application is not running), the user can also delete that **Server** from the Cloud Platform[4] - Figure 37 (b).



(a) (b)

Figure 38: (a) - Cloud Platform prototype "Devices" page. (b) - A specific Device page - top part.



(a) (b)

Figure 39: (a) - A specific Device page - bottom part. (b) - The public "Help" page.

The next page is the "Devices" page - Figure 38 (a). Here the user can see all of his/her **Devices**. The user cannot create here new **Devices** because each **Device** will automatically appear here when the user starts the Endpoint application on each **Device**. When the user has some **Device**, he/she can select it and a new page is presented where he/she can update some the **Device's** information, like changing the **Device's** name, the **Device's** Division and/or the Service to which it belongs - Figure 38 (b). When the **Device** appear to be "down" (stating that the Endpoint application or the correspondent **Device** are not running), here the user can also delete that **Device** from the Cloud Platform. This page also has a section where graphical setters and readers are presented for each one of the **Device's**

[4]This is only deleting the representation of that **Server** on the Cloud Platform. When the Home **Server** application starts to run again (if it ever starts again), its representation will appear again on the Cloud Platform - check section 3.5.2.

properties - Figure 38 (b). The user can use them to send commands to that **Device** (set new values to the **Device** properties) and/or see the data that is coming from that **Device** (see each **Device** property value at each moment). The last section of this page is the "Property Records History", where a history of all Property values/changes is presented over time - Figure 39 (a).

The last page of the Personal Area is the "My account" (account settings) page, where the user can configure what to see on the "Overview" page, and can delete his/her Cloud Platform user account - Figure 30 (b).

RESTful Interface

The mHouse Cloud Platform RESTful API is a RESTful interface used by the Home **Servers** to interact with the Cloud Platform, and is also available to other entities (machines and services) which want to access the framework's resources from the Internet. All the data exchanged is represented with JSON, so the entity (client) interested to connect itself with the mHouse Cloud Platform only needs to support HTTP requests and JSON parsing, making the API very open and interoperable.

All the resources that can be accessed by the HTML interface can always be accessed with the RESTful API as well, and each resource is accessible by a specific URI that follows the REST specifications. In some special cases, queries can be made to filter the presented results using the query fields of the HTTP requests. Every URI of the RESTful API begins with /**api**/ and the semantics associated to each one is the same as in the HTML interface, e.g. if the user wants to access the list of his/her Houses, with the RESTful API he/she should use an URI like /**api**/**houses**/, which will return a JSON list with the user Houses, just like he/she can do with the HTML interface by accessing the URI /**houses**/ (or the "Houses" page on the Personal Area), where a HTML page is returned with the list of user Houses. To work with the API, an user must be authorized with the Basic Auth authorization process, providing his/her mHouse credentials. On Appendix A the URIs, corresponding semantics/meaning and the requests requirements and possible responses are presented for all the mHouse Framework RESTful APIs.

It is also important to notice that, in order to give friendlier interfaces and better perceived performance to the user, the HTML graphical interface also has an embedded javascript client, which runs on the user's web browser and that interacts with the Cloud Platform through its RESTful API to perform all operations like creation, update or deletion of resources. To read/fetch resources or data from the Cloud Platform, the HTML interface can retrieve everything needed with the request of the HTML page itself or also through the RESTful API used by the javascript client, depending on the specific situation.

3.5.2 Home Server Prototype

The Home Server component prototype was also developed in Python. The HTTP part of the Proxy subcomponent (micro server) was developed with the Bottle framework [71]. All the CoAP-based parts, like the CoAP client on the Proxy subcomponent and the CoAP Server subcomponent, were developed with the CoAPthon library implementation of the CoAP protocol for Python. On the Cloud

Communicators subcomponent, the Requests HTTP python library [70] was used to send all HTTP requests to the Cloud Platform.

The first thing that a user must do in order to work with the Home Server prototype is to register himself/herself on the mHouse Cloud Platform. After that he/she should install the Home Server application on the desired machine, by cloning the online Home Server Application code repository. The machine running the Home Server should have some minimum processing and memory capabilities, because the Home Server is a complex application, which runs with dozens of threads and should support requests from hundreds of Endpoints/devices.

After installing, the Home Server can be started using the "*main.py*" file placed on the root directory of the Home Server prototype application. When the Home Server starts, it tries to register itself on the mHouse Cloud Platform by using its "register module" - Figure 26. If the Home Server is starting for the first time ever on that machine, it will ask the user for a name to give to the Home Server, and for his/her mHouse Cloud Platform credentials[5]. After that, the Home Server will save a configuration file on the "*configs*" directory with all the crucial registration informations. In the subsequent restarts, the Home Server will use that file to automatically complete its registration/synchronization with the Cloud Platform, without asking the user for all the required information.

After this Home Server registration/synchronization process, the Endpoints/devices can register themselves on the Home Server. Every time a new Endpoint/device is registered, the Home Server will send a device registration request to the Cloud Platform, making use of the HTTP client[6] present on the mHouse "Cloud Communicators" subcomponent - see Figure 26.

After being registered on the Home Server, every time a device is considered to be "down" by the Devices monitoring module, it is deleted from the Home Server database. Along with this deletion, a device state update request is sent to the Cloud Platform, stating that the correspondent device is "down".

The Devices monitoring module present on the "CoAP Server" is a thread that is always running in parallel with the "CoAP Server" subcomponent. It works similarly to the monitoring modules present on the mHouse Cloud Platform, by periodically comparing the timestamps differences with the devices characteristic "timeouts". Therefore, when an device does not send any request during a time interval equal to or less than its characteristic "timeout" value, this Devices monitoring module sends a GET request to that specific device (which works like a ping request) to see if that device is still running. If the monitoring module does not receive any response to that GET request, it considers that device to be "down", and thus it will erase that device's representation from the CoAP server database. This mechanism provides a simple way to always detect device failures by the Home Server, which is also important in order to provide data consistency, i.e. not having device representations on the Home Server for devices that are no longer running.

Every time an Endpoint sends a "device state update request", the Home Server updates that correspondent Endpoint representation locally, and it redirects that request to the mHouse Cloud Platform. For that, the Home Server uses the "communicators" present on the Cloud Communicators subcompo-

[5]The email and password with which the user previously registered himself/herself in the Cloud Platform.
[6]Which further uses the Cloud Platform RESTful API.

nent. The ”communicators” integrated with the Home Server can be seen on the “*cloudcommunicators*” directory. On this directory, each ”communicator” has its own code file and the several ”communicators” are linked through the configurable file called ”*main.py*”. This file is the bridge point between all the ”communicators” and the remaining Home Server’s subcomponents code, by containing functions that are called every time a request should be sent to the cloud platforms. New ”communicators” can be easily added to the Cloud Communicators subcomponent by adding new ”communicator” code files to this directory, and then linking the new coded functions on the ”main.py” file.

The default ”communicator” used to work with the mHouse Cloud Platform is crucial for the Home Server online integration. Firstly it allows to send data, e.g. Device state update requests, from the Home Server to the mHouse Cloud Platform. Secondly it has the “register module” (coded on the ”register.py” file), which is the module used by the Home Server to perform its registration/synchronization with the mHouse Cloud Platform. Finally, this ”communicator” also has the special function to periodically send an “alive” message to the mHouse Cloud Platform, stating that the Home Server is correctly running.

During the tests phase, a ”communicator” was also developed to integrate the Home Server prototype with the AWS IoT Platform, and it can be configured on the Home Server code to also send the Devices data to the AWS IoT Platform - check chapter 4 for more information.

All the Home Server ”overall settings”, e.g. its ”timeout” values, its logging files names, etc., can all be seen and edited on the ”*settings.py*” file, which is placed on the Home Server’s root directory.

RESTful Interfaces

The Home Server, as explained on section 3.3, provides two RESTful APIs: a HTTP-based one and a CoAP-based one. With those APIs a set of resources can be retrieved and/or updated. The main available resources are: the “info”, the “configs”, the “services” and the “devices” resources.

The “info” resource is where the general information about the Home Server is stored. When retrieved, it presents all Home Server’s information as explained on Appendix B. On this resource, only the Home Server’s ”name” can be updated through the APIs. All the other informations can only be updated by updating the “settings.py” file and then restarting the whole application.

The “configs” resource stores all the configurations available to the devices on that specific Home Server. Usually this resource is always synchronized with all the configurations present on the mHouse Cloud Platform. However, this may not always be true, because this resource can also be updated by other clients/applications different from the mHouse Cloud Platform, through the Home Server APIs. Nonetheless, the default and usual behaviour is that the updates only come from the mHouse Cloud Platform, and in that case the resource is always synchronized with all user’s configurations. When this resource is retrieved, a list of all available configurations is presented, where each one is presented with its default format, as explained on the Appendix B.

The “services” resource stores all the services available to the devices on that specific Home Server. Usually this resource is always synchronized with the user’s online **Services**, but, once again, this may not always be true, due to the same fact as with the “configs” resource. When this resource is retrieved, a list of all the available **Services** is presented, where each one is presented with its default format, as explained on the section B.

The “devices” resource stores a list of **Devices** that are registered on the Home Server. Each element (child) of that list is a “**Device** representation” resource, and its name is always the ”local ID” of the **Device** it represents on the Home Server[7]. Each “**Device** representation” resource can be retrieved, presenting the general **Device’s** information as explained on Appendix B. In that resource only the ”name” information can be updated through the Home Server APIs, unless the request is being made by the Endpoint/device represented by that resource. In that case, in addition to the ”name”, the fields ”device_type” and ”timeout” can also be updated. Each Endpoint can also delete its ”**Device** representation resource”, thus unregistering itself from the Home Server.

Each “**Device** representation” resource also has three child resources: the “type” resource, the “services” resource and the “state” resource.

The “type” resource can only be retrieved and it presents all detailed information about the **Device’s** type, as explained on the section B. The “services” resource can be retrieved and updated. When retrieved, it presents the list of **Service** IDs to which the **Device** belongs, and to be updated, a list of **Service** IDs must be provided[8].

The **Device** representation “state” resource can also be retrieved and updated. When retrieved, it presents the *current* and the *wanted states* of the correspondent Endpoint/device. Here, *state* means the list of all the **Device’s** properties and their corresponding values. The separation of *current* and *wanted states* is made to ensure data consistency. The *wanted state* is the device’s state that is desirable at a certain moment, and it is the state that is always updated by requests not coming from the correspondent Endpoint/device (e.g. requests coming from the Cloud Platform). The *current state* is the device’s actual and real physical state at a certain moment, and it can only be updated by its correspondent Endpoint/device.

For example: when a device has a light bulb attached to it and the user wants to turn on that light, he/she sends a “turn on” command through the Cloud Platform. When that request/command reaches the Home Server, only the *wanted state* is updated for that device. Then, when that device is notified that its *wanted state* was changed[9], it will update its real physical state, thus performing the desired “turn on” action. After that, that device will send a ”state update request” to the Home Server, which will take effect on the *current state* of its **Device** representation “state” resource, thus synchronizing its *wanted* and *current* states.

The Appendix A presents all the relevant informations about the RESTful APIs that are provided by the Home Server (through which every explained resource can be accessed), like the URIs to access each resource, the requests requirements and the possible responses.

3.5.3 Endpoints Application Prototype

The Endpoint Application prototype was developed with three different versions: for C, C++ and Python languages. The point of developing it with different versions is to prove the framework’s capability to adapt and integrate different technologies. The C version was developed to work mainly with

[7]Check Appendix B, or section 3.4 for better understanding.
[8]Check appendices A and B for more information.
[9]Check the last part of this section (Endpoints Application Prototype) for better understanding.

the ESP8266 Open FreeRTOS SDK on WeMos devices, where more sensors and actuators can be handled due to the ability of having multitasking. The C++ version was developed to work also with WeMos devices, but now programming them with the Arduino IDE and its libraries, making it simpler to program and configure, but reducing the capability of having multitasking. Finally the Python version was developed to work on any machine supporting Python.

All the prototype application versions follow the architecture presented on Figure 27. On the C and C++ versions, the Endpoint Functionality Abstraction Module implements a simple CoAP client, which was specifically developed for this purpose and it has a way to handle the CoAP OBSERVE request callbacks. On the Python version, the same module implements also a simple CoAP client, but now developed with the CoAPthon library.

On all versions, the prototype's Endpoint Main module is fully configurable. The user should program the three configurable functions (*Logic Setup Function*, *Sensing Logical Function* and *Actuating Logical Function*) as explained on section 3.3. For that he/she can also use the prototype code examples provided on the Endpoint's code repository for help. Besides programming those configurable functions, the user needs to provide some required settings of the prototype application. Those settings are the Endpoint's name, type, services and timeout (*ENDPOINT_NAME*, *ENDPOINT_TYPE*, *ENDPOINT_SERVICES*, and *ENDPOINT_TIMEOUT*), and the Home Server CoAP IP address and port (*HOMESERVER_IP_ADDRESS* and *HOMESERVER_PORT*)[10].

After being properly programmed by the user to handle each specific physical sensor and/or actuator, the Endpoint prototype application automatically interacts with the remaining prototype's system components in the correct way, by using the already coded *Endpoint State Machine* and the Endpoint Functionality Abstraction Module.

Therefore, when the device turns on, the first step of the Endpoint's state machine is to register it self on the Home Server. To do it, the Endpoint application sends a "register device request"[11] to the Home Server, using the CoAP RESTful API the it provides. If that registration is successful, an Endpoint representation for that specific device is created on the Home Server's database, and a "local ID" is provided by the Home Server to the Endpoint, with which the Endpoint can perform the next steps.

If for some reason the Endpoint's representation already exists on the Home Server, i.e. there's a device already registered with the IP address provided on "register device request", an error will be received by the Endpoint stating that "a device with that IP address already exists and it has the local ID *X*". In this case, the Endpoint uses the ID provided on that error message ("*X*") to update the its representation on the Home Server with its current configured settings. To do that, the Endpoint sends an "update device representation" to the Home Server[12].

To allow receiving data/commands from the Cloud Platform, the CoAP's OBSERVE mechanism is used. Thus, after the first step (registration on Home Server), the Endpoint should send a CoAP OBSERVE request to its representation on the Home Server[13]. By doing that, the Endpoint will receive

[10] All these informations can be better understood by checking appendices A, B and section 3.4.
[11] POST request - check Appendix A for more info.
[12] PUT request - check Appendix A for more info.
[13] Specifically to its "state" resource - see section 3.5.2 and Appendix A.

”notifications” every time a change occurs on its representation ”state” on the Home Server, and it will always update its real/current state accordingly to those changes. Therefore, when a command (”device state update”) comes from the Cloud Platform, it takes effect on the Endpoint’s representation ”state” resource on the Home Server, and that state change is then ”propagated” to the Endpoint, because it is ”observing” that resource.

After these two first steps, the Endpoint is considered to be properly ”present” on the system. Now it should periodically send “state update requests” to its representation ”state” resource on the Home Server[14]. By doing that, the Endpoint periodically “broadcasts” its sensed physical property changes to the remaining mHouse components, ensuring the upstream flow of data (from the Endpoints to the Cloud). The periodicity with which this ”state update requests” should be sent by the Endpoint application, must be directly related to the Endpoint’s “timeout” setting and/or to the user needs/wishes.

For example, if the user has a temperature sensor which is supposed to broadcast the temperature value every 5 minutes, then the ”timeout” setting for that device should be 5 minutes. But, if on another situation the user only wants to broadcast the temperature value when it changes, even if that changes rarely occur (e.g. changes only occur once on a 30 minutes interval), and the user still sets the ”timeout” to remain as 5 minutes, the Endpoint will automatically send a state update 2 seconds before that “timeout” value runs out, in order to notify the Home Server that it is still ”alive”, and it is not be considered “down”, even on these situations where the device state changes rarely occur within the ”timeout” interval as usually expected.

To detect network failures, and to always maintain the Endpoints ”synchronized” with the Home Server, every request exchange between the Endpoints and the Home Server should be confirmable. Thus, every request from or to the Endpoints should have a response, even if it is just an acknowledge of the request reception. If for some reason there is no response to some Endpoint’s request, the Endpoint will try to resend that request until it reaches a configurable ”home server timeout”. After that timeout, if the Home Server remains unresponsive, the Endpoint will consider the Home Server to be “down”, and then it comes back to the first step of its state machine, where it periodically sends a “register device request” until the Home Server responds to it. In that state, the Endpoint uses a random periodic interval to send those “register device requests” in order to minimize the overloading of the Home Server when it restarts, like when several Endpoints try to register themselves at the same time. On the developed prototype applications, the endpoints wait a random number of seconds between 1 and 10, before each one sends its ”device register request” to the Home Server, but other types of time distribution can be used to achieve the same or even better performance.

This mechanism works properly as a ”synchronization” method between the Endpoints and the Home Server, because every Home Server always starts ”empty”, i.e without any records about any device, as it is stateless by nature. Thus, the first thing that a Home Server expects to receive when it restarts, is a registration request from every device, and only then it can receive and handle all the ”state update requests”.

[14]PUT requests - check Appendix A for more info.

4. Tests and Verifications

In order to always verify and test all the functionalities detailed for each component and subcomponent of the system, an Agile methodology was employed. Thus, at the same time each component was being developed, several tests were made to verify the their correct behaviour. Nonetheless, after finishing the development, three main overall test cases were done to prove the framework's utility, and to verify if the proposed goals were properly achieved.

These three test cases were done with the objective to test all the framework's capabilities and functionalities, and not specifically its performance. Thus, they are ”utility”/”functionality” tests and not ”performance” ones, i.e. they test and verify which functionalities the framework offers, and not if the framework performs the best way possible or not. This decision was made because performance tests would take much longer, would be much harder to employ, and the achieved results would not bring much relevant information. On the other hand, ”utility” tests, besides being much simpler and faster to employ, they help to verify and prove that the proposed framework's goals were correctly achieved, and help to demonstrate the several functionalities that the framework offers. Moreover, by doing ”utility” tests, some basic qualitative performance results can be inferred as well, such as if the system is fast or slow when responding to commands, or if it can easily support several different devices at the same time or not.

The three test cases that were employed to test the mHouse Framework were:

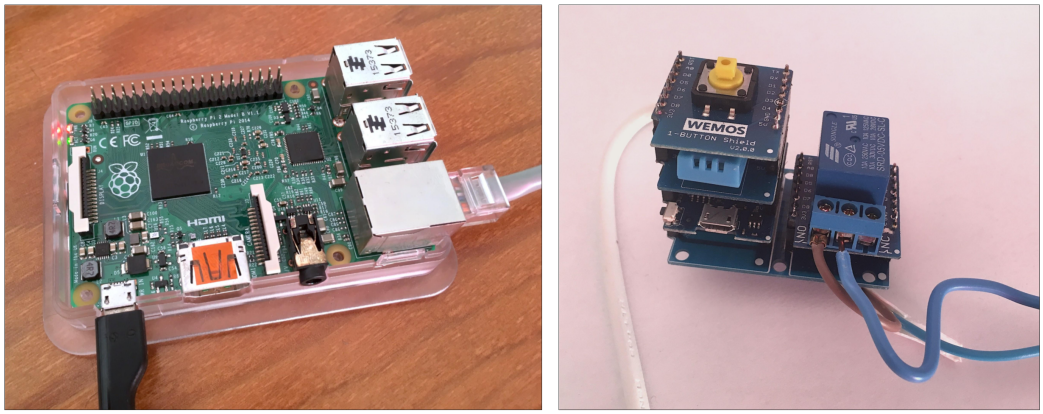
- Deployment of the framework's prototype on a home environment, and test its main functionalities (monitoring and controlling the devices) for some days, thus validating the correct operation of the overall system (infrastructure), and also the easiness to deploy, install, configure, and use the framework.

- Integration of the framework with other cloud service, which in this case is the AWS IoT Platform, in order to prove the framework's openness, generality, modularity and easiness, not only as a ”whole”/turnkey system, but also on its usage by parts. This test was also performed to validate that the framework can be easily integrated with other ”bigger”, and possibly more powerful, cloud services, because, by being possible to integrate it with a service like AWS IoT Platform, all the other services and applications that AWS provides, and that can be integrated with AWS IoT Platform, become out-of-the-box available to use also with the mHouse Framework.

- Development of a basic smartphone application, which uses the Cloud Platform RESTful API, to get information and issue commands towards the system, thus validating the correct behaviour and utility of this API, and once again all the framework's correct operation.

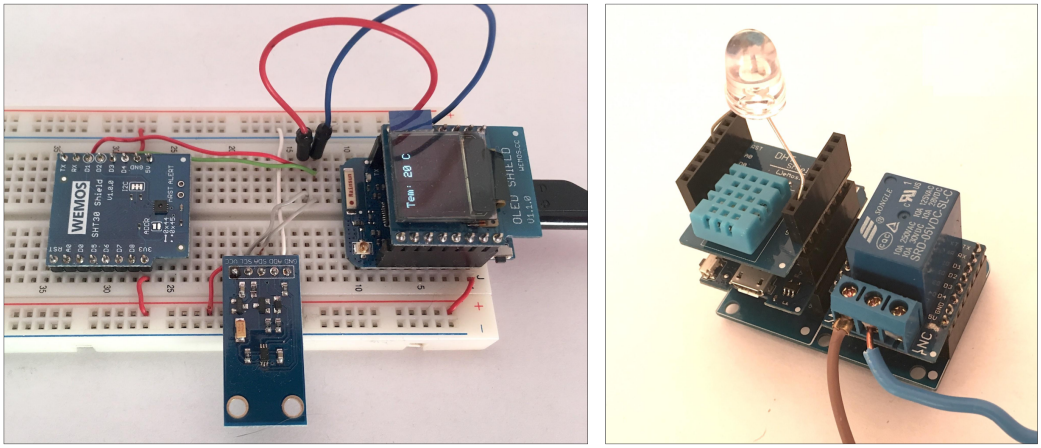
4.1 First Test Case: Deployment of the prototype on a home environment

On this test case, the mHouse Framework was deployed on two different house environments, during 1 week on each one of them. The first one was a country house with 3 floors, each one with approximately 250 m^2 and 10 different divisions. The second one was a city flat with just 1 floor, with approximately 25 m^2 and 4 divisions. On each scenario, the mHouse Home Server prototype application was deployed on a Raspberry Pi 2 which was connected to the home network router via Ethernet cable (Figure 41 (a)), and the mHouse Endpoints Network was formed by 3 WeMos devices and 1 Raspberry Pi 1, where the WeMos devices were connected to the home network via WiFi and the Raspberry Pi 1 via Ethernet cable.



(a) (b)

Figure 40: (a) - Raspberry Pi 2 used as Home Server. (b) - First WeMos device with a relay, an analog temperature and humidity sensor, and a button attached.



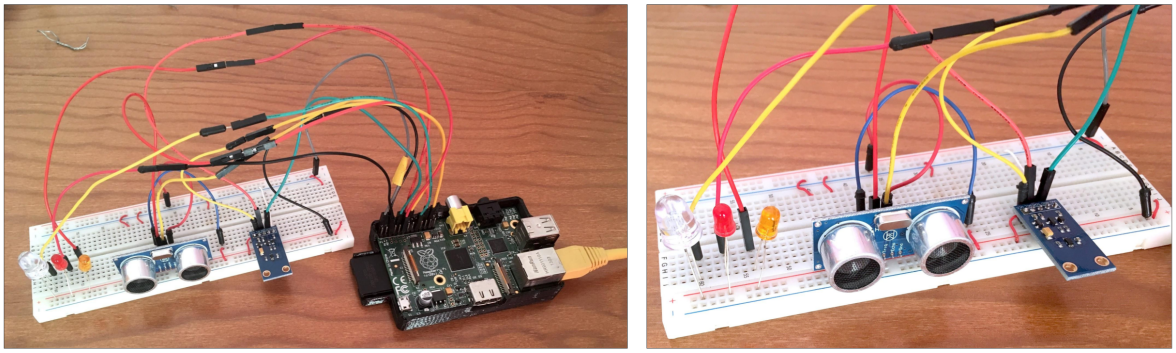
(a) (b)

Figure 41: (a) - Second WeMos device with a digital temperature and humidity sensor, a light sensor and an oled display attached. (b) - Third WeMos device with a relay, an analog temperature and humidity sensor, and a led light attached.

The first WeMos Endpoint/device had a relay, an analog temperature and humidity sensor, and a button attached to it - Figure 40 (b). The sensor was used to retrieve the temperature and humidity of the correspondent house division, and the relay was connected to a light bulb which could be turned on or off through the framework. The button was used to locally control the relay in order to locally turn the light on or off. The prototype application version used on this Endpoint was the C one, developed with the ESP8266 Open FreeRTOS SDK [72].

The second WeMos Endpoint had a digital temperature and humidity sensor, a light sensor and an oled display attached to it - Figure 41 (a). The attached sensors were used to retrieve the temperature, humidity and amount of light on the correspondent house division. The oled display was used to locally present all the collected information. On the mHouse Cloud Platform it was possible to choose which information to present on this display. The prototype application version used on this Endpoint was the C++ one, developed with the Arduino IDE and libraries.

The third WeMos Endpoint had another relay, another analog temperature and humidity sensor, and a led light attached to it - Figure 41 (b). Like on the first WeMos Endpoint, the sensor was used to retrieve the temperature and humidity of the correspondent house division, and the relay was connected to a light bulb which could be turned on or off through the framework. The led light was used to exemplify an adjustable light that could be remotely regulated through the framework. The prototype application version used on this Endpoint was also the C++ one.



(a) (b)

Figure 42: (a) - Raspberry Pi 1 used as fourth Endpoint device. (b) - Focus on the light sensor, ultrasonic sensor and set of led lights attached to the device.

The fourth Endpoint, which was deployed to the Raspberry Pi 1, had a light sensor, a ultrasonic sensor and a set of led lights attached to it - Figure 42. The light sensor was used to retrieve the amount of light on the correspondent house division, the ultrasonic sensor to check presences/passages of people on the correspondent division, and the led lights were controlled by the sensors and/or remotely. The prototype application version used on this Endpoint was the Python one.

To properly test the framework on both house environments, during the tests time, the data collected by the devices was periodically retrieved and stored on the mHouse Cloud Platform. From time to time, some queries were performed on the available collected data in order to prove that the data can not only be stored, but can also be analyzed every time it is necessary. Moreover, some commands were sent to the different devices also using the mHouse Cloud Platform. This way, the ability to access

(retrieve data) and control (send commands) the different devices was properly tested and, implicitly, the ability to monitor the different devices and servers was also tested, as their monitoring was always working on the background.

In order to connect each device/Endpoint to the mHouse Framework, the proper **Device Types**, **Property Types** and **Value Types**, were created on the mHouse Cloud Platform, and then assigned to the respective physical devices. Therefore, the ability to create, map and manage the different devices according to the developed Abstraction Model, was properly tested, as its correct operation was also crucial to perform the remaining tests, by allowing the device connection to the mHouse Framework.

On the other hand, on the mHouse Cloud Platform, the different devices were organized by their position, i.e. the **House**, **Areas** and **Divisions** where each device was placed were created and properly assigned to them. The devices were also organized by their functionality, i.e. the **Services** to which each device belongs were created and properly assigned as well. This way, the ability to create **Houses**, **Areas**, **Divisions** and **Services**, and then manage them and their respective devices, was also properly tested.

Performing this test case allowed to demonstrate the correct operation of the overall framework system, and of each one of its constituent components, as the several functionalities and capabilities, that were proposed to be given by the framework, were properly tested, proving the framework's easy usage and correct operation.

Moreover, this test case also demonstrates the achievement of several proposed objectives, like the easiness to deploy the framework solution on very different house environments, the ability to access and control the different devices from anywhere and at anytime using the mHouse Cloud Platform, and also the framework's ability to, through its Abstraction Model, include very different types of devices that can have very different kinds of sensors and actuators attached to them, furthermore proving the framework's generic, interoperable, global and possibly heterogeneous essence.

4.2 Second Test Case: Integration with AWS IoT Platform

On this test case, the mHouse Framework was integrated with the AWS IoT Platform service. Specifically, a new communicator module was developed and added to the "Cloud Communicators" sub-component on the mHouse Home Server prototype application. Therefore, this communicator is already coded on the available prototype's application, being optional to use. When it is used, it properly allows to access and control the devices that are connected to the Home Server from the AWS IoT Platform. In order to work correctly, the Home Server must be configured with the user's AWS credentials. The new communicator module was developed with the AWS Cloud SDK for Python (boto3 [73]) and the AWS IoT SDK for Python [74]. It allows the Home Server to seamlessly register and unregister the devices on the AWS IoT Platform, and also to send the device's data to it. It also allows to receive commands from the AWS IoT Platform, thus being possible to control of the devices simultaneously from the AWS IoT Platform and/or from the mHouse Cloud Platform.

Allowing this integration of the mHouse Home Server with the AWS IoT Platform is a huge advantage because, by doing that, the Home Server can be also automatically integrated with all the other AWS Cloud services and applications, which can also be used by the user to build complex and smarter solutions, as presented on the section 2.1.3.

Moreover, this test demonstrates that it is easy and fast to add new communicators to the "Cloud Communicators" subcomponent on the mHouse Home Server, thus proving that the framework is open, modular and generic, as it can easily be integrated with different services, each one of them with possibly very specific requirements and characteristics. It is also demonstrated that the framework can be used not only as a "whole"/turnkey system, but also as a modular solution, where just parts, components or subcomponents of it, can be used integrated with or on other services, applications and/or solutions.

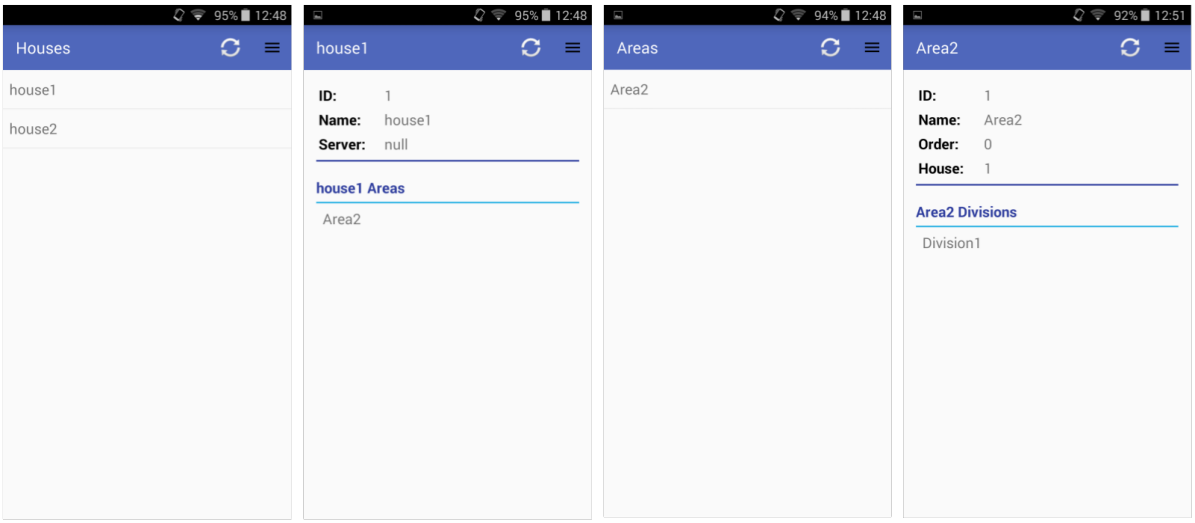
4.3 Third Test Case: Development of a smartphone application

On this test case a simple mobile application was developed to Android smartphones, that enables users to access the mHouse Cloud Platform from a native mobile application and not only from a web browser. This development was performed in order to test and demonstrate that the mHouse Cloud Platform can not only be accessed using its web HTML interface, but also by a diversity of other services or applications, which can use the provided RESTful API[1] to access all the framework's resources.

Therefore, a very simple Android application was developed to allow users to access and control their **Houses**, **Areas**, **Divisions**, **Servers** and/or **Devices**. Note that this is really a very simple mobile application that was only developed to test the working behaviour and capabilities of the developed RESTful API, and it should not be seen as a full (strong) mobile application, neither as a current substitute of the web application/interface, which also works on smartphone browsers by being responsive. Figures 43 and 44 show some screenshots of the developed application, where it can be seen the different resources being accessed and controlled, like the user's **Houses**, **Areas**, and **Devices**. Note that, as it is only a simple application for test purposes, it has a very similar appearance to the developed responsive web application, but on future work, progresses can be done to develop a stronger mobile application, that could provide a more intuitive and simple graphical interface to the users, in order to simplify the access, management and control of the resources made available by the mHouse Framework.

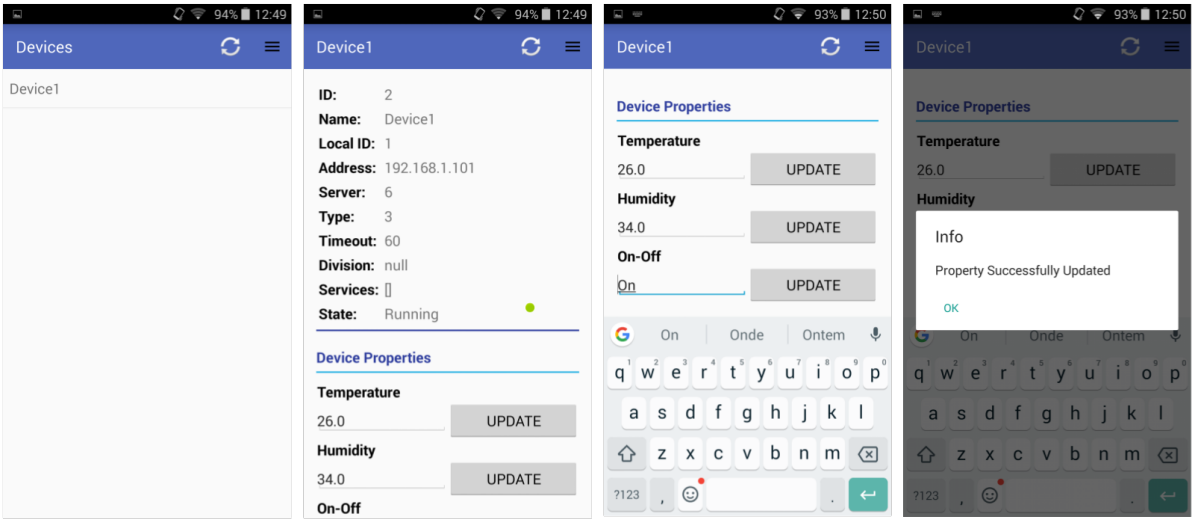
Finally, with this simple test case, the capability to use the mHouse RESTful API from other services and/or applications, to access all the framework's resources, was properly tested and demonstrated, as it proves to be possible to get all kind of information and to issue commands towards the whole framework system, thus accessing and/or controlling it. Moreover, in the background, this test case also helped to test and demonstrate the overall framework functionalities and capabilities, and that they are properly implemented and correctly operating.

[1] In this case the HTTP one, which works *over the system* to access the Cloud Platform.



(a) (b) (c) (d)

Figure 43: Test Application Example Screens: User's Houses (a), a House in detail (b), Areas (c), and an Area in detail (d).



(a) (b) (c) (d)

Figure 44: Test Application Example Screens: User's Devices (a), a Device in detail (b), Updating a device property (c),and after successfully updating a device property (d).

5. Conclusions

5.1 Achievements

The main objective of this thesis was the development of a framework solution that allows to build an open, modular and generic domotic system taking into account both the domotics and IoT fields characteristics and requirements, like interoperability, heterogeneity, ease of deployment and usage, robustness, global access and control.

Although the Home Automation and IoT fields are currently exponentially growing and several commercial solutions can be found (AWS IoT Platform, Google Cloud IoT Core, Microsoft IoT Suite, Samsung SmartThings, Philips Hue, Apple HomeKit, Amazon Echo, etc.) there are still some unresolved problems. Some of those are the complexity, cost and intrusiveness of the systems, the lack of network and device interoperability, and the lack of open, expandable and generic interfaces and systems, as most part of the solutions are proprietary, not completely open and generic, reducing the user's capability to build new applications and to contribute to the system's or solution's evolution towards new, better and stronger functionalities and capabilities.

Moreover, many of the solutions cannot be considered truly IoT ones, as they were built only to address some specific set of objects/things (e.g. Philips Hue), and sometimes they also do not have interfaces open and generic enough to allow their integration on larger and more complex systems, like the IoT requires (e.g. Belkin WEMO, Apple HomeKit).

Besides that, solutions and technologies with a more generic and interoperable mindset can also be found, such as DomoBus, and they can be integrated with other technologies, solutions and paradigms that can properly address the IoT's interoperable, generic and heterogeneous essence, such as CoAP, JSON, UDP, IP and the lower level PAN,LAN and LPWAN technologies, and the REST and Cloud paradigms.

The developed framework - the mHouse Framework - is a high-level RESTful solution, based on DomoBus, that provides an open set of specifications to the overall system architecture and its RESTful interfaces, allowing the users/developers to build generic and expandable domotic system solutions. It is capable to integrate devices with very different characteristics, requirements, and base technologies, and also to access, control and manage such devices from anywhere and at any time. Moreover, it is a modular solution which can not only be used as a whole/turnkey system, but users/developers can also integrate parts of it on other systems or solutions, or even the other way around, to integrate parts of other systems or solutions on it. Therefore, as it is based on standard and open technologies like CoAP, JSON and REST, it is ideal to build big domotic systems included on the IoT, where it can also work side-by-side with other technologies, which can bring other advantages to the final solution as well.

The framework allows users/developers to connect all their devices[1] and to access and control them, i.e. users can retrieve and store the data collected by those devices, and send commands to those devices as well, making use of the mHouse Cloud Platform, where all resources appear in an automatic way after the installation/deployment of the framework. Moreover, the mHouse Framework allows users/developers to organize their devices and house facilities by location/space, by creating and managing virtual entities like Houses, Areas and Divisions, to where the Devices can be assigned and also organized by their functionality, by creating and managing virtual Services to which the Devices can by assigned as well. Finally, the mHouse Framework provides a solution that always monitors all connected devices, making sure that all the information about their failures and working behaviour is properly registered and presented to the user, making it simple to build alarmistic applications where the users can be notified about device failures, misbehaviour or emergency situations.

It is also important to notice that, in order to test all the functionalities, capabilities and specifications developed to the mHouse framework, several functional tests were performed on different house scenarios, and also with different testing approaches, as presented on chapter 4.

To conclude, all the proposed objectives were properly addressed and successfully achieved with the developed solution - the mHouse Framework -, which was presented on this thesis, where all the development steps, documentation and executed tests are provided and detailed in order to help others to understand and possibly use all the work done when building new domotic and IoT solutions, applications and/or services.

5.2 Future Work

Despite the fact that the developed work gives a full working solution with a prototype that offers a vast and great set of functionalities and capabilities, some parts were not fully developed as they were not crucial to achieve the main proposed goals for this dissertation, and so they can and should be developed in future work.

Firstly the most relevant topic that was somehow leaved a little bit aside was security. As stated on the proposed objectives, security is a vast subject that could not be fully investigated and developed on the scope of this thesis, as it can be seen as a whole new theme/work for a new dissertation just by itself. Nonetheless, as it is one of the most critical aspects of IoT systems, and as only the basics of it were addressed on the development of the prototype solution by adoption of just some basic security mechanisms already available, some improvements can be done in the future.

Therefore, a first approach to security on the developed work can be to include HTTPS rather than only HTTP on the communications between the mHouse Cloud Platform and the mHouse Home Servers.

A second approach can be to include some stronger authentication mechanisms on the mHouse Home Server, as it currently does not require the requests to be somehow authenticated. Moreover, although authentication is required on the requests made to the mHouse Cloud Platform, this authenti-

[1] Which can have a large set of sensors and/or actuators attached to them.

cation is only based on the Basic Auth system with a simple username and password, and thus some stronger authentication mechanism can also be employed and developed to this end.

On a lower level, a third approach to security can be to include DTLS on all CoAP communications, as the current solution only uses simple unsecured CoAP communications provided by the CoAPthon implementation of CoAP. Moreover, currently, devices are authenticated on the mHouse Home Server through their IP address, i.e. some actions (e.g. updating the device's current state) can only be performed on a device by itself and not from any other device, and this validation is currently based on the IP address of the device makes the action request[2], making it open to forgery attacks. Thus, an authentication mechanism that do not relies on only the IP address of the several devices, should be employed or developed to this kind of situation.

A second topic that should be developed in future work is the robustness and load balancing of the overall system's architecture. As stated on section 3.3, the overall home network architecture has a start topology where the Home Server is the center point. Therefore, it presents a single point of failure of the overall architecture, as all the requests and communication flows pass and are supported by the Home Server. Thus, a more complex and distributed architecture should be thought of, developed and employed, where, for example, the system could have several Home Server's on the same house, connected and communicating with each other, and then the overall traffic and work load could be better balanced and that single point of failure scenario be solved. Another approach would make each Home Server provide its own WiFi network where the devices/endpoints can be connected, decoupling the devices/endpoints and the Home Server from the home WiFi network/router, but of course this only makes sense for solutions where WiFi is used on the lower network layers.

A third topic that can be developed in future work is to provide new and more high-level functionalities, like the ability to share house access between users, allowing, for example, a family of users to share the same mHouse account to control their house; the ability to create automation scenarios, like proposed by DomoBus, but where the automation points are made both on the high-level (on the Cloud Platform, for more complex automation patterns) and on the low-level (on each device/endpoint or on the Home Server, for simpler automation patterns like turn a light on or off accordingly to a presence sensor). Moreover, an improvement of the overall online Cloud Platform frontend appearance/look can also be done, to provide more intuitive and simpler ways to access, control and manage the different resources and entities.

Finally, given the open, modular and generic essence of the developed framework, some gateway modules can be developed to the mHouse framework in order to integrate it with a larger set of devices, which can already work with low level technologies that do not natively support IP or that are different from WiFi, and can be also typically found on home automation and/or IoT scenarios, like ZigBee or Z-Wave.

[2]Ex: If device 1, with IP 1.2.3.4, tries to update the current state of the device 2, which has the IP 5.6.7.8, it gets blocked.

6. Bibliography

[1] M. Andersson, A. Bastos, S. Blomberg, E. Nyborg, A. Palsson, and P. Tham. "Home Automation for Existing Devices". Technical report, Chalmers University Of Technology - University Of Gothenburg, June, 2015. Last accessed on 6/2017. Source: http://publications.lib.chalmers.se/records/fulltext/218961/218961.pdf.

[2] Cisco. "Internet of Things", June, 2016. Last accessed on 6/2017. Source: http://www.cisco.com/c/dam/en/us/products/collateral/se/internet-of-things/at-a-glance-c45-731471.pdf.

[3] Gartner. "Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015", November, 2015. Last accessed on 6/2017. Source: http://www.gartner.com/newsroom/id/3165317.

[4] McKinsey&Company. "Unlocking the potential of the Internet of Things", June, 2015. Last accessed on 6/2017. Source: http://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/the-internet-of-things-the-value-of-digitizing-the-physical-world.

[5] Ilmari Ahonen. "Internet of things – wireless technologies in home automation solutions". Technical report, University of Jyväskäy, 2015. Last accessed on 6/2017. Source: https://jyx.jyu.fi/dspace/bitstream/handle/123456789/45101/URN%3ANBN%3Afi%3Ajyu-201501161120.pdf?sequence=1.

[6] Andrew Meola, Business Insider. "How IoT & Smart Home Automation Will Change the Way We Live", December, 2016. Last accessed on 6/2017. Source: http://www.businessinsider.com/internet-of-things-smart-home-automation-2016-8.

[7] A. B. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, and C. Dixon. "Home Automation in the Wild: Challenges and Opportunities". Technical report, Microsoft Research and University of Washington, May, 2011. Last accessed on 6/2017. Source: https://www.microsoft.com/en-us/research/publication/home-automation-in-the-wild-challenges-and-opportunities/.

[8] Renato Jorge Caleira Nunes. "Implementing Tiny Embedded Systems With Networking Capabilities". Technical report, Instituto Superior Técnico / INESC-ID, February, 2005. Last accessed on 6/2017. Source: IADIS International Conference on Applied Computing 2005, Algarve, Portugal, http://domobus.net/papers/05-IADIS05-long.pdf.

[9] S. Mitchell, N. Villa, M. Stewart-Weeks, and A. Lange. "The Internet of Everything for Cities". *Cisco*, 2013. Last accessed on 6/2017. Source: https://www.cisco.com/c/dam/en_us/solutions/industries/docs/gov/everything-for-cities.pdf.

[10] Renato Jorge Caleira Nunes. ”DomoBus Official Website”, Last accessed on 9/2017. Source: http://domobus.net.

[11] ITU-T. Recommendation Y.2060: ”Overview of the Internet of things”. Technical report, International Telecommunications Union, June, 2012. Last accessed on 6/2017. Source: https://www.itu.int/rec/T-REC-Y.2060-201206-I.

[12] Cammy Team. ”What Is The Internet Of Things”, Last accessed on 08/2017. Source: https://blog.cammy.com/the-internet-of-things.

[13] Arnaud Germis. ”An Open Home Automation System, Analysis And Implementation of New Solutions for HomePort”. Master's thesis, Aalborg University, Denmark, May, 2010. Last accessed on 6/2017. Source: http://projekter.aau.dk/projekter/files/61077302/1274717394.pdf.

[14] Paul Korzeniowski, TechTarget. ”IoT battle of the titans: Bluetooth 5 vs. Wi-Fi HaLow”, August, 2016. Last accessed on 6/2017. http://internetofthingsagenda.techtarget.com/feature/IoT-battle-of-the-titans-Bluetooth-50-vs-Wi-Fi-HaLow.

[15] Z. Shelby and C. Bormann. *”6LoWPAN: The Wireless Embedded Internet”*. John Wiley and Sons Ltd, November, 2009. Last accessed on 6/2017. Source: http://keshi.ubiwna.org/2015IotComm/6LoWPAN_The%20Wireless%20Embedded%20Internet.pdf.

[16] Clemens Puhringer. ”Cloud Computing for Home Automation”. Technical report, Faculty of Informatics, Vienna University of Technology, March, 2014. Last accessed on 6/2017. Source: https://www.auto.tuwien.ac.at/bib/pdf_TR/TR0167.pdf.

[17] N. Kushalnagar, G. Montenegro, and C. Schumacher. RFC 4919 - ”IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs):Overview, Assumptions, Problem Statement, and Goals”. Technical report, Internet Engineering Task Force (IETF), August, 2007. Last accessed on 6/2017. Source: https://tools.ietf.org/html/rfc4919.

[18] Ericsson. ”Cellular networks for massive IoT”. *Ericsson*, January, 2016. Last accessed on 6/2017. Source: https://www.ericsson.com/res/docs/whitepapers/wp_iot.pdf.

[19] CNXSoft. ”Comparison Table of Low Power WAN Standards for Industrial Applications”, September, 2015. Last accessed on 6/2017. Source: http://www.cnx-software.com/2015/09/21/comparison-table-of-low-power-wan-standards-for-industrial-applications/.

[20] Margaret Rouse, TechTarget. ”LPWAN (low-power wide area network)”, November, 2015. Last accessed on 6/2017. Source: http://internetofthingsagenda.techtarget.com/definition/LPWAN-low-power-wide-area-network.

[21] LinkLabs. ”SigFox Vs. LoRa: A Comparison Between Technologies & Business Models”, January, 2016. Last accessed on 6/2017. Source: http://www.link-labs.com/sigfox-vs-lora/.

[22] W3C. ”Web Services Architecture”. Technical report, World Wide Web Consortium, February, 2004. Last accessed on 6/2017. Source: https://www.w3.org/TR/ws-arch/.

[23] F. Jammes and H. Smit. "Service-Oriented Paradigms In Industrial Automation". In *Parallel and Distributed Computing and Networks*, February, 2005. Last accessed on 6/2017.

[24] Fuguo Huang. "Web Technologies for the Internet of Things". Master's thesis, Aalto University School of Science, July, 2013. Last accessed on 6/2017. Source: https://into.aalto.fi/download/attachments/12324178/Huang_Fuguo_thesis_2.pdf.

[25] IST. Internet Based Systems Architecture course presentation: "04 - Webservices", 2016, Last accessed on 6/2017. Source: https://fenix.tecnico.ulisboa.pt/downloadFile/1689468335573497/05-a-webservices.pdf.

[26] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *"Distributed Systems, Concepts and Designs"*. Addison-Wesley, January, 2013. Last accessed on 6/2017. Source: https://azmuri.files.wordpress.com/2013/09/george-coulouris-distributed-systems-concepts-and-design-5th-edition.pdf.

[27] Roy Thomas Fielding. *"Architectural Styles and the Design of Network-based Software Architectures"*. PhD thesis, University of California, 2000. Last accessed on 6/2017. Source: http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[28] Douglas E. Comer. *"Computer Networks and Internets"*. Pearson Education, Inc., 2008. Last accessed on 6/2017. Source: https://web.njit.edu/~bm245/IT420/Computer-Networks-and-Internets-Fifth-Edition.pdf.

[29] Frank Matthias Kovatsch. *"Scalable Web Technology for the Internet of Things"*. PhD thesis, ETH Zurich, 2015. Last accessed on 6/2017. Source: http://www.vs.inf.ethz.ch/publ/papers/mkovatsc-2015-dissertation.pdf.

[30] Z. Shelby, K. Hartke, and C. Bormann. RFC 7252 - "The Constrained Application Protocol (CoAP)". Technical report, Internet Engineering Task Force (IETF), June, 2014. Last accessed on 6/2017. Source: https://tools.ietf.org/html/rfc7252.

[31] Z. Shelby. RFC 6690 - "Constrained RESTful Environments (CoRE) Link Format. Technical report, Internet Engineering Task Force (IETF), August, 2012. Last accessed on 6/2017. Source: https://tools.ietf.org/html/rfc6690.

[32] Toby Jaffey. "MQTT and CoAP - IoT Protocols", February, 2014. Last accessed on 6/2017. Source: https://eclipse.org/community/eclipse_newsletter/2014/february/article2.php.

[33] Amazon. "AWS Greengrass Website", Last accessed on 09/2017. Source: https://aws.amazon.com/greengrass/.

[34] Amazon. "AWS IoT Platform Website", Last accessed on 09/2017. Source: https://aws.amazon.com/iot-platform/.

[35] Microsoft. "Azure IoT Suite Website", Last accessed on 9/2017. Source: https://www.microsoft.com/en-us/internet-of-things/azure-iot-suite.

[36] Microsoft. "Windows IoT Core Website", Last accessed on 9/2017. Source: https://developer.microsoft.com/en-us/windows/iot.

[37] Microsoft. "Azure IoT Suite Documentation Website", Last accessed on 9/2017. Source: https://docs.microsoft.com/en-us/azure/iot-suite/iot-suite-overview.

[38] Microsoft. "Azure IoT Hub Documentation Website", Last accessed on 9/2017. Source: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-what-is-iot-hub.

[39] Microsoft. "Azure IoT Stream Analytics Documentation Website", Last accessed on 9/2017. Source: https://docs.microsoft.com/en-us/azure/stream-analytics/stream-analytics-introduction.

[40] Microsoft. "Windows IoT Core Documentation Website", Last accessed on 9/2017. Source: https://developer.microsoft.com/en-us/windows/iot/Explore/IoTCore.

[41] Google. "Google IoT Core Website", Last accessed on 9/2017. Source: https://cloud.google.com/iot-core/.

[42] Google. "Android Things Website", Last accessed on 9/2017. Source: https://developer.android.com/things/hardware/index.html.

[43] Google. "Google Cloud IoT Website", Last accessed on 9/2017. Source: https://cloud.google.com/solutions/iot/.

[44] Google. "Android Things Documentation Website", Last accessed on 9/2017. Source: https://developer.android.com/things/sdk/index.html.

[45] David Cuacos Encarnación. "Automation and Information Technology Solutions for Buildings and Homes". Master's thesis, Universidad Carlos III of Madrid, April, 2008. Last accessed on 6/2017. Source: http://e-archivo.uc3m.es/bitstream/handle/10016/8197/David_Cuacos_Masther_Thesis.pdf?sequence=2.

[46] Tanel Lips. "Analysis and Comparison of Smart Homes". Technical report, University Of Tartu, Faculty Of Mathematics And Computer Science, 2012. Last accessed on 6/2017. Source: https://comserv.cs.ut.ee/home/files/thesis.pdf?study=ATILoputoo&reference=ED12992FDFBF9DB956E15960D11282F61224CD64.

[47] Simply Automated Inc. "Home Automation blog", August, 2014. Last accessed on 6/2017. http://simply-automated.com/blog/2014/09/home-automation-technology-protocol-best/.

[48] Alessandro Gonella. "Home Automation Standards", November, 2013. Last accessed on 6/2017. Source: http://www.alessandrogonella.com/2013/11/home-automation-standards/.

[49] "Home Automation blog", August, 2014. Last accessed on 6/2017. http://buildyoursmarthome.co/home-automation/protocols/.

[50] KNX Association. ”KNX Official Website”, Last accessed on 12/2016. https://www.knx.org/knx-en/index.php.

[51] Renato Jorge Caleira Nunes. ”DomoBus - A New Approach to Home Automation”. Technical report, IST - Technical University of Lisbon / INESC-ID, July, 2003. Last accessed on 6/2017. Source: 8CLEEE - 8th International Congress on Electrical Engineering, Portugal, pp.2.19-2.24, http://domobus.net/papers/03-CLEEE03.pdf.

[52] Miguel Fonseca Gonçalves Martinho. ”Emulador de Dispositivos Domóticos”. Master's thesis, Instituto Superior Técnico, November, 2016.

[53] Renato Jorge Caleira Nunes. ”Modelo de Especificação e Programação de um sistema domótico”. Technical report, Instituto Superior Técnico / INESC-ID, October, 2004. Last accessed on 6/2017. Source: IADIS Conferência Ibero-Americana WWW/Internet 2004, Madrid, Spain, pp. 377-384, http://domobus.net/papers/04-CIAWI04.pdf.

[54] Renato Jorge Caleira Nunes. *”The DomoBus System Specification Language V2.0c”*, February, 2016. Last accessed on 6/2017. Source: http://domobus.net/papers/.

[55] Belkin. ”Belkin WeMo Official Website”, Last accessed on 9/2017. Source: http://www.belkin.com/us/Products/home-automation/c/wemo-home-automation/.

[56] Apple. ”Belkin WeMo Application Shop page”, Last accessed on 9/2017. Source: https://itunes.apple.com/pt/app/wemo/id511376996?mt=8.

[57] Philips. ”Philips Hue Official Website”, Last accessed on 9/2017. Source: http://www2.meethue.com/en-us.

[58] Philips. ”Philips Hue About Webpage”, Last accessed on 9/2017. Source: http://www2.meethue.com/en-us/about-hue.

[59] Philips. ”Philips Hue Documentation Webpage”, Last accessed on 9/2017. Source: https://developers.meethue.com/documentation/how-hue-works.

[60] Apple. ”Philips Hue Application Shop page”, Last accessed on 9/2017. Source: https://itunes.apple.com/us/app/philips-hue-gen-2/id1055281310?ls=1&mt=8.

[61] Samsung SmartThings. ”Samsung SmartThings Official Website”, Last accessed on 9/2017. Source: https://www.smartthings.com/.

[62] Samsung SmartThings. ”Samsung SmartThings Documentation Webpage”, Last accessed on 9/2017. Source: http://docs.smartthings.com/en/latest/architecture/index.html.

[63] Apple. ”Samsung SmartThings Application Shop page”, Last accessed on 9/2017. Source: https://itunes.apple.com/pt/app/smartthings-mobile/id590800740?mt=8.

[64] Apple. ”Apple HomeKit Official Website”, Last accessed on 9/2017. Source: http://www.apple.com/ios/home/.

[65] Apple. "Apple HomeKit Documentation Website", Last accessed on 9/2017. Source: https://developer.apple.com/homekit/.

[66] Amazon. "Amazon Alexa Documentation Webpage", Last accessed on 9/2017. Source: https://developer.amazon.com/alexa.

[67] Amazon. "Amazon Echo Webpage", Last accessed on 9/2017. Source: https://www.amazon.com/Amazon-Echo-Bluetooth-Speaker-with-WiFi-Alexa/dp/B00X4WHP5E.

[68] Google. "Google Assistant Webpage", Last accessed on 9/2017. Source: https://assistant.google.com/.

[69] Google. "Google Home Webpage", Last accessed on 9/2017. Source: https://madeby.google.com/home/.

[70] Kenneth Reitz. "Requests Official Website", Last accessed on 9/2017. Source: http://docs.python-requests.org/en/master/.

[71] Marcel Hellkamp. "Bottle Official Website", Last accessed on 9/2017. Source: http://bottlepy.org/docs/dev/.

[72] Espressif. "ESP-OPEN-RTOS Git Repository", Last accessed on 9/2017. Source: https://github.com/SuperHouse/esp-open-rtos.

[73] Amazon. "AWS Cloud SDK for Python Documentation Website", Last accessed on 9/2017. Source: https://boto3.readthedocs.io/en/latest/index.html.

[74] Amazon. "AWS IoT SDK for Python Git Repository", Last accessed on 9/2017. Source: https://github.com/aws/aws-iot-device-sdk-python.

A. mHouse Framework RESTful APIs

The mHouse RESTful API is a JSON over REST API to access all the relevant resources of the mHouse Framework. Every resource is represented with JSON and each URI follows the REST specifications and, in some special cases, some queries can be made to filter the results.

A.1 Cloud Platform HTTP RESTful API

All the URIs that belong to the Online Cloud HTTP RESTful API start with **api**/, so for example, if we want to access the houses list for the current logged in user, we must send a GET request to the URL `http://mhouseframework.eu-west-1.elasticbeanstalk.com/api/houses/`. To work with the API, an user must be authorized with the Basic Auth authorization process, which requires the user email (as the username) and a password.

Note: as stated above, every URI that follows must be preceded with **api**/.

A.1.1 General Possible Error Response Codes and Bodies

- Every error response have a description body that can be one of the Error Formats - check section B.
- For every request to the Cloud API the client must provide a valid Basic Authorization header (based on the user email and password) and a valid X-CSRFToken header with the CSRF token provided from a cookie (this cookie can be setted with a GET or HEAD request to a not authentication protected URL like /login). If this informations are not provided or if the use is unauthorized to perform the request, a 403 FORBIDDEN error response with a detail message will be returned.
- If the request method is not allowed for the requested resource a 405 METHOD NOT ALLOWED error response with a detail message will be returned.
- If the “Content-Type” header is different from “application/json”, a 415 UNSUPPORTED MEDIA TYPE error response with a detail message will be returned.
- If some other unacceptable header is included in the request, a 406 NOT ACCEPTABLE error response with a detail message will be returned.
- If the request is being made to a resource that doesn’t exist on the database or to a resource which the user doesn’t have access to (ex: request a House that belongs to another user), a 404 NOT FOUND error response with a detail message will be returned.
- To other bad formatted requests 400 BAD REQUEST error responses with the respective detail message bodies will be returned.
- If something is wrong with the Cloud Server, a 500 INTERNAL SERVER ERROR is also possible as an error response.

A.1.2 Configuration Related URIs

- **URI:** /configs/
  - **Method:** GET
    - **Action:** Get all the Configurations (both core and custom) for the authenticated user.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with all the Configurations.

- **URI:** /configs/device_types/
  - **Method:** GET
    - **Action:** Get all the Device Types (both core and custom) for the authenticated user.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with all the Device Types.
  - **Method:** POST
    - **Action:** Add (create) a new Device Type.
    - **Request Body:** JSON describing the new Device Type
      - Required fields:
      - **name** - The name of the new Device Type
      - **properties** - A non-empty list of Property Type IDs that identify the Device Type properties.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with all the Device Types.
      - 400 BAD REQUEST - Something is wrong with the request.

- **URI:** /configs/device_types/<device_type_id>/
  - **Method:** GET
    - **Action:** Get the Device Type identified with the ID <device_type_id>.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with the Device Type identified with ID <device_type_id>.
      - 404 NOT FOUND - If the requested Device Type doesn't exist for the authenticated user.
  - **Method:** DELETE
    - **Action:** Delete the Device Type identified with the ID <device_type_id>.
    - **Possible Response Codes and Body:**
      - 204 NO CONTENT - Empty Response confirming the Device Type deletion.
      - 404 NOT FOUND - If the requested Device Type doesn't exist for the authenticated user.

- **URI:** /configs/property_types/
  - **Method:** GET

    - **Action:** Get all the Property Types (both core and custom) for the authenticated user.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with all the Property Types.
  - **Method:** POST
    - **Action:** Add (create) a new Property Type.
    - **Request Body:** JSON describing the new Property Type
      - Required fields:
      - **name** - The name of the new Property Type
      - **value_type_class** - "SCALAR" / "ENUM".
      - **value_type_id** - The ID of the Value Type to which the new Property Type will be referring. This ID must exist on the Value Types subset defined by the field **value_type_class**.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with all the Property Types.
      - 400 BAD REQUEST - Something is wrong with the request.

- **URI:** /configs/property_types/<property_type_id>/
  - **Method:** GET
    - **Action:** Get the Property Type identified with the ID <property_type_id>.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with the Property Type identified with ID <property_type_id>.
      - 404 NOT FOUND - If the requested Property Type doesn't exist for the authenticated user.
  - **Method:** DELETE
    - **Action:** Delete the Property Type identified with the ID <property_type_id>.
    - **Possible Response Codes and Body:**
      - 204 NO CONTENT - Empty Response confirming the Property Type deletion.
      - 404 NOT FOUND - If the requested Property Type doesn't exist for the authenticated user.

- **URI:** /configs/scalars/
  - **Method:** GET
    - **Action:** Get all the Scalar Value Types (both core and custom) for the authenticated user.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with all the Scalar Value Types.
  - **Method:** POST
    - **Action:** Add (create) a new Scalar Value Type.
    - **Request Body:** JSON describing the new Scalar Value Type
      - Required fields:

        - **name** - The name of the new Scalar Value Type
        - **units** - “SCALAR” or “ENUM”.
        - **default_value** - The value by default for the Scalar Value Type. This value must be between **min_value** and **max_value**.
        - **min_value** - The minimum value that the Scalar Value Type can take.
        - **max_value** - The maximum value that the Scalar Value Type can take.
        - **step** - The step value for the Scalar Value Type, i.e. the minimum increment value that the Scalar Value Type can take.
    - **Possible Response Codes and Body:**
        - 200 OK - JSON object with all the Scalar Value Types.
        - 400 BAD REQUEST - Something is wrong with the request.

- **URI:** /configs/scalars/<scalar_value_type_id>/
  - **Method:** GET
    - **Action:** Get the Scalar Value Type identified with the ID <scalar_value_type_id>.
    - **Possible Response Codes and Body:**
        - 200 OK - JSON object with the Scalar Value Type identified with ID <scalar_value_type_id>.
        - 404 NOT FOUND - If the requested Scalar Value Type doesn’t exist for the authenticated user.
  - **Method:** DELETE
    - **Action:** Delete the Scalar Value Type identified with the ID <scalar_value_type_id>.
    - **Possible Response Codes and Body:**
        - 204 NO CONTENT - Empty Response confirming the Scalar Value Type deletion.
        - 404 NOT FOUND - If the requested Scalar Value Type doesn’t exist for the authenticated user.

- **URI:** /configs/enums/
  - **Method:** GET
    - **Action:** Get all the Enumerated Value Types (both core and custom) for the authenticated user.
    - **Possible Response Codes and Body:**
        - 200 OK - JSON object with all the Enumerated Value Types.
  - **Method:** POST
    - **Action:** Add (create) a new Enumerated Value Type.
    - **Request Body:** JSON describing the new Enumerated Value Type
        - Required fields:
        - **name** - The name of the new Enumerated Value Type
        - **default_value** - A Choice ID that indicates the value by default for the new Enumerated Value Type. This value (ID) must be present on the **choices** field list.

        - **choices** - A non-empty list of Choice IDs that identify the Enumerated Value Type possible choices.
    - **Possible Response Codes and Body:**
        - 200 OK - JSON object with all the Enumerated Value Types.
        - 400 BAD REQUEST - Something is wrong with the request.

- **URI:** /configs/enums/<enum_value_type_id>/
  - **Method:** GET
    - **Action:** Get the Enumerated Value Type identified with the ID <enum_value_type_id>.
    - **Possible Response Codes and Body:**
        - 200 OK - JSON object with the Enumerated Value Type identified with ID <enum_value_type_id>.
        - 404 NOT FOUND - If the requested Enumerated Value Type doesn't exist for the authenticated user.
  - **Method:** DELETE
    - **Action:** Delete the Enumerated Value Type identified with the ID <enum_value_type_id>.
    - **Possible Response Codes and Body:**
        - 204 NO CONTENT - Empty Response confirming the Enumerated Value Type deletion.
        - 404 NOT FOUND - If the requested Enumerated Value Type doesn't exist for the authenticated user.

- **URI:** /configs/choices/
  - **Method:** GET
    - **Action:** Get all the Choices (both core and custom) for the authenticated user.
    - **Possible Response Codes and Body:**
        - 200 OK - JSON object with all the Choices.
  - **Method:** POST
    - **Action:** Add (create) a new Choice.
    - **Request Body:** JSON describing the new Choice
        - Required fields:
        - **name** - The name of the new Choice.
        - **value** - A value that can be intrinsically related to the physical action/thing.
    - **Possible Response Codes and Body:**
        - 200 OK - JSON object with all the Choices.
        - 400 BAD REQUEST - Something is wrong with the request.

- **URI:** /configs/choices/<choice_id>/
  - **Method:** GET
    - **Action:** Get the Choice identified with the ID <choice_id>.

        - **Possible Response Codes and Body:**
            - 200 OK - JSON object with the Choice identified with ID <choice_id>.
            - 404 NOT FOUND - If the requested Choice doesn't exist for the authenticated user.
    - **Method:** DELETE
        - **Action:** Delete the Choice identified with the ID <choice_id>.
        - **Possible Response Codes and Body:**
            - 204 NO CONTENT - Empty Response confirming the Choice deletion.
            - 404 NOT FOUND - If the requested Choice doesn't exist for the authenticated user.

Note: Each user can only individually get/delete his/her own custom Custom Configurations. For core ones an unauthorized error will be presented and for other users custom Configurations a 404 error will be presented.

A.1.3 Space Related URIs

- **URI:** /houses/
    - **Method:** GET
        - **Action:** Get all the Houses for the authenticated user.
        - **Possible Response Codes and Body:**
            - 200 OK - JSON object with all the Houses.
    - **Method:** POST
        - **Action:** Add (create) a new House.
        - **Request Body:** JSON describing the new House
            - Required fields:
            - **name** - The name of the new House. This field must be unique for the user.
            - **server** - The ID of the Server (Home Server) that is running on the new House. This field can be null. This field must be unique for each House, i.e., the House-Server relation is a One-to-One relation.
        - **Possible Response Codes and Body:**
            - 200 OK - JSON object with all the Houses.
            - 400 BAD REQUEST - Something is wrong with the request.
- **URI:** /houses/<house_id>/
    - **Method:** GET
        - **Action:** Get the House identified with the ID <house_id>.
        - **Possible Response Codes and Body:**
            - 200 OK - JSON object with the House identified with ID <house_id>.
            - 404 NOT FOUND - If the requested House doesn't exist for the authenticated user.

- **Method:** PUT/PATCH
  - **Action:** Update the House identified with the ID <house_id>.
  - **Request Body:** JSON with the House updated fields.
    - Required fields (for PUT method only): see request body for **POST /houses/**.
  - **Possible Response Codes and Body:**
    - 200 OK - JSON object with the House identified with ID <house_id>.
    - 404 NOT FOUND - If the requested House doesn't exist for the authenticated user.
- **Method:** DELETE
  - **Action:** Delete the House identified with the ID <house_id>.
  - **Possible Response Codes and Body:**
    - 204 NO CONTENT - Empty Response confirming the House deletion.
    - 404 NOT FOUND - If the requested House doesn't exist for the authenticated user.

- **URI:** /areas/ [? house=<house_id>]
  - **Method:** GET
    - **Action:** Get all the Areas for the authenticated user.
    - **Possible query filters:**
      - **house** - Get only the Areas that belong to the House with ID <house_id>.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with all the Areas.
  - **Method:** POST
    - **Action:** Add (create) a new Area.
    - **Request Body:** JSON describing the new Area
      - Required fields:
      - **name** - The name of the new Area. This field must be unique for the related House.
      - **house** - The ID of the House to which the new area belongs.
      - **order** - The order of the area, i.e. a representative number that works to order the areas among themselves. For example, if each area represents a house floor, the order number can be the floor number, i.e. order 0 for the ground floor, 1 for the first floor, -1 for the underground floor, etc.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with all the Areas.
      - 400 BAD REQUEST - Something is wrong with the request.

- **URI:** /areas/<area_id>/
  - **Method:** GET
    - **Action:** Get the Area identified with the ID <area_id>.

        - **Possible Response Codes and Body:**
            - 200 OK - JSON object with the Area identified with ID <area_id>.
            - 404 NOT FOUND - If the requested Area doesn't exist for the authenticated user.
    - **Method:** PUT/PATCH
        - **Action:** Update the Area identified with the ID <area_id>.
        - **Request Body:** JSON with the Area updated fields.
            - Required fields (for PUT method only): see request body for **POST /areas/**.
        - **Possible Response Codes and Body:**
            - 200 OK - JSON object with the Area identified with ID <area_id>.
            - 404 NOT FOUND - If the requested Area doesn't exist for the authenticated user.
    - **Method:** DELETE
        - **Action:** Delete the Area identified with the ID <area_id>.
        - **Possible Response Codes and Body:**
            - 204 NO CONTENT - Empty Response confirming the Area deletion.
            - 404 NOT FOUND - If the requested Area doesn't exist for the authenticated user.

- **URI:** /divisions/ [? house=<house_id>][&] [area=<area_id>]
    - **Method:** GET
        - **Action:** Get all the Divisions for the authenticated user.
        - **Possible query filters:**
            - **house** - Get only the Divisions that belong to the House with ID <house_id>.
            - **areas** - Get only the Divisions that belong to the Area with ID <area_id>.
        - **Possible Response Codes and Body:**
            - 200 OK - JSON object with all the Divisions.
    - **Method:** POST
        - **Action:** Add (create) a new Division.
        - **Request Body:** JSON describing the new Division
            - Required fields:
            - **name** - The name of the new Division. This field must be unique for the related Area.
            - **area** - The ID of the Area to which the new division belongs.
        - **Possible Response Codes and Body:**
            - 200 OK - JSON object with all the Divisions.
            - 400 BAD REQUEST - Something is wrong with the request.

- **URI:** /divisions/<division_id>/
    - **Method:** GET
        - **Action:** Get the Division identified with the ID <division_id>.
        - **Possible Response Codes and Body:**
            - 200 OK - JSON object with the Division identified with ID <division_id>.

        - 404 NOT FOUND - If the requested Division doesn't exist for the authenticated user.
    - **Method:** PUT/PATCH
        - **Action:** Update the Division identified with the ID <division_id>.
        - **Request Body:** JSON with the Division updated fields.
            - Required fields (for PUT method only): see request body for **POST /divisions/**.
        - **Possible Response Codes and Body:**
            - 200 OK - JSON object with the Division identified with ID <division_id>.
            - 404 NOT FOUND - If the requested Division doesn't exist for the authenticated user.
    - **Method:** DELETE
        - **Action:** Delete the Division identified with the ID <division_id>.
        - **Possible Response Codes and Body:**
            - 204 NO CONTENT - Empty Response confirming the Division deletion.
            - 404 NOT FOUND - If the requested Division doesn't exist for the authenticated user.

- **URI:** /divisions/<division_id>/devices/
    - **Method:** GET
        - **Action:** Get all the Devices present on the Division identified with the ID <division_id>.
        - **Possible Response Codes and Body:**
            - 200 OK - JSON object with the Devices list.
            - 404 NOT FOUND - If the requested Division doesn't exist for the authenticated user.

A.1.4 Service Related URIs

- **URI:** /services/
    - **Method:** GET
        - **Action:** Get all the Services for the authenticated user.
        - **Possible Response Codes and Body:**
            - 200 OK - JSON object with all the Services.
    - **Method:** POST
        - **Action:** Add (create) a new Service for the authenticated user.
        - **Request Body:** JSON describing the new Service
            - Required fields:
            - **name** - The name of the new Service. This field must be unique for the user.
            - Optional fields:
            - **core_service_ref** - The ID of a Core Service that the user wants to use as a "own"/custom service. If this field is provided and valid, the name of the new

service will always be the name of the Core Service referenced by this field, disregarding the **name** field.

    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with all the Services of the authenticated user.
      - 400 BAD REQUEST - Something is wrong with the request.

- **URI:** /services/<service_id>/
  - **Method:** GET
    - **Action:** Get the Service identified with the ID <service_id>.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with the Service identified with ID <service_id>.
      - 404 NOT FOUND - If the requested Service doesn't exist for the authenticated user.
  - **Method:** PUT/PATCH
    - **Action:** Update the Service identified with the ID <service_id>.
    - **Request Body:** JSON with the Service updated fields.
      - Required fields (for PUT method only): see request body for **POST /services/**.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with the Service identified with ID <service_id>.
      - 404 NOT FOUND - If the requested Service doesn't exist for the authenticated user.
  - **Method:** DELETE
    - **Action:** Delete the Service identified with the ID <service_id>.
    - **Possible Response Codes and Body:**
      - 204 NO CONTENT - Empty Response confirming the Service deletion.
      - 404 NOT FOUND - If the requested Service doesn't exist for the authenticated user.

- **URI:** /services/<service_id>/devices/
  - **Method:** GET
    - **Action:** Get all the Devices present on the Service identified with the ID <service_id>.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with the Devices list.
      - 404 NOT FOUND - If the requested Service doesn't exist for the authenticated user.

A.1.5 Servers and Devices Related URIs

- **URI:** /servers/
  - **Method:** GET

    - **Action:** Get all the Servers for the authenticated user.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with all the Servers.
  - **Method:** POST
    - **Action:** Add (create) a new Server for the authenticated user.
    - **Request Body:** JSON describing the new Server
      - Required fields:
      - **name** - The name of the new Server.
      - **proxy_address** - The public IP address of the "Proxy" subcomponent of the new Server.
      - **proxy_port** - The HTTP port used by the "Proxy" subcomponent on the new Server.
      - **coap_address** - The local IP address of the "CoAP Server" subcomponent of the new Server. This field must be unique for the user.
      - **coap_port** - The CoAP port used by the "CoAP Server" subcomponent on the new Server.
      - **timeout** - The time, in seconds, that defines the rate at which the Home Server sends an "alive" message to the mHouse Cloud Platform.
      - **multicast** - A boolean stating if the new Server is using a multicast address for CoAP communication or not.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with all the Servers of the authenticated user.
      - 400 BAD REQUEST - Something is wrong with the request.

- **URI:** /servers/<server_id>/
  - **Method:** GET
    - **Action:** Get the Server identified with the ID <server_id>.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with the Server identified with ID <server_id>.
      - 404 NOT FOUND - If the requested Server doesn't exist for the authenticated user.
  - **Method:** PUT/PATCH
    - **Action:** Update the Server identified with the ID <server_id>.
    - **Optional Query Setters:**
      - **fromserver** - Boolean stating if the request is being made from the Home Server or not. Requests from the Home Server must be like **PUT/PATCH /servers/<server_id>/?fromserver=True**. Note that all requests whithout **fromserver=True** will issue a subsequent request from the online cloud platform to the Home Server itself, so responses for this kind of requests may

take a while to arrive, with the maximum timeout of 10 seconds.

    * **Request Body:** JSON with the Server updated fields.
        · Required fields (for PUT method only): see request body for **POST /servers/**.
    * **Possible Response Codes and Body:**
        · 200 OK - JSON object with the Server identified with ID <server_id>.
        · 404 NOT FOUND - If the requested Server doesn't exist for the authenticated user.
        · 504 GATEWAY TIMEOUT - If the Home Server is down.
  - **Method:** DELETE
    * **Action:** Delete the Server identified with the ID <server_id>. The Server must not be running to be able to delete it.
    * **Possible Response Codes and Body:**
        · 204 NO CONTENT - Empty Response confirming the Server deletion.
        · 403 FORBIDDEN - If the Server is running.
        · 404 NOT FOUND - If the requested Server doesn't exist for the authenticated user.

- **URI:** /servers/<server_id>/devices/
  - **Method:** GET
    * **Action:** Get all the Devices associated to the Server identified with the ID <server_id>.
    * **Possible Response Codes and Body:**
        · 200 OK - JSON object with the Devices list.
        · 404 NOT FOUND - If the requested Server doesn't exist for the authenticated user.

- **URI:** /servers/<server_id>/state/
  - **Method:** GET
    * **Action:** Get the current State of the Server identified with the ID <server_id>.
    * **Possible Response Codes and Body:**
        · 200 OK - JSON object with the Server state. This is a JSON like "status": "down/running"
        · 404 NOT FOUND - If the requested Server doesn't exist for the authenticated user.

- **URI:** /devices/ [?] [server=<server_id>] [&] [division=<division_id>] [&] [type=<device_type_id>][&] [services=<services_ids_list>]
  - **Method:** GET
    * **Action:** Get all the Devices for the authenticated user.
    * **Possible Query Filters:**
        · **server** - The ID of the Server from which the results must be retrieved.

        - **division** - The ID of the Division from which the results must be retrieved.
        - **type** - The ID of the Device Type of the results that must be retrieved.
        - **services** - A non-empty list of Service IDs from which the results must be retrieved.
    - **Possible Response Codes and Body:**
        - 200 OK - JSON object with all the Devices.
  - **Method:** POST
    - **Action:** Add (create) a new Device for the authenticated user.
    - **Request Body:** JSON describing the new Device
        - Required fields:
        - **name** - The name of the new Device.
        - **address** - The IP address of the new Device. This field must be unique for the user. This field must be unique for each Server.
        - **device_type** - The ID of the Device Type that is the type of the new Device.
        - **server** - The ID of the Server to which the new Device is connected.
        - **local_id** - The ID of the new Device on the Server. This ID is local to each Server.
        - **timeout** - The maximum time interval, in seconds, within which the device must update its state. If a Device don't update its state within this interval, it is considered to be offline (down).
        - Optional fields:
        - **division** - The ID of the Division to which the new Device will belong.
        - **services** - A list of Service IDs to which the new Device will belong.
    - **Possible Response Codes and Body:**
        - 200 OK - JSON object with all the Devices of the authenticated user.
        - 400 BAD REQUEST - Something is wrong with the request.

- **URI:** /devices/<device_id>/
  - **Method:** GET
    - **Action:** Get the Device identified with the ID <device_id>.
    - **Possible Response Codes and Body:**
        - 200 OK - JSON object with the Device identified with ID <device_id>.
        - 404 NOT FOUND - If the requested Device doesn't exist for the authenticated user.
  - **Method:** PUT/PATCH
    - **Action:** Update the Device identified with the ID <device_id>.
    - **Optional Query Setters:**
        - **fromserver** - Boolean stating if the request is being made from the Home Server or not. Requests from the Home Server must be like **PUT/PATCH /servers/<server_id>/?fromserver=True**. Note that all requests

whithout **fromserver=True** will issue a subsequent request from the online cloud platform to the Home Server itself, so responses for kind of requests may take a while to arrive, with the maximum timeout of 10 seconds.

    * **Request Body:** JSON with the Device updated fields.
        · Required fields (for PUT method only): see request body for **POST /devices/**.
    * **Possible Response Codes and Body:**
        · 200 OK - JSON object with the Device identified with ID <device_id>.
        · 404 NOT FOUND - If the requested Device doesn't exist for the authenticated user.
        · 504 GATEWAY TIMEOUT - If the Home Server is down.
  - **Method:** DELETE
    * **Action:** Delete the Device identified with the ID <device_id>. The Device must not be running to be able to delete it.
    * **Possible Response Codes and Body:**
        · 204 NO CONTENT - Empty Response confirming the Device deletion.
        · 403 FORBIDDEN - If the Device is running.
        · 404 NOT FOUND - If the requested Device doesn't exist for the authenticated user.

- **URI:** /devices/<device_id>/state/
  - **Method:** GET
    * **Action:** Get the current state of the Device identified with the ID <device_id>.
    * **Possible Response Codes and Body:**
        · 200 OK - JSON object with the Device state. Check section B.
        · 404 NOT FOUND - If the requested Device doesn't exist for the authenticated user.
  - **Method:** PUT/PATCH
    * **Action:** Update the state of the Device identified with the ID<device_id>.
    * **Optional Query Setters:**
        · **fromserver** - Boolean stating if the request is being made from the Home Server or not. Requests from the Home Server must be like **PUT/PATCH /servers/<server_id>/?fromserver=True**. Note that all requests whithout **fromserver=True** will issue a subsequent request from the online cloud platform to the Home Server itself, so responses for kind of requests may take a while to arrive, with the maximum timeout of 10 seconds.
    * **Request Body:** JSON with the Device state fields.
        · Required fields (for PUT method only): see State Formats (check section B).
    * **Possible Response Codes and Body:**
        · 200 OK - JSON object with the wanted and current states of the Device identified

with ID <device_id>. See State Formats (check section B).
      - 404 NOT FOUND - If the requested Device doesn't exist for the authenticated user.
      - 502 BAD GATEWAY - If the Device is not responding.
      - 504 GATEWAY TIMEOUT - If the Home Server is down.

- **URI:** /devices/<device_id>/history/ [?] [start=<start_date>] [&] [end=<end_date>] [&] [id=<property_change_id>] [&] [source=<source>] [&] [limit=<limit_number_of_results>] [&] [<property_name>=<property_value>] [...]
  - **Method:** GET
    - **Action:** Get the History of changes performed on the properties of the Device identified with the ID <device_id>.
    - **Possible Query Filters:**
      - **start** - The older date that the results can have.
      - **end** - The newer date that the results can have. Both this field and start field must have the format YYYY-MM-DD[T HH:MM:ss[.uuuuuu]][TZ].
      - **id** - The specific ID of the property change we want to retrieve.
      - **source** - The source of the property change. This field can be DEVICE / CLOUD.
      - **limit** - The maximum number of results that we want to retrieve. If this field is inexistent or null, the whole history will be returned.
      - **<property_name>** - The specific value of a Device Property which we want to filter the results by. This field can appear multiple times on the query, so many as the number of properties of the Device.
    - **Possible Response Codes and Body:**
      - 200 OK - JSON object with all the Devices.

A.1.6 History Related URIs

- **URI:** /history/ [?] [start=<start_date>] [&] [end=<end_date>] [&] [id=<action_id>] [&] [action=<action_type>] [&] [instance_class=<instance_class>] [&] [instance_id=<instance_id>] [&] [limit=<limit_number_of_results>]
  - **Method:** GET
    - **Action:** Get the History of actions performed.
    - **Possible Query Filters:**
      - **start** - The older date that the results can have.
      - **end** - The newer date that the results can have. Both this field and start field must have the format YYYY-MM-DD[T HH:MM:ss[.uuuuuu]][TZ].
      - **id** - The specific ID of the action we want to retrieve.
      - **action** - The type of actions. This field can be CREATE / UPDATE / READ / DELETE / ERROR / STAT_UP / STAT_DOWN.
      - **instance_class** - The instance class where the action was performed. This field

can be HOUSE / AREA / DIVISION / SERVER / DEVICE / SERVICE.

- **instance_id** - The ID of the instance where the action was performed.
- **limit** - The maximum number of results that we want to retrieve. If this field is inexistent or null, the whole history will be returned.

* **Possible Response Codes and Body:**
  - 200 OK - JSON object with the Actions History.

A.2 Home Server HTTP and CoAP RESTful APIs

These APIs are RESTful APIs over both HTTP or CoAP which can be accessed on the Home Server. The HTTP one is mainly used by the Online Cloud to access resources on the Home Server but it is accessible to any client which wants to access the resources and supports the HTTP protocol. The CoAP one is supposed to be used by any Endpoint Device which wants to access resources on the Home Server and that uses the CoAP protocol.

Note: The HTTP API is made available with the HTTP-CoAP Proxy which runs in the Home Server. Thus, note that every call to this API will issue a semantic similar call to the CoAP server using the CoAP protocol and translating the corresponding response from CoAP to HTTP. Note also that all request methods and response codes will be described together for both the HTTP and CoAP APIs. The distinction between the APIs should be inherently understood when reading this documentation, e.g. if a presented possible response code is described as being 200 OK/2.05 CONTENT, it should be understood that for the HTTP API the possible response code is 200 OK and for the CoAP API the possible response code is 2.05 CONTENT. Some specific requests and/or responses are only accessible from one of the APIs, and in that situations indicative notes are properly provided.

A.2.1 General Possible Error Response Codes and Bodies

- Every error response have a description body that can be one of the Error Formats - check section B.
- If the request method is not allowed for the requested resource a 405/4.05 METHOD NOT ALLOWED error response with an error code and message body will be returned.
- If the “Content-Type” header is different from “application/json”, a 415 UNSUPPORTED MEDIA TYPE/4.15 UNSUPPORTED CONTENT FORMAT error response with an error code and message body will be returned.
- If some other unacceptable header is included in the request, a 406/4.06 NOT ACCEPTABLE error response with an error code and message body will be returned.
- If the request is being made to a resource that doesn’t exist on the Home Server a 404/4.04 NOT FOUND error response with an error code and message body will be returned.
- To other bad formatted requests 400/4.00 BAD REQUEST error responses with the respective error codes and message bodies will be returned.

- If the CoAP Server is down at the time of the request, a 504 GATEWAY TIMEOUT error response with an error code and message body will be returned.
- If something is wrong with the Home Server, a 500/5.00 INTERNAL SERVER ERROR is also possible as an error response.

- **URI:** / or /info
  - **Method:** GET
    - **Action:** Get all the general informations about the Home Server.
    - **Possible Response Codes and Body:**
      - 200 OK/2.05 CONTENT - JSON object with all the information. See Home Server description (check Appendix B).
  - **Method:** PUT
    - **Action:** Update the general informations about the Home Server.
    - **Request Body:** JSON describing the new Home Server informations.
      - Required Fields:
      - **name** - The new name for the Home Server.
    - **Possible Response Codes and Body:**
      - 200 OK/2.04 CHANGED - JSON object with all the information. See Home Server description (check Appendix B).
      - 400/4.00 BAD REQUEST - Something is wrong with the request.

- **URI:** /services
  - **Method:** GET
    - **Action:** Get all the Services available on the Home Server.
    - **Possible Response Codes and Body:**
      - 200 OK/2.05 CONTENT - JSON object with all the Services available.
  - **Method:** PUT
    - **Action:** Update the Services available on the Home Server.
    - **Request Body:** JSON describing the list of Services that should be available on the Home Server.
      - Required Fields:
      - **SERVICES** - A list of JSON objects each one describing a Service. See Service description (check Appendix B).
    - **Possible Response Codes and Body:**
      - 200 OK/2.04 CHANGED - JSON object with all the Services available.
      - 400/4.00 BAD REQUEST - Something is wrong with the request.

- **URI:** /configs [? type=<configuration_type>]
  - **Method:** GET
    - **Action:** Get all the Configurations available on the Home Server.

    - **Possible Response Codes and Body:**
      - 200 OK/2.05 CONTENT - JSON object with all the Configurations available. See Configurations description (check Appendix B).
  - **Method:** PUT
    - **Action:** Update the Configurations available on the Home Server.
    - **Required Query Setters:**
      - **type** - The type of Configurations that is being updated. This field can be DEVICE_TYPES / PROPERTY_TYPES / SCALAR_TYPES / ENUM_TYPES. Ex: to update the Device Types available on the Home Server, the request to be made must be something like **PUT /configs?type=DEVICE_TYPES**.
    - **Request Body:** JSON describing the new Home Server Configurations.
      - Required Fields:
      - **DEVICE_TYPES** / **PROPERTY_TYPES** / **SCALAR_TYPES** / **ENUM_TYPES** - A list of JSON objects each one describing the configuration being updated, taking into account the type of configurations being updated. See Configurations description (check Appendix B).
    - **Possible Response Codes and Body:**
      - 200 OK/2.04 CHANGED - JSON object with all the Configurations available.
      - 400/4.00 BAD REQUEST - Something is wrong with the request.

- **URI:** /devices
  - **Method:** GET
    - **Action:** Get all the Devices registered on the Home Server.
    - **Possible Response Codes and Body:**
      - 200 OK/2.05 CONTENT - JSON object with all the Home Server Devices.
  - **Method (only for the CoAP API):** POST
    - **Action:** Add (create/register) a new Device on the Home Server.
    - **Request Body:** JSON describing the new Devices.
      - Required fields:
      - **name** - The name of the new Device.
      - **device_type** - The ID of the Device Type that is the type of the new Device.
      - **timeout** - The maximum time interval, in seconds, within which the device must update its state. If a Device doesn't update its state within this interval, it is considered to be offline (down).
      - **services** - The list of Service IDs to which the Device belongs.
    - **Possible Response Codes and Body:**
      - 2.01 CREATED - JSON object with the new registered Device. See Device description (check Appendix B).
      - 4.00 BAD REQUEST - Something is wrong with the request.

- **URI:** /devices/<device_id>
  - **Method:** GET
    - **Action:** Get the Device identified with the local ID <device_id> from the Home Server.
    - **Possible Response Codes and Body:**
      - 200 OK/2.05 CONTENT - JSON object with Device informations.
  - **Method:** PUT
    - **Action:** Update the Device identified with the local ID <device_id> on the Home Server.
    - **Request Body:** JSON describing the updated Device
      - Required Fields:
      - **name** - The new name for the Device.
      - Only for self updates using the CoAP API (the device itself wants to update its description):
      - **device_type** - The new ID of the Device Type that is the type of the Device updated.
      - **timeout** - The new maximum time interval, in seconds, within which the device must update its state. If a Device doesn't update its state within this interval, it is considered to be offline (down).
      - **services** - The new list of Service IDs to which the Device belongs.
    - **Possible Response Codes and Body:**
      - 200 OK/2.04 CHANGED - JSON object with all the updated Device informations.
      - 400/4.00 BAD REQUEST - Something is wrong with the request.
    - **Method (only for the CoAP API):** DELETE
      - **Action:** Delete the Device identified with the ID <device_id>. This method is only available from the Device itself, i.e. only the Endpoint device with ID <device_id> can delete its representation on the Home Server.
      - **Possible Response Codes and Body:**
      - 2.02 DELETED - Empty Response confirming the Device deletion.
      - 4.03 FORBIDDEN - If the request is coming from another Endpoint device.
      - 4.04 NOT FOUND - If the requested Device doesn't exist on the Home Server.

- **URI:** /devices/<device_id>/state
  - **Method:** GET
    - **Action:** Get the State of the Device identified with the local ID <device_id> from the Home Server.
    - **Possible Response Codes and Body:**
      - 200 OK/2.05 CONTENT - JSON object with the Device State. See state formats (check Appendix B).
  - **Method:** PUT
    - **Action:** Update the State of the Device identified with the local ID <device_id> on the

Home Server.

    * **Request Body:** JSON with the Device state fields.
        · Required Fields (for PUT method only): See state formats (check Appendix B).
    * **Possible Response Codes and Body:**
        · 200 OK/2.04 CHANGED - JSON object with all the updated Device State. See state formats (check Appendix B).
        · 400/4.00 BAD REQUEST - Something is wrong with the request.
        · 503 BAD GATEWAY (only for HTTP API) - The Device with local ID <device_id> is not responding and probably is down.

- **URI:** /devices/<device_id>/type
  - **Method:** GET
    * **Action:** Get the Device Type of the Device identified with the local ID <device_id> from the Home Server.
    * **Possible Response Codes and Body:**
        · 200 OK/2.05 CONTENT - JSON object with the Device Type. See Device Types formats (check Appendix B).

- **URI:** /devices/<device_id>/services
  - **Method:** GET
    * **Action:** Get the Services of the Device identified with the local ID <device_id> from the Home Server.
    * **Possible Response Codes and Body:**
        · 200 OK/2.05 CONTENT - JSON object with the Device Services. See Device Service description (check Appendix B).
  - **Method:** PUT
    * **Action:** Update the Services of the Device identified with the local ID <device_id> on the Home Server.
    * **Request Body:** JSON with the list of Device Services.
        · Required Fields:
        · List of Service IDs that the Device should belong to. The IDs provided must exist on the available Services on the Home Server.
    * **Possible Response Codes and Body:**
        · 200 OK/2.04 CHANGED - JSON object with the Device Services.
        · 400/4.00 BAD REQUEST - Something is wrong with the request.

B. mHouse Framework Formats and Descriptions

B.1 Error Formats

- Error Format 1 (Cloud Platform main errors):
  - This format has an HTTP error code (4** or 5**) with the respective error phrase.
  - It also has a body message with more detailed information about the error.
  - **Body Format:**
    - { “detail”: <detailed_error_information> }
- Error Format 2 (Home Server main errors):
  - This format has an HTTP or CoAP error code (4**/5** for HTTP, or 4.**/5.** for CoAP) with the respective error phrase.
  - It also has a body message also with the error code and an error message with more detailed information about the error.
  - **Body Format:**
    - { “err_code”: <error_code>, “err_msg”: <detailed_error_information>>}

Note: With the unique exception of the URI /**api/configs**/, when expected a list like response, for each resource type the global format is {<resource_type>: <list_of_resources>} (Ex: the request GET /api/houses/ expects a response with a list of Houses so the response will be {“houses”: <list_of_houses>}).

B.2 Spaces Description

- House Format:
  - When retrieving a House resource, the global format is:
    - { ”id”: <house_universal_id>>, ”name”: <house_name>, ”server”: <home_server_id>, ”area_count”: <number_of_areas_of_the_house>}
- Area Format:
  - When retrieving an Area resource, the global format is:
    - { ”id”: <area_universal_id>>, ”name”: <area_name>, ”house”: <house_id_where_the_area_belongs>, ”division_count”: <number_of_divisions_of_the_area>}
- Division Format:
  - When retrieving an Division resource, the global format is:
    - { ”id”: <division_universal_id>>, ”name”: <division_name>, ”area”: <area_id_where_the_division_belongs>,

”device_count”: <number_of_devices_of_the_division>}

B.3 Configurations Description

- Device Type Format:
  - When retrieving a Custom Device Type resource, the global format is:
    - { ”id”: <device_type_universal_id>>, ”name”: <device_type_name>, ”properties”: <list_of_property_IDs_that_belong_to_this_device_type>}

  Note: When retrieving the list of Device Types, the global format for the Core Device Types has also another field like {“user”: null}.

  - When retrieving the Type resource of a specific Device on the Home Server, the global format is:
    - { ”device_id”: <device_local_id>, ”device_type”: { “properties”: [<JSON_list_of_properties: each element follows the Property Type Format>], “id”: <device_type_id>, “name”: <device_type_name> }}

- Property Type Format:
  - When retrieving a Custom Property Type resource, the global format is:
    - { ”id”: <property_type_universal_id>>, ”name”: <property_type_name>, ”access_mode”: <property_type_access_mode: RO / WO / RW>, ”value_type_class”: <value_type_class_of_the_property_type: SCALAR / ENUM>, ”value_type_id”: <specific_ID_of_the_value_type_of_the_property_type>}

  Note: When retrieving the list of Property Types, the global format for the Core Property Types has also another field like {“user”: null}.

- Scalar Value Type Format:
  - When retrieving a Custom Scalar Value Type resource, the global format is:
    - { ”id”: <scalar_value_type_universal_id>>, ”name”: <scalar_value_type_name>, ”units”: <scalar_value_type_units>, ”default_value”: <default_value_of_the_scalar_value_type>, ”min_value”: <min_value_of_the_scalar_value_type>, ”max_value”: <max_value_of_the_scalar_value_type>, ”step”: <step_value_of_the_scalar_value_type>}

  Note: When retrieving the list of Scalar Value Types, the global format for the Core Scalar Value Types has also another field like {“user”: null}.

- Enumerated Value Type Format:
  - When retrieving a Custom Enumerated Value Type resource, the global format is:
    - { ”id”: <enumerated_value_type_universal_id>>, ”name”: <enumerated_value_type_name>, ”choices”: <list_of_choices_IDs_of_the_enumerated_value_type>, ”default_value”: <default_value_of_the_enumerated_value_type>}

Note: When retrieving the list of Enumerated Value Types, the global format for the Core Enumerated Value Types has also another field like {"user": null}.

- Choices Format:
  - When retrieving a Custom Choice resource, the global format is:
    - { "id": <choice_universal_id>>, "name": <choice_name>, "value": <semantic_related_value_of_the_choice>}

Note: When retrieving the list of Choices, the global format for the Core Choices has also another field like {"user": null}.

B.4 Service Description

- When retrieving a Service resource on the Online Cloud Platform, the global format is:
  - { "id": <service_universal_id>, "name": <service_name>, "core_service_ref": <ID_of_the_core_service_which_this_service_is_based_on / null>, "device_count": <number_of_devices_of_the_service>}
- When retrieving the Services list resource on the Home Server, the global format is "SERVICES": <service_list> , where each element of the <service_list> has the format:
  - { "id": <service_universal_id>, "name": <service_name>, "core_service_ref": <ID_of_the_core_service_which_this_service_is_based_on / null>}
- When retrieving a Device Services resource on the Home Server, the global format is:
  - { "device_id": <device_local_id>, "services": <list_of_device_services_IDs>}

B.5 Home Server Description

- When retrieving a Home Server resource on the Online Cloud Platform, the global format is:
  - { "id": <server_universal_id>, "name": <server_name>, "proxy_address": <proxy_subcomponent_IP_address>, "proxy_port": <proxy_subcomponent_port>, "coap_address": <coap_server_subcomponent_IP_address>, "coap_port": <coap_server_subcomponent_port>, "timeout": <home_server_timeout>, "multicast": <server_multicast_functioning_option: true / false>, "active": <server_active_flag: true / false>, "device_count": <number_of_devices_of_the_server>}
- When retrieving the info resource on the Home Server, the global format is:
  - { "server_id": <server_universal_id>, "name": <server_name>, "proxy_address": <proxy_subcomponent_IP_address>, "proxy_port": <proxy_subcomponent_port>, "coap_address": <coap_server_subcomponent_IP_address>, "coap_port": <coap_server_subcomponent_port>,

”timeout”: <home_server_timeout>, ”multicast”: <server_multicast_functioning_option: true / false>}

B.6 Device Description

- When retrieving a Device resource on the Online Cloud Platform, the global format is:
  - { ”id”: <device_universal_id>, ”local_id”: <home_server_device_local_id>, ”name”: <device_name>, ”address”: <device_IP_address>, ”device_type”: <device_type_ID_for_the_device>, ”timeout”: <device_alive_update_timeout>, ”active”: <device_active_flag: true / false>, ”server”: <device_server_ID>, ”division”: <device_division_ID>, ”services”: [<list_of_services_IDs_to_which_the_device_belongs>]}
- When retrieving a Device resource on the Home Server, the global format is:
  - { ”universal_id”: <device_universal_id>, ”local_id”: <home_server_device_local_id>, ”name”: <device_name>, ”address”: <device_IP_address>, ”port”: <device_CoAP_port>, ”device_type”: <device_type_ID_for_the_device>, ”timeout”: <device_alive_update_timeout>}

B.7 State Formats

- When retrieving a Server State resource on the Cloud Platform, the global format is:
  - If the Server is running:
    - { ”server_id”: <server_universal_id>, ”name”: <server_name>, ”address”: <server_IP_address>, ”port”: <server_CoAP_port, ”multicast”: <server_multicast_functioning_option: true/false>, ”status”: “running”}
  - If the Server is down:
    - { ”status”: “down” }
- When retrieving a Device State resource on the Cloud Platform, the global format is:
  - { ”status”: <device_status: down / running>, ”state”: { <device_property_1_name>: <device_property_1_value>, <device_property_2_name>:<device_property_2_value>, (....) <device_property_N_name>:<device_property_N_value> }, ”device_id”: <device_universal_id> }
- When retrieving a Device State resource on the Home Server, the global format is:
  - { ”wanted_state”: { <device_property_1_name>: <device_property_1_value>, <device_property_2_name>:<device_property_2_value>, (....) <device_property_N_name>:<device_property_N_value> }, ”current_state”: { <device_property_1_name>: <device_property_1_value>, <device_property_2_name>:<device_property_2_value>, (....) <device_property_N_name>:<device_property_N_value> }, ”device_id”: <device_universal_id> }