2015-ENST-0033

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « Informatique et Réseaux »

présentée et soutenue publiquement par

Gérôme BOVET

le 22 juin 2015

Architecture Évolutive et Efficiente du Web des Bâtiments

Directeur de thèse : **Ahmed SERHROUCHNI**
Co-encadrement de la thèse : **Jean HENNEBERT**

TELECOM ParisTech
école de l'Institut Mines-Télécom - membre de ParisTech

46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

T H È S E

Abstract

Due to environmental and financial motivations, the concept of smart buildings has emerged in recent years, striving to optimise building facilities such as heating, air conditioning and lighting from an energy point of view. Such buildings make an intensive use of sensors and actuators to endow intelligence in central management systems. However, technology providers of building automation systems (BASs) have been functioning, for a long time, with dedicated networks, communication protocols and APIs. Internet of Things (IoT) principles are now appearing in buildings as a way to simplify and standardise the network infrastructure around the Internet Protocol (IP). Recent strides in electronics and embedded devices have led to *smart things*, natively able to communicate over IP. These devices are now gently forming new kinds of building networks and are populating our daily life by being present in appliances such as TVs and fridges.

Eventually, a mix of different technologies could even be present in a given building and will result in the impossibility to have a global management. In such cases, building automation systems form small, incompatible and isolated islands unable to talk to each other, inducing issues in terms of integration, interoperability and maintenance. To make a joint management of different systems achievable, research and industry came up with complex middleware relying on service oriented architectures (SOA), originally intended for business applications. Such solutions suffer from relying on tightly-coupled technologies, resulting in inert systems that are difficult to maintain. Internet of Things devices are not yet able to play the role of a middleware solution around these technologies.

On the other hand, the Internet, and especially the Web with its associated technologies is a compelling example of a scalable and homogeneous network independent of hardware and software platforms. The simple and open standards have contributed to make it an ubiquitous framework to build flexible systems available from a broad range of devices. The Web of Things architecture bases its foundation around the aforementioned strengths of the Web by pushing associated technologies such as HTTP down to field devices. More specifically, the REST architectural style is adapted to match the requirements of embedded devices, and becomes therefore the federating concept for composing a global integration platform. Smart things are exposing their functionalities in the form of Application Programming Interfaces (APIs) that can be used by other Web resources to create new composite applications. Relying on open and standard technologies makes the Web of Things an outstanding candidate to homogenise building automation systems.

In this thesis we leverage on the Web of Things and the emerging Web technologies to rally building automation systems and smart things into one integration platform. By drawing upon tools and techniques of both worlds, we define the fundamental building blocks of the *Web of Buildings* as an extension of current Web paradigms and especially the Web of Things. After evaluating the limitations of current middleware with respect to the requirements of IoT and BASs, we propose practical solutions to alleviate the integration of legacy devices and smart things into one shared framework. We particularly focus on enabling efficient development of scalable management applications encompassing novel data-driven approaches. Finally, we come up with an end-to-end, fully Web-based ecosystem including underlying systems that fosters the conception of homogeneous global control applications running on top of legacy building automation systems and smart things.

In contrast to existing research in sensor networks applied to the context of building automation, the main question explored in this thesis is how much of the emerging Web paradigms can be reused to homogenise smart buildings. We further examine the common belief that Web standards are insufficient and inappropriate to build an efficient building automation framework. Experimental results and prototypes are provided to support the hypothesis that homogenisation can be achieved by relying on REST and the semantic Web. Our results further show that the Web is not only a good candidate from an integration point of view, but actually permits to build an autonomous and self-sufficient architecture around lightweight protocols.

We provide a comprehensive, conceptual and empirical investigation of the usage of Web technologies to share information between building automation devices. The contributions of this work are multiple. First, we explore the different layers composing building automation systems and propose for each one a set of building blocks being the essence of the *Web of Buildings* architecture. These layers all rely on Web standards to promote reusability in the form of loosely-coupled RESTful APIs. We put a particular attention on the self-organising aspect so that deployment and maintenance require no particular knowledge, following new in-network cloud approaches. We also investigate how novel data-driven algorithms can be integrated as automation layer in the core of the Web. Second, we demonstrate the feasibility to incorporate legacy building automation systems through transparent *smart gateways*, allowing access to legacy functionalities in a straightforward manner. Third, our investigations show that Web application protocols are not equivalent in terms of energy consumption. Savings can be achieved on smart things by intelligently selecting the most appropriate protocol according to some conditions.

Finally, we hope the work presented in this thesis will serve as inspiration for future research. Connecting both worlds, the Web and building automation systems will very likely open new doors upon unexplored domains such as data-driven algorithms to endow smart buildings and the Web with more intelligence, paving the way for new types of applications.

Acknowledgements

Earning a Ph.D. degree is a rather long journey and it is necessary to be surrounded by committed individuals to reach the end of the road. I would like therefore to thank all the people having contributed to the pleasant conditions which made my work possible during these last years.

My first and deepest gratitude goes to Prof. Jean Hennebert. He is the person who offered me the chance to live this wonderful adventure. He allowed me to explore this interesting topic with an entire liberty. In order to make work even more pleasant, he always found the best way to motivate its staff by establishing a good atmosphere.

I am extremely thankful to Prof. Ahmed Serhrouchni who opened the doors of Télécom ParisTech and always warmly welcomed me during my stays in Paris. I highly appreciated his full confidence throughout my work.

I would also like to thank my awesome colleague Antonio Ridi for all these years of collaboration. Sharing the office with him was a real pleasure. Our funny moments helped us to overcome the most stressful times. From our collaborative efforts emerged very interesting new fields thanks to his expertise in machine learning.

The contributions of this thesis would never have been possible without the participation of many people. First, I would like to thank my other officemate Sébastien Baudin who shared with me his knowledge in software engineering and application servers. I was also lucky to supervise talented students: Gautier Briard, Loïc Folly, Samuel Mertenat and others. They all have been very significant during the implementation of the various components described in this thesis. I can only hope I was able to share with them my knowledge so that they can use it in their future careers.

The projects conducted during this thesis have been done in collaboration with brilliant entrepreneurs and scientists. It was a great pleasure to exchange ideas and visions about the Internet of Things with Novaccess. I wish them the greatest success with their innovative products and solutions. I thank the EPFL LESO team for their cooperation and for bringing me new knowledge in building physics. Very appreciable, they offered me the possibility to use their facilities as playground for some of my contributions.

A special thank goes to Dominique Guinard alias Misterdom who can be considered as the pioneer of the Web of Things and certainly my mentor. It is his works which motivated me to choose this highly exciting domain of research. His guidance and sharp advices were a valuable source of inspiration.

Finally, thanks to all my friends who supported me in rough moments and for being there to cheer with a good beer. Thank you Hélène for the sharp corrections of my prose. I think particularly to my girlfriend Laure for her affection and for giving me a helpful hand at the end of the thesis.

Last but not least, I would like to express my greatest gratitude to my family, and especially to

my mother who always believed in me, motivated me to study and always found the right words.
Thank you.

Contents

Chapter 1

Introduction

Contents

During the last years, the European population has become more and more concerned by the environmental dimension resulting from its behaviour. Apart of being just an ecological trend, the economical interests are nowadays more predominant due to raising energy costs. Relayed at the political level, the reduction of environmental impacts is nowadays a major societal goal, reinforced by the will of the European Community and countries to inject public money in the reduction of energy consumption and the promotion of renewable energy [18]. In developed countries, buildings are a major issue as they are a main source of energy consumption. They represent between 30% and 40% of the total bill in Europe and in the USA, more important than the industry and the transportation. In a building, half of the energy consumption comes from the Heating, Ventilation and Air Conditioning (*HVAC*), followed by lighting and other appliances contributing by 30%, completed by hot water boilers [19]. It is therefore obvious that buildings represent a large potential of energy savings. Many works in the fields of materials science are currently undertaken to reduce the loss of energy induced by insulation in new and renovated buildings [20]. On the other hand, energy efficiency leveraging on a better use of HVAC equipments and lighting by considering local production and storage capacity as well as temporal occupation of rooms is gaining momentum [21, 22]. To summarise, the ultimate goal lies in the optimisation of the ratio between the energy savings and the user comfort. This objective requires increasingly sophisticated *building management systems*. The joint management of heating, lighting or local energy production will be done through real information systems based on a variety of interconnected sensors and actuators, leading to the concept of *Smart Buildings*.

1.1 Problem Statement

Historically, *building management systems* (BMSs) or, these days referred as *building automation systems* (BASs) were composed of vendor-specific electronics controlling only distinct

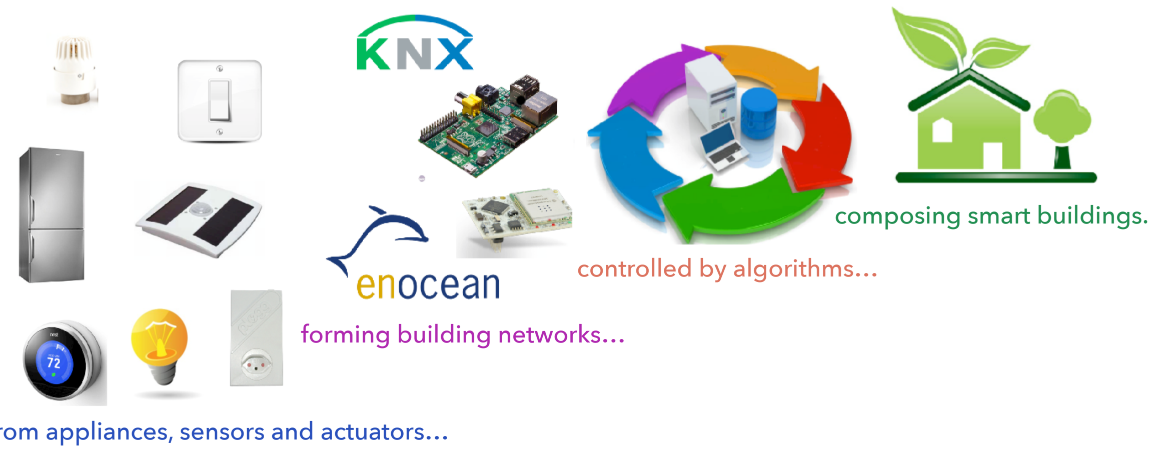

KNX
enocean
72
From appliances, sensors and actuators...
forming building networks...
controlled by algorithms...
composing smart buildings.

Figure 1.1: A couple of stages are required until energy savings and comfort improvement can be obtained in Smart Buildings. Control algorithms rely on functionalities offered by building networks themselves composed of smart things. They range from home appliances to sensors and actuators.

components of the HVAC. Control loops targeting a threshold value according to room thermostats were used to save energy and to keep the comfort at a constant level. With advances made in telecommunications, building automation systems evolved to interconnected networks of sensors and actuators. Nowadays, most automated buildings are working with dedicated building networks like KNX, BACnet, LonWorks or EnOcean [23, 24, 25]. These networks were especially conceived for the purpose of building automation, including all layers of the OSI model, starting from the physical one, up to the application one. Thanks to the miniaturisation of electronics and to the increase of computational power, devices offering native IP connectivity are appearing, leading to so-called *smart things*. Sensor networks based on 6LoWPAN or Wi-Fi are nowadays in competition with classical building networks. The classical approach relies on central *control systems* connected to the network, managing the whole building by implementing some predefined automation rules [26].

It is today common to find mixed technologies in one building, being a consequence of wiring issues or technology obsolescence. This leads to heterogeneous networks, resulting in complex BASs where each network protocol present in the building has to be integrated [27, 28, 29]. As those protocols are working differently with distinct proprietary layers, their integration is costly and induces high maintenance efforts in the case of version change. We can summarise this situation with the observation that there is a lack of standardisation especially at the application level between building automation systems. In the upper layers of BAS, we also observe the emergence of new types of building controls that, on top of the classical rule based controls, are now including stochastic sensor data modelling based on data-driven approaches. While showing higher complexity, such approaches are achieving better results than traditional controls. Standards for integrating such algorithms into BASs are still missing.

At the application level, a simple and easy-to-use protocol on top of different heterogeneous networks is indispensable to form modern building management systems. Such a protocol should support the whole process for composing smart buildings, starting with the integration of field devices like appliances and sensors, up to the decision logic, as shown in Figure 1.1. Finally, the scalability and energy efficiency which are often neglected should be key characteristics of the protocol.

1.2 Research Context

Many research in the fields of ubiquitous and pervasive computing are aiming at the integration of physical objects with the digital world [30, 31, 32, 33, 34]. In an increasingly digitalised world, embedded devices are gaining more popularity by interfering with the daily life. Interconnecting them together, forming sensor networks of physical objects opens new dimensions for environmental monitoring, smart cities and context adaptive homes [35, 36, 37]. Some protocols working at the physical and the network layers tackling the problematic related to constrained devices have been proposed by the research community and the industry. We can mention among others ZigBee, Bluetooth, IEEE 802.15.4 or, more recently, low-power Wi-Fi, 6LoWPAN and RPL [38, 39] [289, 290]. In spite of those technologies helping to compose uniform networks, things still remain isolated from each other at the application layer. This situation requires developers to have expert knowledge of each embedded device platform. As final consequence, integrating smart things into applications is still a challenging task [40, 41].

First attempts trying to facilitate the integration of heterogeneous devices into applications have been inspired by enterprise computing. Interoperability at the application layer is achieved by leveraging on `WS-*` Web Services [42]. Also called "Big Web Services" [43], they rely on two XML formalisms: WSDL for service description and SOAP for service consumption, as well as a set of additional components (WS-Adressing, WS-Security, WS-Discovery among others). The aim of Web Services can be seen as providing what we can call *a loosely-coupled conglomerate of services on smart things*. In a broader point of view, this makes possible to develop new applications composed of widely distributed services. Due to the large overhead of XML, the overly high required CPU power and memory capacity, this approach may not be applied as such to smart things. Some initiatives overcoming this limitation by adapting these services for the real-world led to lighter forms of `WS-*` [44, 40, 45, 46, 41]. Despite those optimisations, implementing the `WS-*` stack on embedded devices remains an ambitious work [47, 43].

The Internet acts as real-live example of a scalable heterogeneous network where various hardware and software platforms coexist. On top of it, the Web contributes with its well-known open standards (e.g. HTTP, HTML, JSON, RDF, etc.) to build a flexible applicative ecosystem. Following this direction, the Web of Things Architecture focuses on a set of scalable and powerful building blocks (i.e. accessibility, findability, sharing and composition layer) while relying on Web technologies [48]. The ultimate goal of the Web of Things (WoT) can be summarised as trying to *ease the development of distributed composite applications across smart things* [49, 50, 51]. Representing services as resources of the Web, which naturally fits with physical objects, the *REST* (Representational State Transfer) [52, 53] architectural style is therefore the key foundation of the Web of Things. Smart things get identifiable by URIs pointing to resources hosted on their embedded Web servers. More than a simple transport protocol, the role of HTTP in this approach is central by becoming a true application layer.

Following this direction, we aim at merging all those technologies in order to propose one common framework dedicated to smart buildings. This framework should offer ways for managing buildings based on a unified application layer working across building networks and IoT devices, relying on open Web technologies. By managing we mean discovering devices, reading and changing their state, storing their states as historical data as well as executing more advanced based on data-driven algorithms.

To achieve such an homogeneity, we propose to rely on the Web of Things to ensure an application-level compatibility among building automation systems. Besides benefiting from the strong interaction style of REST, we suggest to extend the smart thing concept with Web semantics. The semantic Web provides a set of mechanisms allowing data to be shared and reused across applications. This becomes especially important in the context of smart buildings where each technology has its own data representation. Using semantics empowers smart things with *the ability to understand data and to give it a meaning.* In a more broader view, the semantic Web allows uniformly to describe capabilities of smart things, targeting an interoperability between building automation systems.

1.3 Constitutive Hypothesis

In the research context outlined above, we pointed out that emerging network technologies contribute in the heterogeneity of building systems by only barely focusing on interoperability at the application layer. Furthermore, only few existing approaches allow to interconnect applications, especially when it comes to address the requirements of sensor networks composed of constrained devices. Based on the preceding observations, this work is grounded on the following hypotheses:

- The Internet of Things and its related technologies are emerging.
- A standardisation on top of the Internet of Things is inevitable to ensure interoperability.
- Internet of Things devices will penetrate the building automation market with novel sensors and actuators.
- The paradigm of the Web of Things and Web technologies will increasingly be adopted in sensor networks.

These hypotheses will be further detailed and confronted with a literature survey presented in Chapter 2 and Chapter 3. As it will be demonstrated, we have good reasons to believe that theses assumptions will be true in a near future. Considering these, our constitutive hypothesis and main research question in this thesis is the following:

Can building automation systems composed of novel IoT nodes and legacy devices be homogenised around emerging Web technologies and especially the paradigm of the Web of Things?

More research questions bound to this main question are detailed in the next section.

1.4 Research Questions and Methodologies

In addition to the aforementioned assumptions and the constitutive hypothesis, this thesis is driven by the following research questions:

- Are applicative Web protocols compatible with building automation system interaction models?
- Are RESTful APIs descriptive enough to allow automatic service consumption in the context of buildings?
- Can applications be built as in-network clouds and distributed on smart things?
- Are data-driven building management algorithms compatible with a Web approach?
- Do applicative Web protocols used by notification mechanisms differentiate between themselves in terms of energy efficiency and traffic load?
- Can legacy building automation systems such as KNX be seamlessly interfaced with Web-based sensor networks?

These research questions are addressed throughout this thesis by surveying related works and discussing the current shortcomings and limitations of existing solutions. We then propose to overcome these problems not only from a conceptual point of view, but through the development of proof of concepts to validate our proposals. We then test our experimental implementations as far as possible in scenarios reflecting real use cases. Finally, and whenever possible, we confront our results to existing state-of-the-art solutions in order to position our contributions.

1.5 Contributions

In this thesis, we go beyond standardising the application layer for building management systems by proposing an enhanced Web of Things architecture for smart buildings called the *Web of Buildings*. We reconsider the classical architecture of a building automation system to become a distributed Web-based application platform. Our proposition relies on a set of building-blocks to create decoupled and reusable applications. As data-driven building management is gaining momentum, we endow our architecture with dedicated machine-learning-centric blocks bearing future control strategies. Our goal is to provide a common platform for buildings, that developers can use to easily develop management applications while reusing existing services and devices. In the same way as developers build Web 2.0 applications by creating Web mashups of services, they also should be able to design building management systems by associating services offered within the building. Finally, due to the already large scope of this thesis and knowing that standardisation organisms are currently working on solutions, security aspects are not addressed in this work.

Furthermore, the main contributions of this thesis are the followings:

Using Web technologies in building automation: The Web of Things provides a set of techniques to push Web technologies on objects. We rely on this paradigm and adapt it to the context of smart buildings by proposing specific architectural guidelines [1, 2, 3, 4]. We also show a concrete example how Web technologies can be used in daily-life objects [5].

Considering sensor networks as cloud systems: Smart things embed nowadays an appreciable amount of memory and computational power that is generally underexploited. In our approach, we reuse these available resources to migrate services usually residing server-side or in the cloud to become distributed in the building network, forming in-network clouds. Concretely, we are able to provide a naming service [6], to store historical data [7] and to execute advanced algorithms [8] directly in the network without the need of dedicated hardware.

Integrating machine learning in the Web: Building automation based on data-driven algorithms are increasingly used to exploit the large amount of data produced by sensors [17]. As current implementations are running on rather closed and dedicated software, it becomes challenging to integrate them within sensor networks. We therefore propose to consider machine learning, and especially HMMs as Web resources being an inherent part of the Web. Furthermore, we show how runtime-executables can be deployed on smart things [9].

Connecting legacy building automation systems to the Web: Legacy building automation systems such as KNX and EnOcean are installed in a wide variety of buildings. These systems need to be seamlessly integrated within the Web to ensure interoperability with novel sensor networks. We implemented and successfully deployed gateways translating the legacy protocols into reusable RESTful APIs, allowing an integration in Web environments [10, 11, 12, 13].

Energy efficiency of notifications: Notifications are the major communication model in buildings and are therefore the main factor influencing the energy consumption of smart things. While several protocols are available, we analysed their impact on energy consumption for devices producing data [14]. Based on their energy footprint, we are able to optimise their use, which, as a consequence, allows to reduce their footprint [15, 16].

1.6 Thesis Outline

The remainder of this thesis is structured as follows: Chapter 2 presents a high-level survey of related works with emphasis on the emergence of new sensor network technologies and building automation systems.

In Chapter 3 we provide insights on current Web-related paradigms and technologies aiming to push the Web down to daily objects and sensor networks. We also discuss their potential use in smart building applications.

In Chapter 4 we present our Web of Buildings architecture. We expose the required building blocks to successfully turn building automation systems into an homogeneous Web ecosystem. Our proposal includes four blocks addressing different levels of functionalities required by building automation systems. The Field Level provides communication capabilities with sensor and actuator devices. The Automation Level challenges the creating and deployment of automation rules. The Intelligent Adaptation Level outlines how the machine learning training phase can be integrated into the Web. Finally the Management Level contributes to the overall working with a set of general supporting services for a building network.

In Chapter 5 we challenge our proposition of the Web of Buildings architecture from two angles. First, we apply our Web of Buildings layered architecture to integrate the KNX and EnOcean legacy networks into the Web. For this, we present two gateway implementations that were successfully deployed in real environments. Then, we address the energy consumption of smart things and propose a set of mechanisms to limit their footprint. We first apply an intelligent module on data producers that automatically switches to the most economic application protocol. We then suggest to adapt CoAP in order to make multicast notifications possible.

Finally, in Chapter 6 we provide a summary of our contributions. By taking a step back, we also discuss some open challenges and interesting future directions the Web of Buildings will have to address.

Chapter 2

Background and Motivations

Contents

In the last years, installations of building automation systems have democratised, reaching not only business and industry buildings, but also individual homes. Many technologies have also been emerging, leading sometimes to the presence of a mix of technologies in a given installation. For example, we now observe an increasing number of facilities where legacy building automation systems and native IP devices have to coexist. Control applications are also becoming more and more complex, relying on new types of algorithms that are often tuned using live-data from the building. Due to this increasing diversity and complexity, new approaches to federate building networks and to implement control strategies are needed.

In this chapter, we survey and discuss the related work in the field of heterogeneous building automation systems integration. We first overview the last trends in building automation technologies. We then review the existing and emerging building automation systems such as KNX, EnOcean and IoT. We then summarise the classical architectures for elaborating computerised distributed services, including Object-Oriented-Architectures, Service-Oriented-Architectures and Resource-Oriented Architectures. We then review several middleware that are emerging in the context of building automation such as SPINE or oBIX. From this technology overview, we finally conclude that the emergence of these new trends is a strong motivation to propose a consolidated architecture able to homogenise the application layer, which is the main topic of this thesis.

2.1 Emergence of New Technologies in Building Automation Systems

Thanks to the miniaturisation of electronics and the increase of computing power, devices offering native IP connectivity are appearing. Some devices embed sensing and actuation capabilities, fully dedicated to building automation [54]. Building systems may now include *Wireless sensor networks* (WSNs) based on 6LoWPAN or Wi-Fi that are nowadays in competition with classical building networks such as outlined in [55]. Another trend is to give communication capabilities to everyday objects through IPv4 and IPv6 networks, offering new functionalities like machine-to-machine communications [30]. This has led to the emergence of a new paradigm called the *Internet of Things* (IoT) [32, 33, 30]. This research field attempts to find answers about the best ways to connect objects to the Internet. From the network point of view, the IoT covers the first four layers of the OSI model [56, 57].

In the next sections, we first have a look at emerging wireless sensor networks that are triggering new types of applications. We then analyse the impact of using heterogeneous technologies in smart buildings. We finally introduce the new trends in building control that are based on more elaborated algorithms including for example data-driven approaches.

2.1.1 Emerging Wireless Sensor Networks

We here outline the domains composing new wireless sensor networks by providing insights on the network infrastructure and the applications that can be created on top.

New infrastructures

Wireless sensor networks have emerged in the last years by forming autonomous and self-organised networks of sensors often battery powered (also known as *nodes*). These nodes are able to monitor the physical environment (temperature, illumination, humidity, motion, etc.) as well as to interact with the physical world through actuation (light control, blinds, etc.). Forming so-called *mesh networks* where communication is achieved by forwarding data to neighbour nodes (multi-hop), they can be placed geographically where a direct transmission between edge nodes would not be possible. The data directed to outward networks is forwarded to a border router, as illustrated in Figure 2.1. Setting up a border router can be realised, for example, by connecting a node to a USB port on a computer on which routing functionalities are enabled. The border router then acts as gateway between the wireless sensor network and the rest of the world. WSNs are nowadays commercialised [58, 59] and can be found in various fields such as road traffic monitoring, smart metering, inventory monitoring, healthcare, and obviously building automation [60, 61, 62, 63, 64]. As the number of WSN applications increases, there is an urgency for a federating application protocol allowing data sharing and interoperability between wireless sensor networks. This point is especially crucial as WSN applications will have to coexist with existing network infrastructures.

New applications

At the origin, WSNs were only performing simple tasks such as sensing and actuating. The measured data are pushed by the nodes through the border router into databases either site-local or in the cloud. In a second step, applications retrieve these data for further processing and display. This use case can be illustrated with the sen.se platform [291], showing that WSNs are no more restricted to professional use and have made their way into our daily-life. The sen.se Internet of Things solution is composed of what they call the mother (border router) and a set of battery powered cookies (sensors). According to the placement of the sensors, the cloud-based Web application collecting the data will be able to perform diverse functions such as analysing the sleep cycles (placed between the mattress and the sheet), the quality of toothbrushing (fixed on the toothbrush) or to build a calendar showing when you get out and come home (attached to the key ring).

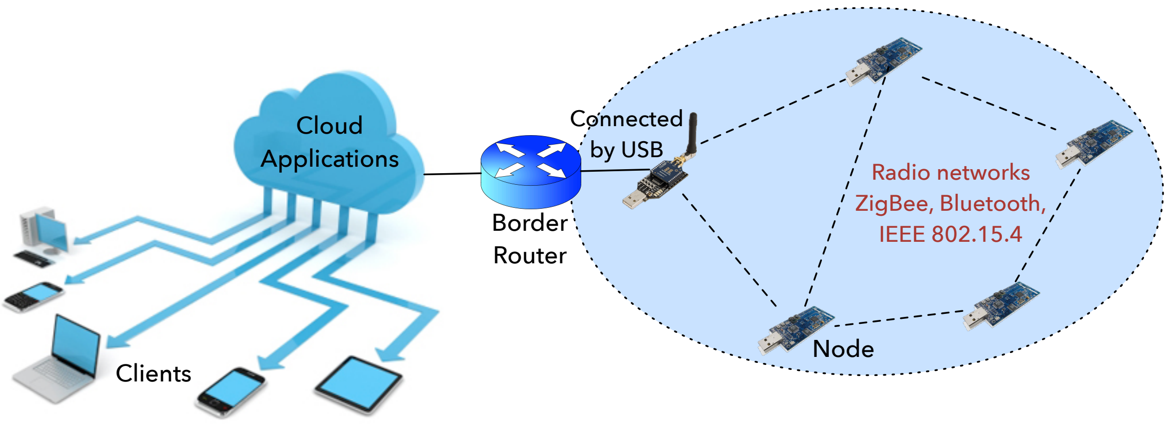


Figure 2.1: Example of a wireless sensor network where nodes are distributed among the physical environment for monitoring purposes. The communication is performed through low-power radio. The traffic directed to outward networks transits via a border router connected to the Internet.

With the advances made in embedded microelectronic, nodes are now endowed with significant computational power and memory, and can perform far much more than simply sensing the environment. For example, decision making based on rule sets or fuzzy logic can be executed by single nodes [65, 66]. From a broader perspective, WSNs appear as distributed computing and storage spaces that can be merged to process complex tasks [67]. From this observation, it is possible to envision the next step by considering WSNs as a sort of local clouds. The concept of cloud computing has been ported to WSNs under the term of *cloudlets* [68, 69] or also named *fog* by Cisco [70]. In the same way as traditional cloud services offer dematerialised storage or computing, a cloudlet distributes the services among the nodes without requiring any technical competence from the users. From an application developer point of view, there is no difference between clouds and cloudlets as software is provided as a service (SaaS). The cloudlet automatically adapts the distribution of services depending on the necessary resources [71]. While clouds leverage heavy infrastructures of servers, cloudlets have to deal with the dynamic nature of WSNs where nodes can appear, disappear and potentially move within the network. The capacity can simply be extended by adding additional nodes to the network, which therefore allows to easily build highly scalable networks.

The Internet of Things opens new dimensions by merging the digital and physical worlds relying on the same infrastructures. Although the IP protocol being a common denominator to provide interoperability between networks, the interoperability at the application layer has received only little attention. When considering that Gartner projects 26 billions IoT devices will be connected to the Internet by 2020 [72], this observation only emphasises the need to build scalable distributed applications on top of heterogeneous devices.

2.1.2 Heterogeneity in Smart Buildings

We here discuss what are the main causes of heterogeneity in smart buildings, and show the consequences of such situations. Finally, we describe two current gateway-based approaches solving some of the problems due to heterogeneity.

Origins of Heterogeneity

Smart buildings are a logical extension of today's high modernised society where everything is connected through networks and controlled by computers. Networks of sensors and actuators enable intelligent control of building equipments such as the HVAC. The access to those field devices is achieved through dedicated networks, often specifically tailored according to their purpose. This includes the physical architecture as well as the protocols for carrying the information. In recent years, smart buildings have been equipped with an increasing amount of networks [73]. Moreover, new systems emerged to control non-traditional equipments such as local energy production using solar panels, or smart metering to have detailed electricity billing [74, 75]. Although allowing specific control of building automation equipments, their incompatibility prevents the creation of overall management systems. The physical constraints and technology obsolescence are other arguments why buildings are equipped with more than one network. First, it can sometimes happen that a network should be extended in locations where installing additional wiring (power or twisted pair) is not feasible. Secondly, equipment can become defect and cannot be replaced due to the fact that there are no more spare parts. Such situations impose the integration of a new system, leading to so-called *heterogeneous systems* together forming isolated islands (Figure 2.2). A unified control could contribute in achieving a better management by combining all the available functionalities and therefore taking global optimal decisions instead of performing solely local optimisation [76, 77, 78].

Consequences of Heterogeneity

Putting together global management systems is not an utopian goal. We can indeed observe this in some systems present in several big buildings that successfully provide a way to save energy while ensuring an optimal comfort for inhabitants [79, 80, 81]. While feasible, they represent a daunting task from a developer point of view. Each technology has indeed to be integrated in the management system. A first condition before realising this requires a deep knowledge of building networks and equipment [82, 83]. The particular protocols must be understood and implemented in the management system. Furthermore, the data models representing measurements or actuation orders can be of different natures, as some systems are command-driven while others are event-based. This creates an inconsistency in how to represent and to manage data internally. From a business perspective, those heterogeneous systems are currently inducing higher costs during their entire life cycle. Upgrading with newer functionalities or including new technologies will often require reviewing the foundations of the architecture. In more general terms, they offer little flexibility to any changes.

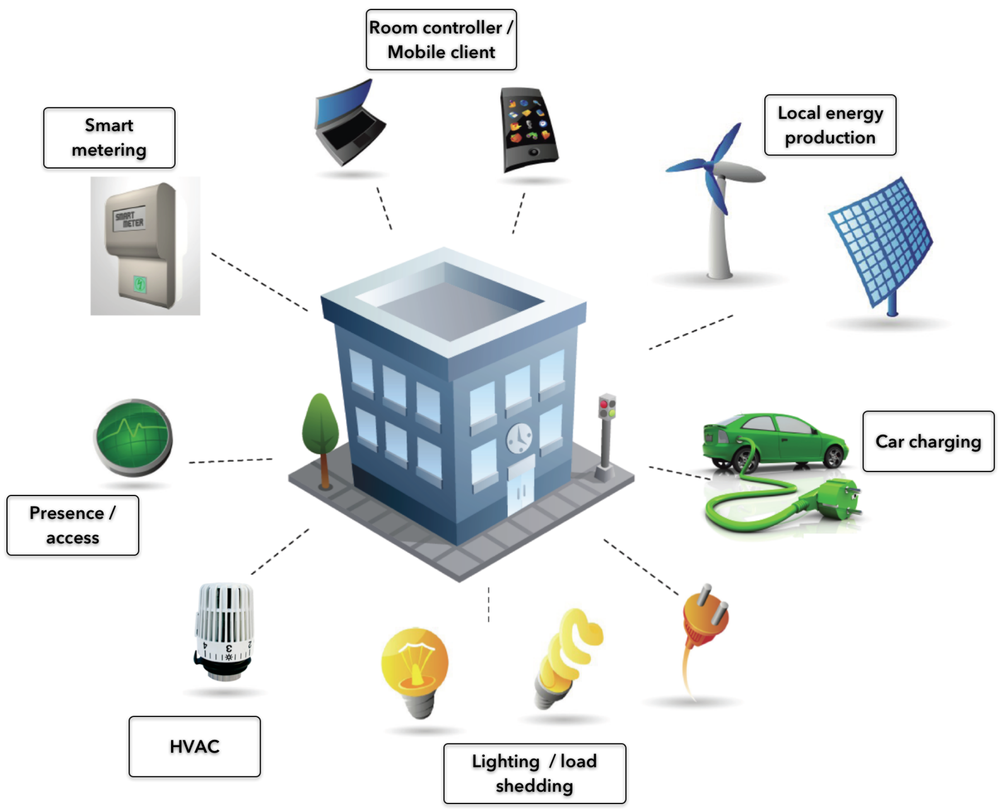


Figure 2.2: Nowadays, buildings are endowed with tremendous different technologies managing distinct facilities. Those systems are barely compatible with each other, creating isolated networks of sensors and actuators.

Gateway Mappings

Some works have proposed to rely on multi-protocol devices [29]. The drawback of this approach resides in the integration cost of such devices that are, anyway, quite non-existent on the market. Another solution consists of providing gateways. As illustrated in Figure 2.3, gateways can operate according to two modes, a *N-to-N* protocol mapping or a *N-to-1** approach mapping all sub-systems to only one central protocol. The *1** refers to a new protocol stack suited for IoT interactions. In the N-to-N case, gateways between BASs translate telegrams from the originating network to their destination. Those gateways have knowledge about each protocol stack present on the network. Although this approach solves the heterogeneity across networks, it induces some limitations. First, it is not possible to ensure that all capabilities of a BAS can be mapped to another one, thus restricting functionalities. Secondly, this approach requires $\frac{n*(n-1)}{2}$ bidirectional mappings between BASs, representing a considerable effort.

In contrast to the N-to-N approach, the N-to-1* one considerably simplifies the number of gateways needed to n by introducing a common technology. The key challenge resides in the 1* technology where no standard is currently defined. Its components have still to be identified, according to Internet-of-Things constraints. In this thesis, we focus on this latter promising vision where the common 1* protocol still needs to be specified.

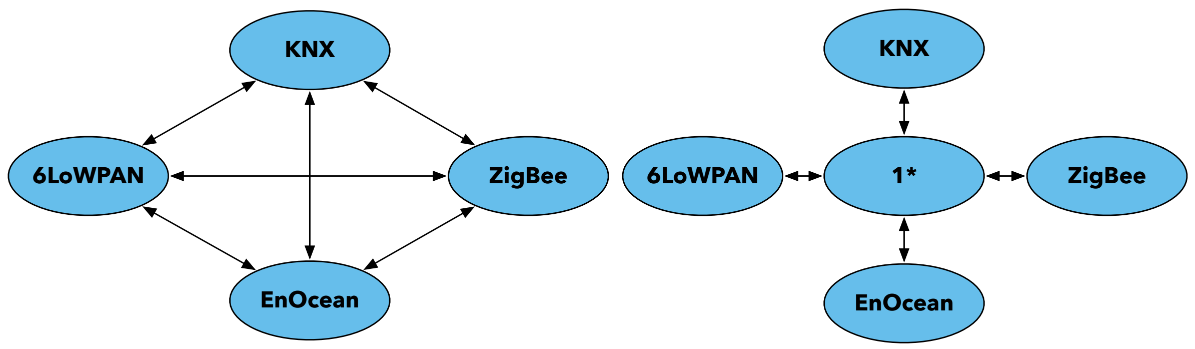


Figure 2.3: In the N-to-N approach (left) each device is able to understand the entire protocol set deployed in the building. The N-to-1* approach (right) advocates the use of a shared protocol between all the technologies.

2.1.3 Data-Driven Building Management

Another strategic evolution concerns the actual intelligence targeting energy savings and comfort increase. Currently, almost all systems follow a reactive optimisation. They implement more or less complex control loops and predefined rule sets, which certainly reach a substantial optimisation. Changes in the surrounding environment will trigger reactions, and commands containing new set-points will be sent to actuators [84]. However, the user behaviour and other parameters have a direct influence on the achievable gains and a transition to dynamic adaptive systems is therefore required [22].

For such a purpose, data-driven approaches are gaining momentum in smart buildings. Especially machine learning algorithms offer the possibility to learn how the building is used in terms of room usage, user comfort preference and seasonal weather among others [85, 86]. Those algorithms require a considerable amount of historical data to train mathematical models. Smart buildings are also not only controlled according to local conditions, but benefit from broader strategies. Machine learning approaches also allow to anticipate the control by guessing future states. Applying this at the temporal presence of users, the building should for example be able to anticipate their leaving, and therefore turning off the heating in advance according to the thermal inertia. Machine learning algorithms can also be very useful for sensor fusion and for the creation of higher level sensing applications. The fields of appliance and activity recognition are examples among others [87, 88, 89]. Only little real-life implementations have been deployed yet, being mainly confined to the scientific community. Proposing reusable blocks to deploy machine learning applications is therefore of high interest.

The machine learning process usually involves two distinct steps [90, 91], as shown in Figure 2.4:

Training: The training phase relies on historical data of sensor measurements and a learning algorithm. The algorithm processes the data in order to find relations able to characterise the output classes. Those relations will then be represented as mathematical models or trees (depending on the type of algorithm) to be executed by the runtime.

Runtime (testing): The runtime process (referred as testing in the machine learning literature[1]) will use the mathematical models generated by the training. Along with the sensors live-data, it will perform a prediction or a classification for example to define which are the most probable classes and states. Further, it can also predict the most plausible next step.

[1]More precisely testing refers also to the evaluation of a trained model on an independent set of data, i.e. the testing data. If this test phase is successful, the model is then used at runtime relying on the very same procedure, i.e. attempting to label or predict using unseen data as input.

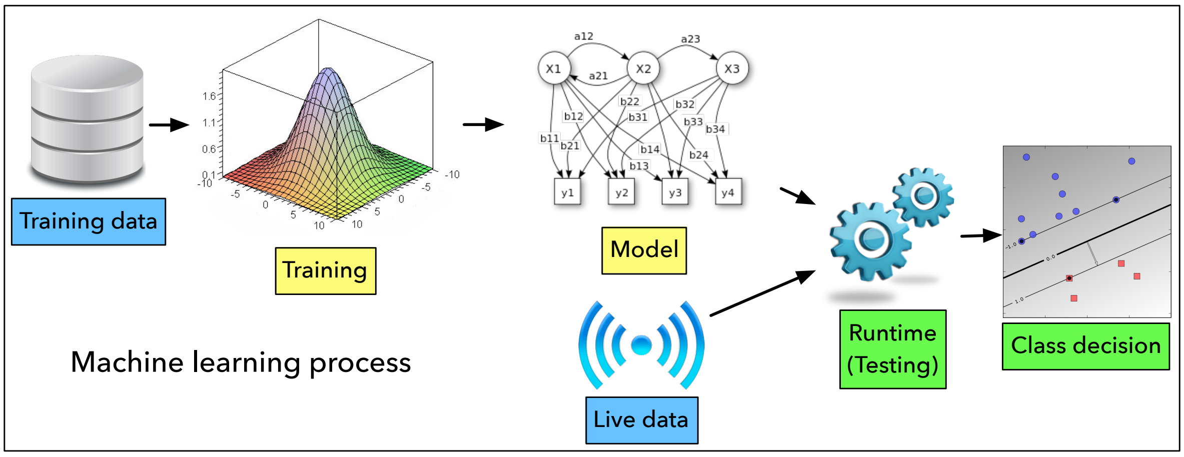


Figure 2.4: The machine learning process can be decomposed in distinct steps. The training part uses historical data as input and generates mathematical models. The runtime step (or testing step) applies the models on live data to perform the classification.

2.2 Building Automation Systems (BASs)

As their name says, building automation systems aim at endowing buildings with an intelligent layer to optimise the energy consumption while keeping the comfort perception of users at its highest possible level. They are also used to trigger alarms when abnormal conditions are observed, requesting an intervention or a maintenance from expert people. With the evolution of networking and electronics, building networks have grown in complexity and are now comparable to fully stacked computer networks. Such ones like KNX, BACnet and EnOcean work over their own proprietary protocols, meanwhile following the OSI model for most of them.

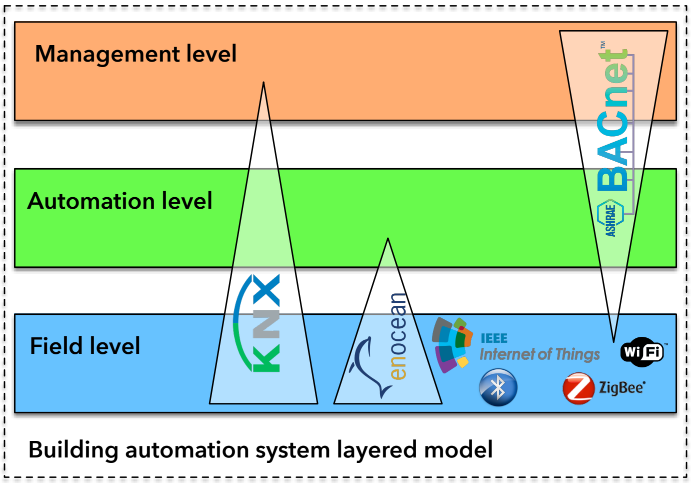


Figure 2.5: Building automation systems are composed of three layers. The field level gives access to devices functionalities while the automation level implements control strategies (control loops and rule sets). Finally the management level offers configuration interfaces for commissioning and other enterprise-level tasks.

Apart from offering proprietary application layers (freely accessible for some of them), building networks providers began to include more and more functionalities into their system. It starts by offering configuration and commissioning tools, up to the possibility to define simple control loops that will be implemented by devices (for example local heating regulation). Although facilitating the handling by installers having to master only one ecosystem, the tight coupling between the components impairs the evolvability and openness. In order to counter this trend, it was proposed to apply a separation of concerns striving a distinct management and composition of the different features offered by BASs [92, 28]. This has led to the layered model shown in Figure 2.5, which is composed of the following layers:

Field level is composed of devices geographically disposed within the building. They form together a building network either being wired or using wireless technologies. All the underlying protocols necessary for routing and addressing are specific to the vendor's specification and implementation. The sensors and actuators offer accessibility to their internal representation of measurements or states by relying on the vendor's application protocol.

Automation level is at the heart of the intelligence of smart buildings by implementing control strategies. They range from simple sense and react interaction models (for example turn the light on when movement is detected) up to complex decisions involving weather forecasts and the enterprise's information systems, such as room occupancy plans and access management. This level depends on the functionalities offered by the field level to gather data from the physical environment and to send actuation orders when applying the control strategies.

Management level provides information from throughout the entire system. Typically, a GUI is presented to the administrator for manual intervention. Access to the automation and field levels is possible to modify parameters including schedules and control loops, as well as the addressing and provisioning. Alerts are generated for exceptional situations like technical faults or critical conditions. Databases store long-term historical data with the possibility to generate reports and statistics.

Although the intention of this layered approach was to increase interoperability, vendors have given only little attention to this proposition. As a result, building automation systems remain closed systems only compatible with devices or software components that were especially conceived for a particular technology, and sometimes approved by vendors.

In this section, we give further insights on the most widespread commercial solutions in Europe by outlining the fundamental network and application concepts of KNX, EnOcean and BACnet. We also present the foundations of the emerging Internet of Things related technologies, and we finally conclude by discussing the consequences of heterogeneity.

2.2.1 KNX

The KNX protocol is a standardised (ISO/IEC 14543) network technology for building automation systems. It is the successor and a merging of three previous standards: the European Home Systems Protocol (EHS), BatiBUS and the European Installation Bus (EIB). Its history has started around 25 years ago and the protocol has not undergone any major changes since then. It is used for several automation purposes like lighting, HVAC and shutter controls. Additionally, it can monitor alarm components and manage energy with smart meters. The KNX standard is defined by the KNX Association involving a large number of device manufacturers.

Network Layer

KNX is not restricted to a particular transport medium and supports several ones like twisted pair, radio (KNX-RF) and power-line, as shown in Figure 2.6. The twisted pair is meanwhile the most commonly used medium (also due to the historical background) as it can also directly power a majority of devices itself. On the twisted pair, the data rate is of 9600 bits/s.

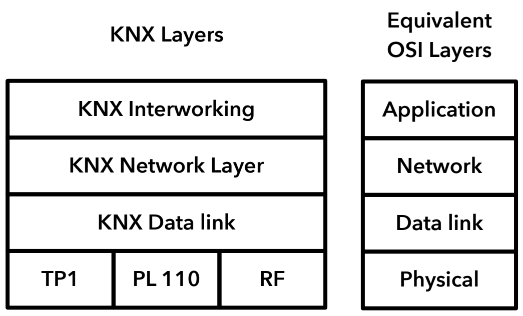


Figure 2.6: KNX protocol stack compared to the OSI model.

In the same way an IP network is composed of sub-networks connected via routers, a KNX installation is divided in areas and lines, as illustrated in Figure 2.7. A line is the smallest network form that can live independently in a building, where up to 256 devices can be connected. Up to 15 lines can be connected together forming an area which can handle a maximum of 3840 devices. Finally, a large building can host up to 15 zones, allowing a maximum number of 57'600 devices. Each line and zone are galvanically protected from each other with couplers to prevent interference, hence requiring a large amount of power supplies.

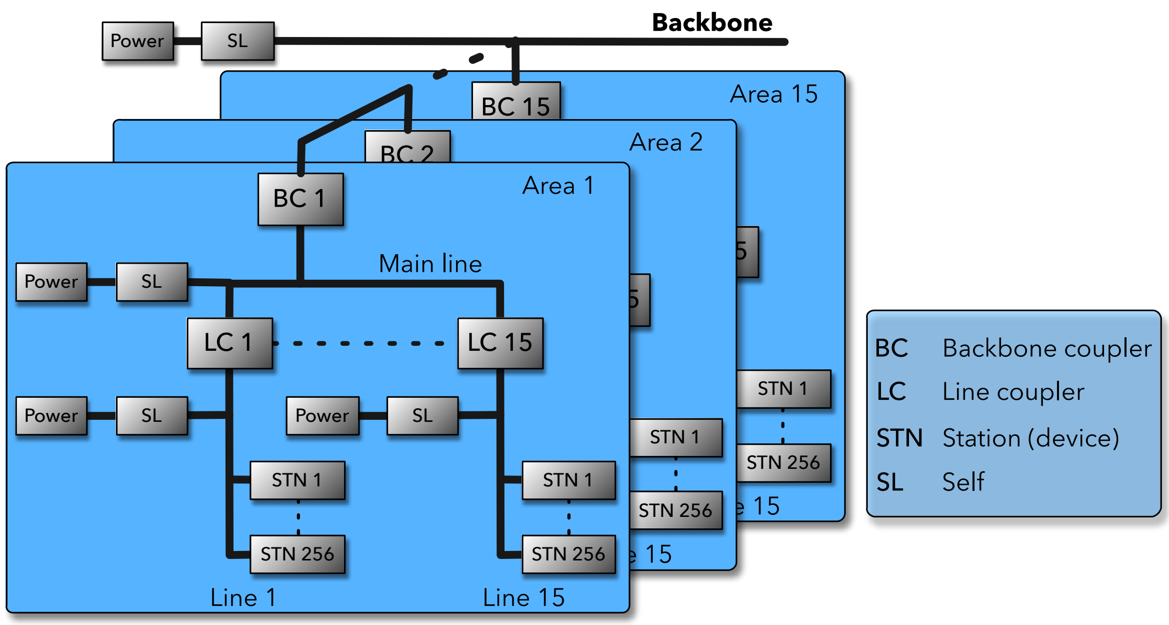


Figure 2.7: A KNX installation is composed of lines forming areas, themselves connected to a backbone.

Individual addresses are attributed to devices, forming a three-level address space following the network architecture as follows *Area[1-15]/Line[1-15]/Station[1-256]*. By using a 16 bit address space, 65'536 devices could be theoretically addressable. Group addresses (multicast) are also encoded with 16 bits and can follow 3 different structures: totally free (user defined), 2-level (main group/subgroup) or a 3 level (main group/middle group/subgroup). This level structure is used to identify scenarios such as main group = floor, middle group = functional domain (e.g. switching or dimming) and subgroup = function of load (e.g. kitchen light on/off).

Application Protocol

The KNX application layer, also known as *KNX interworking*, was thought to ensure interoperability between devices of various manufacturers. It standardises the way how payload data inside telegrams have to be structured and interpreted. Like many systems dedicated to automation, the interworking is based on so-called functional blocks to describe system functionality [93]. Logical parts of a device, such as a specific function are symbolised by those Functional Blocks (FBs). We can illustrate this principle with an example of a light switch FB that is a logical function of a four channels relay. A functional block is always attached to one device only.

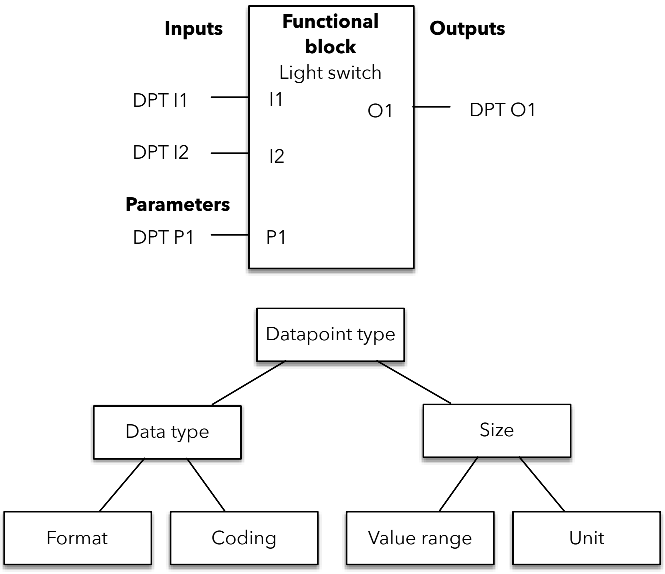


Figure 2.8: The upper part shows a functional block structure composed of datapoints acting as communication endpoints. The sub-components of a datapoint type including data representation and signification are depicted in the lower part.

As visible in the upper part of Figure 2.8, FBs are composed of a set of datapoints (DPs). Those datapoints are communication endpoints of devices, allowing access to the functions of a block [94]. The inputs (DPT I) stand for states that can be altered by other devices. The parameters (DPT P) are configured by administrators or engineers to change the behaviour of the FB. At last, the outputs (DPT O) give insights of the actual state of the block. KNX administrators simply link outputs and inputs together to associate devices, and can therefore easily create scenarios. Datapoint types are also standardised in terms of syntax and semantics as visible in the lower part of Figure 2.8. The standard also defines several categories where datapoint types are regrouped according to their FBs purposes. By knowing the datapoint type, one can find in the KNX specification all the information regarding the datapoint, including the format, coding, value range and unit.

The KNX protocol identifies two categories of DPs: group objects (GOs) and interface object properties (IOPs). IOPs are only used for configuration and management purposes. One can address an IOP only with the physical address of the device. On the other hand, GOs are used by sensors, actuators and control devices to exchange information during runtime. The GOs are endpoints involved in group communications between producers and consumers basing on a multicast approach. An administrator can define groups by linking GOs of the same type together. Sensors can only be part of one group while there is no restriction for actuators. During runtime, sensors will send telegrams by indicating the group address of the GO as the destination, besides marking a flag indicating a write command. All actuators having a GO with the same group address will consider the telegram and write the payload value in their own GO. According to the value, the internal program will then actuate the physical outputs. Besides the write command, the protocol also allows to read GOs. Group objects can therefore be seen as shared variables among devices. This group approach over multicast allows to create dynamic scenarios as shown in Figure 2.9. The light switch in group 1/1/1 (green) will only be able to drive one light while the one in group 1/1/2 (red) will be able to drive both lights simultaneously.

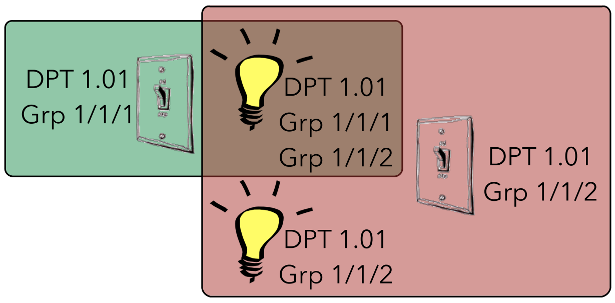


Figure 2.9: KNX basic automation by defining groups of functionalities.

Configuration

KNX describes three modes to configure devices in a network:

A-mode or automatic mode is intended for end-users as it requires no configuration. Devices automatically configure themselves and are ready out-of-the-box. This mode has never been implemented so far.

E-mode or easy mode is used with devices that are already preloaded with partial configuration. Only basic installation skills are required for allowing the devices to work. It is used in medium-sized deployments with simple tools.

S-mode or system mode is preferred in large-sized deployments where specialists will configure all the parameters. The devices have no default behaviour and need to be commissioned with a dedicated software.

The ETS (Engineering Tool Software) is developed by the KNX association and sold to professionals. Its main purpose is to configure the devices. The configuration, or commissioning, includes setting the individual and group addresses, as well as uploading the internal program. After having downloaded the product description (including internal program) from the manufacturers website, depending on the device type, other parameters influencing its behaviour can be configured (for example thresholds of control loops). After ETS having written this data in the memory of the device, it will become a participant of the building network able to communicate with other devices. Network monitoring, alarming and telegram storage are other features accessible through ETS as well.

2.2.2 EnOcean

The EnOcean company started in 2001 with a simple concept: to propose a building automation system requiring no wires and where most devices do not need any power supply. Various types of devices are proposed, ranging from push buttons, to HVAC sensors, blind controls and motion sensing. A large quantity of devices are self-powered through energy harvesting. For some devices, small solar panels are mounted on the housing and provide sufficient energy.

Other devices gather their power from mechanical actions (piezo-electricity) done by users, as for example pressing a button or moving a window, respectively a door handle. The EnOcean company develops these energy harvesting technologies and offers integrable circuits to manufacturers. A small quantity of sensors (for example CO_2) needs an external power supply by either using batteries or being plugged into a socket, because the actual power gathered by energy harvesting would not be sufficient. Actuators from their side cannot be powered by energy harvesting and have to be supplied with external power. This can be explained by the required energy, for example to drive motors and is due to the fact that they are never put in hibernation. By avoiding wiring, users have a large field of liberty regarding the placement of the sensors, as they can be placed anywhere in a room.

Network Layer

The EnOcean network architecture is kept as simple as possible. The EnOcean protocol stack only comports two levels, as shown in Figure 2.10. There is no notion of sub-networking as all messages are broadcasted to whom may receive them. In order to be energy efficient, sensors are hibernating most of the time and wake up only when a telegram should be sent. A telegram transmission is triggered if the physical property measured has changed by a certain predefined threshold (often 5%) or after a period of time has elapsed. Actuators on the other hand are constantly listening for incoming telegrams, such that they can react in a real-time manner. Devices communicate in simplex mode as sensors are only equipped with a transmitter while actuators only embed receivers. Although having a positive influence on energy consumption, this communication model (only event-based) strongly limits the scope of applications as it does not offer the possibility to retrieve current states (request-response). A central control system must therefore listen to all telegrams in order to determine the last measured value of a sensor. Another shortcoming of simplex communications is the inability to confirm the reception of telegrams. EnOcean devices working with the Dolphin platform (an evolution of the first commercialised platform) however are able to send acknowledgements as their radios are featured with transceivers. Meanwhile most devices available on the market are not using the Dolphin platform yet.

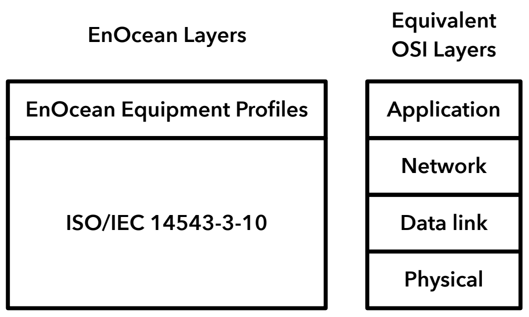
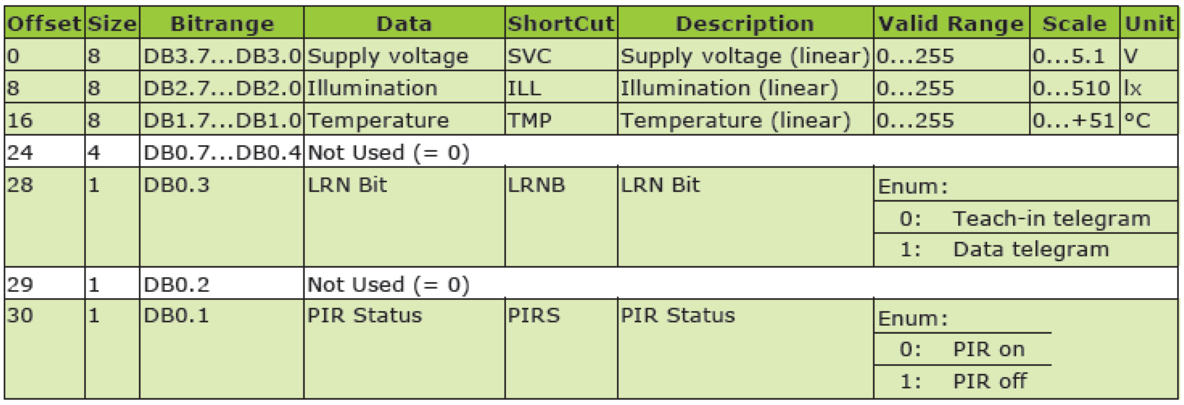


Figure 2.10: EnOcean protocol stack compared to the OSI model.

Depending on the type of power source (energy harvesting or external), the range in outdoor can cover a distance up to 300m. In indoor conditions, the maximal attainable distance is 30m. This range may represent a limitation for some scenarios. The use of repeaters can extend the distance a device covers. Wireless telegrams are relatively small, with the packet being only 14 bytes long, transmitted at a rate of 125 kbits/s. EnOcean uses a 4 bytes addressing scheme where the ID (equivalent to an address in the EnOcean jargon) is defined by the manufacturer and not further configurable [95]. Originally, only the ID of the sender was figuring in the telegrams. As this is not compatible with bidirectional communications, the telegram format has been adapted by encapsulating the destination ID within the payload, where a header flag indicates if encapsulation is used or not.

Application Protocol

The EnOcean application layer, composed of EnOcean Equipment Profiles (EEPs), is intended to ensure compatibility between manufacturers. The EEP definitions are managed by the EnOcean Alliance grouping several manufacturers. An EEP defines the format of payload data according to the sensor type and its functionalities [96]. The vast majority of EEPs are related to sensors as only few actuators are communicating bidirectionally, and are therefore not able to send telegrams. A three-level hierarchy is used to categorise EEPs, and allows to uniquely identify a device type. The first level gives insights on the type of telegram, especially the payload length by allowing the following sizes: 1 byte, 4 bytes and variable length. The second level defines the functionalities of devices, or more precisely what they can measure (temperature, humidity, occupancy, gas, etc). The last level is finer grained and targets the type of sensor (-30°C to +50°C).



Offset	Size	Bitrange	Data	ShortCut	Description	Valid Range	Scale	Unit
0	8	DB3.7…DB3.0	Supply voltage	SVC	Supply voltage (linear)	0…255	0…5.1	V
8	8	DB2.7…DB2.0	Illumination	ILL	Illumination (linear)	0…255	0…510	lx
16	8	DB1.7…DB1.0	Temperature	TMP	Temperature (linear)	0…255	0…+51	°C
24	4	DB0.7…DB0.4	Not Used (= 0)					
28	1	DB0.3	LRN Bit	LRNB	LRN Bit	Enum:		
						0: Teach-in telegram		
						1: Data telegram		
29	1	DB0.2	Not Used (= 0)					
30	1	DB0.1	PIR Status	PIRS	PIR Status	Enum:		
						0: PIR on		
						1: PIR off		

Figure 2.11: Example of an EnOcean equipment profile describing a light, temperature and occupancy sensor (EEP A5-08-01).

Unlike the KNX datapoints identifying single endpoints, an EnOcean equipment profile can hold more than one data. For example an occupancy sensor able to measure the temperature, illumination and presence will transmit all the measurements within one telegram. In order to properly interpret the data, an EEP specifies the data position in the payload, as shown in Figure 2.11. Besides giving information about the position and size of fields, the valid range and scale are used to convert the binary encoded value in its physical dimension by applying a cross-multiplication.

Configuration

The configuration of EnOcean devices is easy to handle by people without extensive technical knowledge, but can represent a daunting task for large buildings. Interaction between devices (sensors to actuators) is based on a physical pairing approach. Devices have to be teach to each other during the teach-in phase, requiring physical access to the devices, as this procedure cannot be realised through software. First, the actuator has to be put in learning mode by pushing a button or turning a track wheel with a screwdriver. At this time, the actuator is ready to receive special telegrams from sensors. These special telegrams called teach-in telegrams contain the

EEP identification number of a sensor. Such telegram is triggered on the sensor by passing a magnet at a specific place. If the actuator is able to understand this EEP, it will record the ID of the sensor and react to its telegrams in the future.

This configuration mode is especially disadvantageous for large buildings when controlled by a central unit. The teach-in procedure has to be repeated for each actuator in the building, having to learn its useful neighbour sensors and the central unit. It can lead to a high problematic situation whether the control unit has to be changed for any reason. The whole teach-in procedure will have to be repeated on all actuators. This lack of evolvability is related to the ID (comparable to a MAC address) that cannot be changed on a radio. The EnOcean Alliance provides the freely available Dolphin View software, including functionalities to monitor the wireless network traffic and to analyse telegrams, but offers no higher-level services.

2.2.3 BACnet

The Building Automation and Control Network Protocol (BACnet) was especially developed to provide a high level of interoperability between various vendors and network topologies. It targets to be used in control systems of all sizes and types. Unlike KNX and EnOcean, its conception follows a top-down approach by offering more high-level services (management and integration within information systems) than being focused on the field level (physical connections and topology). Its conception started in 1987 and led to several standards: ANSI/ASHRAE in 1995 [97] and ISO in 2003 [98]. So far, the BACnet specification is under continuous maintenance and further development.

Network Layer

BACnet tries to make abstraction from the network level by almost guaranteeing that messages can be conveyed over any type of network. However the specification standardises a small number of networks that are de-facto supported by BACnet in order to maximise the probability that devices are using the same network. The chosen networks are Ethernet, ARCNET, Master-Slave/Token-Passing (MS/TP), LonTalk and Point-to-Point (PTP). Figure 2.12 shows the entire protocol stack with the natively supported lower-layers. By relying on the BACnet Virtual Link Layer (BVLL), other networks that are based on IP can be easily integrated.

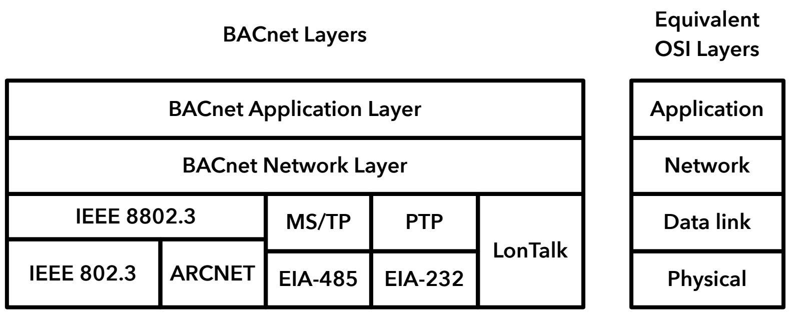


Figure 2.12: BACnet protocol stack compared with the OSI model.

The segment is the base element in a BACnet topology. Segments are physical connections of cables, which can be coupled together using repeaters and bridges, forming a network. These networks that may use different mediums are linked with routers forming a BACnet internetwork. BACnet routing is kept simple and there should only exist one path between any two devices of an internetwork. The addressing in BACnet is realised with a 2-level scheme. The first 2 bytes identify the network while the second level targets a local address within a network, which can be up to 255 bytes long. The BACnet routers interconnecting the networks route the packets relying on the network number [99]. They need to learn the routes by themselves. The only information they are provided with are the network numbers that are directly connected to their ports. With this little information, they are able to learn the topology by using routing protocols offering dedicated services such as Who-Is-Router-To-Network [292].

Application Protocol

BACnet functionalities are represented in terms of objects. Each object is a collection of data elements, all of them related to a particular function. These objects are comparable to endpoints used by the control application. Individual data elements of an object are called properties. For example, an analogue input object of a device measuring an outside temperature will have a "Present_Value" property containing the actual value read from the physical input. Plenty of properties can be used to describe objects, such as minimum and maximum possible values, resolution, the unit and reliability statutes. Each object type is well defined and specifies whether properties are mandatory or optional, and if they should be writeable or read-only. The BACnet standard defines 54 different standard object types. They range from simple object types such as binary input and output, analogue input and output, up to more complex and higher-level objects related to scheduling, trending (data storage), alarming and life safety procedures. Every device participating in a BACnet network may have none or several objects of each type, apart from the mandatory "device" object. This special object is used to have access and control over various configuration properties of the device [28]. The protocol allows to add user-defined objects and properties with no restriction, meanwhile being not compatible with other facilities.

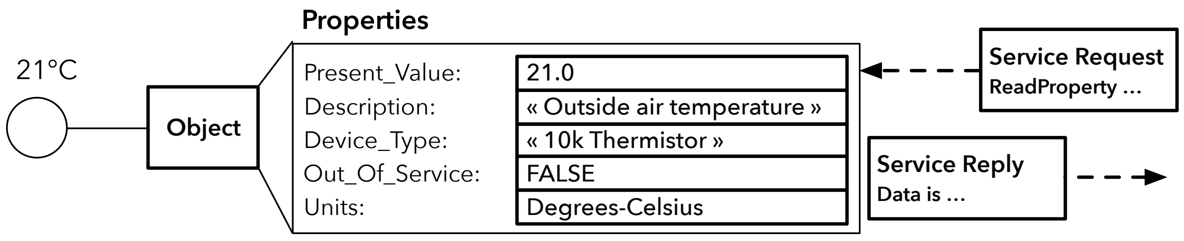


Figure 2.13: BACnet application layer components. Objects represent endpoints such as an analogue input characterised by a set of properties, which can be managed by using service messages.

In the context of BACnet, services are messages dedicated to the manipulation of properties. The communication follows a request-response model. Services are grouped into five categories of functionalities - object access (read, write, create, delete); device management (discover, time synchronisation, initialise, backup and restore database); alarm and event (alarms and changes of state); file transfer (trend data, program transfer); and virtual terminal (human machine interface via prompts and menus). BACnet also provides three types of eventing models depending on their purpose (alarming, custom thresholds, and simple change of value).

While the BACnet standard defines a large quantity of objects and services, only a few of them need to be really implemented on devices. Offering all these services on the devices would increase complexity and costs without adding any benefit. BACnet has therefore introduced the concept of BIBBs (BACnet Interoperability Building Blocks) to concisely describe the functionalities offered or required by a particular device [100]. A BIBB describes a specific functionality according to following interoperability areas: data sharing, alarm and event management, scheduling, trending and finally device and network management. BIBBs are specified in client/server pairs, giving insights whether a specific device works as requester, responder to a service request or both. For example, the BIBB "AE-ACK-A" requires that the server-acting device implements the service "AcknowledgeAlarm".

Although those BIBBs make of BACnet a well-defined protocol, most of the objects are not implementable on constrained devices as they target higher-level management application services. Furthermore, their object model is rather rigid as all functionalities are predefined, leaving only little space for evolvability.

Configuration

BACnet offers useful services to discover devices and specific objects on the network. Services such as Who-Is/I-Am and Who-Has/I-Have can be used for remote device management. Devices become discoverable through their object names and identifiers. Other services in this category allow for time synchronisation and reinitialisation of devices. These services are used by room controllers and control software to create scenarios between sensors and actuators.

No specific software is particularly recommended for commissioning or managing a BACnet network in contrast with KNX and EnOcean. Network administrators have to look for software able to manage the BIBBs or objects that are intended to be used.

2.2.4 Internet of Things

In recent years, several WSN technologies have emerged under the umbrella of what is called the Internet of Things (IoT). The IoT-idiom has several origins, one of the very early ones being in the field of RFID. Rapidly, the concept of a global network of things has been further pushed, closer and closer to Internet protocols. It now includes not only RFID tagged objects but all kinds of embedded devices offering connectivity. In short, the basic idea of the IoT concept is the pervasive presence around us of a variety of things or objects that are able to interact with each other and cooperate with their neighbours to reach common goals [101]. While the boundaries of the IoT are still evolving, the scientific community has started to enumerate a number of IoT principles covering both technological and economical aspects [102]:

- Things are physical objects uniquely identified and connected to the Internet [103].
- Things do not embed fully-fledged computers but can feature tiny computers, in which case they are sometimes qualified as smart things. For example, a temperature sensor embedded on a Wi-Fi module, including a small footprint TCP/IP stack and a simple Web server could probably be qualified as smart thing. Other (not-so-smart) things will be potentially more constrained, having indirect connectivity and exposing single state information. For example, a consumer good with a passive RFID tag may just expose its identity to a mobile device.
- Things are machine-centric instead of user-centric. Internet-based services are mostly targeting human beings as users, e.g., the Web, emails, chat, file transfer, etc. In most IoT-applications, smart things communicate amongst each other and with computers in the Internet. The IoT is therefore related to machine-to-machine technologies.
- Things focus on sensing. The IoT is presented as a very finely granulated nerve system of the physical world with trillions of new nerve endings. The huge measurement capability of the IoT opens the doors to many new applications.

Early Technologies

Technologies such as ZigBee and Bluetooth are since many years reference technologies for WSNs, especially in building automation, smart cities and smart metering [104, 105, 106, 107]. Developers are able to rapidly prototype applications as all layers apart from the application one are specified by the protocols. Additionally, their low energy consumption (thanks to optimised physical layers and small overhead protocols) allows devices to be battery-powered while offering a significant autonomy, particularly if combined with duty-cycling methods [108]. ZigBee has the advantage over Bluetooth to rely on the concept of mesh networks augmenting the network's range, while requiring no dedicated infrastructure. However those networks are not compatible with IP, therefore requiring gateways for address translation [109]. Moreover, TCP or UDP are mandatory transport protocols in some scenarios and cannot be relayed at the sensor network, enforcing the use of different application protocols. In order to cope with these drawbacks, Wi-Fi appears as key candidate due to its native support of all IP-related protocols. Device miniaturisation induces memory restrictions, hence the necessity for low footprint stack implementations. A demonstration of the feasibility to embed the TCP/IP stack in very limited hardware architectures has been proven in [110], where a 10 kbytes implementation is running on 8-bit micro-controllers. Since then, small-sized devices such as the FlyPort OpenPicus (shown in Figure 2.14) featuring a low-power WiFi module with full TCP/IP support have democratised. This makes them seamlessly integrating into the Internet.

Figure 2.14: The OpenPicus FlyPort embeds a low-power Wi-Fi module and a full TCP/IP stack.

6LoWPAN

While Wi-Fi allows native Internet connectivity, it quickly appeared not adapted for some WSN applications where battery operation is mandatory. Unlike ZigBee and Bluetooth, it was not intended to be used on low-power devices and is therefore not optimised in terms of power consumption. A constrained Wi-Fi device will consume approximately four times more energy than ZigBee [111]. Recently, developments worked on adapting the emerging IPv6 protocol to meet the energy constraints of these devices and proposed the 6LoWPAN architecture [38, 39]. The IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) standard was first specified in RFC4944 [112]. It lays the foundations how IPv6 can be transported over IEEE 802.15.4 physical and data link layers. As the IPv6 header is originally too big (40 bytes) for an IEEE 802.15.4 frame (127 bytes), leaving only little space for payloads, it addresses this shortcoming by specifying an IPv6 compressed header (2-7 bytes). By using the RFC6282 [113] specifying the compression of the UDP header (4 bytes instead of 8) allows for 70-75 bytes of payload with enabled security, as shown in Figure 2.15. Meanwhile only a compressed version of UDP has been standardised, TCP (20 bytes header) can also be used, although not recommended because of reassembling complexity and due to the quite frequent packet losses. The header compressions along with the dispatch header indicating the type of packet on top of IEEE 802.15.4 are together called the 6LoWPAN adaptation layer.

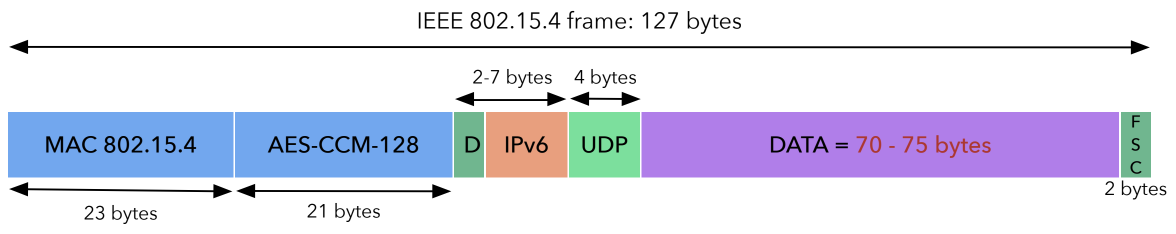


Figure 2.15: IEEE 802.15.4 frame composition (with AES security) using compressed IPv6 and UDP headers.

The routing in a 6LoWPAN network can be realised at two distinct levels:

Route-over performs the routing at the network level uniquely by considering the IPv6 header. In route-over, each IPv6 packet is reconstituted on each intermediate equipment in order to take the routing decision. Route-over is recommended in unsteady conditions (packet loss). RPL (Routing Protocol for Low power and Lossy Networks) being standardised by the IETF is the most used protocol following the route-over principle. It is a IPv6 with distance vectors protocol building a directed acyclic graph [114].

Mesh-under on the other hand performs the routing decision at the 6LoWPAN level and thus only with the fragments of the IPv6 packet. In this case, the IPv6 packet is reconstructed only once having reached its destination, allowing a shorter transmission delay than route-over [115]. LOADng (Lightweight On-demand Ad hoc Distance-vector routing protocol – Next Generation) is the most used in its category [116]. Despite the fact that it is still a draft, it has already been used in large deployments in the field of smart metering by ERDF in France (35 million devices) and by Enexis in the Netherlands (2.5 million devices), attesting its maturity.

Networks relying on 6LoWPAN have successfully been deployed as prototypes in several contexts. In [35], a network of 6LoWPAN devices is used to perform home automation. A large scale deployment in buildings is presented in [117] where custom made BatMote nodes running the Contiki operating system are used for electricity metering and light control applications. Moreover, the use of 6LoWPAN in the field of building automation has been deeply surveyed in [118]. Industrial automation is no exception to the trend of using 6LoWPAN networks [119]. Contrary to what one might think, this technology is not limited to low spanning networks, but was already tested for larger-scale implementations such as smart grids [120, 121]. Although 6LoWPAN does not specify any particular application protocol, most of the aforementioned research rely on the Constrained Application Protocol (CoAP).

2.2.5 Discussion

Interoperability is a key enabler in heterogeneous networks. However, because building automation systems have their own set of protocols, messaging principles and data models make coexistence difficult to achieve. Being compatible with other technologies can be a major advantage over the competition when clients have to decide for a system, as this leaves the door open for a possible evolution. This is the reason why some manufacturers offer multi-protocol gateways as for example between KNX and EnOcean. The possibility to offer gateways largely depends on the messaging and data models. As KNX and EnOcean both define their data representation with datapoints types and EEPs, some of them may be translated. Data models being specified according to the available device functionalities, only a small subset of endpoints can be translated. The functionalities offered by a network composed of different protocols will therefore be restrained to the ones having common representations. This consequence leads to the thought about the importance of data models on interoperability. This interoperability can only be achieved if data models are themselves compatible with each other. A shared higher-level data model making abstraction of subsystem particularities is therefore required.

Table 2.1 summarises the features of the previously outlined building automation systems. We base our comparison according to the ISO 16484-3 standard [293] describing functions that should be offered in building automation (ranging from configuration up to data storage). We complete this list with features that in our point of view should also characterise such systems. From this comparison, we can observe that BACnet is the most complete protocol in terms of functions, followed by KNX and EnOcean. However, according to our own criteria, KNX appears as the one being the most flexible and complying with evolvability, followed by EnOcean. BACnet for its part offers many functionalities but induces a higher complexity.

Feature	KNX	EnOcean	BACnet
Configuration	+++	+	++
Grouping/zoning	+++	++	+++
Event-based notification	+++	+++	+++
Alarming	++	+	+++
Scheduling	++	+	+++
Historical data storage	++	+	+++
Data model flexibility	++	++	+
Energy harvesting	++	+++	+
Application protocol simplicity	++	+++	+
Request-response interaction	+++	+	+++
User-defined applications	++	+	++

Table 2.1: Features comparison between the different building automation systems (+ means low, ++ means medium and +++ means high).

Apart from BACnet defining objects for control loops and data storage, none of the other offer services to run user-defined control applications. Control loops are preloaded in the devices that will be responsible for controlling the physical environment, as for example heating thermostats. As consequence, complex control strategies are implemented server-side with all the inherent risks, such as single point of failure, privacy and possibly network congestion due to the need of carrying all the data to a central sink. This goes against the principle of distributed systems and offers only little evolvability. The concept of building control must therefore evolve towards vendor-neutral applications that implement higher-level strategies interoperable among building networks. In this direction the Web 2.0 along with mashups is a success story for composing applications using content from heterogeneous sources. This approach could be transposed to control applications envisioning a unified control relying on heterogeneous building networks.

2.3 Distributed Systems Architectures

Distributed applications that are hardware independent have been studied for a while in computer science, and various concepts according to application types have been proposed [122]. The interoperability at the application layer between software components is the first concern of these concepts. They are of course less concerned by the particular challenges of sensor networks in smart buildings. While the classic middleware solutions could be used at the Management Level to create reporting applications or at the Automation Level by including enterprise information such as occupation plans in the control strategies, they cannot be used natively on things or field devices, because being business-driven.

The notion of architectural style describes a set of generic patterns and constraints to develop distributed applications. An architecture for its part is a reusable system structure built of components, interconnections and interactions between them, which are defined by standards and policies. They all together provide a reusable template for subsystem structure and communication among them. A software architecture establishes how system elements are identified and allocated, and especially how they interact to form together a system. The communication between the components is also an important aspect which is characterised by the amount of granularity needed for interaction.

In this section we provide an overview of the major distributed architectures (object-oriented, service-oriented and resource-oriented). Then, we evaluate their suitability in contrast to the requirements of smart buildings and discuss the advantages and drawbacks of each style.

2.3.1 Object-Oriented Architectures (OOA)

As mentioned in [123], distributed instances are the key concept in SOA to build distributed applications. Remote procedure calls (RPC) is the main interaction style to communicate in a stateful manner with distant objects. The interfaces used during the remote calls as well as the data representation (marshalling of data structures) is standardised according to the used technology. Scalability and maintenance are rather difficult to handle with RPC because of the tight coupling. Furthermore, each slightly modification of an object can cause inconsistency throughout the system as interfaces could potentially be no longer valid. We can find this architectural style in distributed systems where many standards for RPC are available, such as DCOM (.NET) [294], RMI (Java) [295], Pyro (Python) [296] and more recently CORBA (multi-languages) [297] and JavaSpaces (Java) [298].

Although allowing a transparent interaction with distant logical entities of an application (remote objects), this architecture suffers from a major drawback. Indeed the available RPC systems make use of closed or proprietary protocols not able to communicate in the Internet. This prevents from deployments at a large scale between cross-boundaries applications. Because of the tight coupling ensuring maximal performance, this style is the preferred one for closed or critical use cases (banking, health and stock exchange). On the other hand, SOA/ROA architectures are progressively gaining more momentum thanks to their flexibility and lower learning curve, which significantly improve system integration.

2.3.2 Service-Oriented Architectures (SOA)

Adopting the interaction principle of RPC, SOA also implements a communication model relying on messages sent to service endpoints. SOA however offer several qualitative advantages comparing to RPC architectures. First, flexibility is enhanced through the use of SOAP [299] (XML-based). This model intends to describe the exchange structure and some application-defined datatypes not included in the basic specification. Every request and response is formalised with SOAP, acting as a common application protocol. Then, the scalability is another improved feature over RPC as stateless communications avoid the server to store client-related information.

Messages contain all the required knowledge, so that requests can be processed more rapidly. The system is more robust with SOA as interfaces and endpoints are well-described. The also XML-based WSDL provides a semantic to describe endpoints in terms of service contracts. Furthermore, a lower coupling between clients and servers can be achieved with WSDL as clients can dynamically understand how a service can be consumed. This fosters late-binding between the components of a distributed system.

SOA has been the subject of an intensive specification effort leading to a complete application stack. It is encompassing additional functionalities, going far away from simple service consumption. One can for example mention WS-Discovery [300] providing techniques to dynamically find interesting services. Web services are also able to generate events by following the WS-Eventing [301] guidelines. Finally, WS-Security [302] addresses several critical points to ensure that Web services can be used in business applications where secure interactions are an important aspect. All these Web service extensions are standardised either by the World Wide Web Consortium, or by OASIS (also a not-for-profit consortium). SOA along with all its extensions have contributed in their acceptance as a reference architecture in distributed applications. We will refer to the global SOAP-based Web service architecture including all the extensions as `WS-*` throughout this thesis

2.3.3 Resource-Oriented Architectures (ROA)

ROA mainly differentiate from SOA by providing a finer granularity of the endpoints. The services are indeed directly linked with final endpoints, whereas this concept is different with SOA. Multiple services are available on a single endpoint, therefore lowering the flexibility. With ROA, every component acts itself as an endpoint by being individually addressable and exposing an uniform interface (API) [53]. This has a beneficial impact on the simplicity of use, making out of ROA an interesting alternative to SOAP-based Web services known to be rather difficult to handle [43, 124].

This architecture is the one used by the Web to provide content to clients. They can indeed directly manipulate data instead of focusing on the services that are wrapped around the data. ROA can therefore by qualified as more flexible. Developers interact in a straight way with data presented in its neutral format rather than being formalised with SOAP. Developers exposing their data to other participants define a limited set of operations that can be performed on the endpoints. According to the nature of the resource, these operations will seldom change.

Resource-oriented architectures may claim to offer more scalability and robustness thanks to the finer-grained level of interaction. These properties are the major reasons why ROA is increasingly used on the Web to expose services enabling access to data and functionalities. Following this approach, HTTP interfaces working according to the representational state transfer architectural style (REST) are a successful example and further outlined in Section 3.2.

2.3.4 Discussion

We have presented the most used architectural styles that target interoperability at different levels. RPC was the de-facto style in the early years of distributed systems when networks were composed of only a few machines installed and maintained by the same organisation. Their tight coupling was only slightly influenced by the performance and the scalability of the system. OOA are especially recommended only in special cases where applications are closed and all components are predefined in advance or rarely changing, besides being managed by the same entity. In more recent times, the Web has become a more important medium to exchange information between organisations, hence the popular growing of SOAP-based Web services along with WSDL, as they significantly improved interoperability.

While the Internet was growing and gaining more importance, companies felt the necessity to exchange data with each other. At this time, more loosely-coupled and open middleware solutions like CORBA and JINI were gaining momentum and are still used, as they improve interoperability between applications by offering language-agnostic APIs. Additionally, the protocol stacks and extensions are well described and offer a variety of features (some times complex) that range from security to discovery. XML and HTTP are natively supported by programming languages or can be included through libraries, allowing developers to build stubs to interact with specific services that remain valid as long as the WSDL file does not change. In spite of available tools helping developers to create services and automatically generate stubs, understanding SOAP-based Web services remains a non-trivial task.

SOAP-based Web services inherit another issue: being a legacy of RPC interaction styles, flexibility is not their key strength and are therefore not suited for dynamic environments where devices continuously appear and disappear. In addition, HTTP is employed by `WS-*` only as transport protocol to perform remote calls. `WS-*` are not considered as being part of the Web otherwise there would be no need for any additional API or descriptions, as the interaction would be realised through Web resources.

Feature	OOA	SOA	ROA
Granularity	objects	services	data (resources)
Main focus	marshalling	creation of payload	addressing (URI)
Addressing	object instance	unique endpoint	individual resource
API	language-specific	application-specific	generic
Pervasiveness	+	++	+++
Scalability	+++	++	+++
Loose coupling	+	++	+++
Cloud-likeness	+	++	+++
Ease of use	++	+	+++
Adaptive automation	++	++	+
Energy efficiency	++	+	++
Available tools	+++	++	+
Security	+++	++	++
Verbosity	+	+++	++

Table 2.2: Features comparison between the different architectural styles (+ means low, ++ means medium and +++ means high).

In Table 2.2, we reproduce partly a comparison of the various architectural styles that was initially proposed by Trifa [123] to show their differences and properties. In this table, we have added to the initial comparison our requirements for smart buildings outlined in Section 2.5. According to this table, we suggest the ROA style as de-facto architectural style for implementing a scalable and homogeneous infrastructure for building automation systems. This choice is relevant as long as applications do not require particular behaviours such as security or throughput. The loose coupling of ROA offers therefore many advantages and features useful for sensor networks. Still, as ROA tend to be simpler, they can more easily be directly used on resource constrained devices than SOA.

2.4 Middleware for Building Automation Systems

The principal challenge in networked embedded systems is the integration of mixed devices into one common application. For this purpose, middleware represent an interesting approach by reducing the gap between high-level requirements of pervasive applications and the access to basic functionalities of sensor networks [125]. Bernstein [126] defines it as *"a distributed system service that includes standard programming interfaces and protocols"*. A middleware is therefore a common ground to achieve interoperability when disparate components have to be integrated in a distributed system. In recent years, several works have been undertaken attempting to provide a middleware for heterogeneous building automation systems.

The SPINE [127] domain-specific framework allows rapid prototyping of sensor applications by providing a Java API which will handle all the discovery and offers access to sensors through coordinators. They use their own specified protocol to communicate with SPINE-compatible devices. Ducreux et al. [128] present an abstract data model to depict basic automation capabilities of a building by relying on a distributed tuple-space storage. The interactions defined by the model are kept to simple tasks such as read and actuate, in order to be easily integrable in applications with as less code as possible.

Several works have proposed to map field devices functionalities to Web services and therefore to ease the development of applications. Gateways for mapping the FS20 proprietary building protocol to RESTful APIs have been studied in [129]. The address of the end device is wrapped within the URL so that no mapping table is required. The standardised oBIX data model has been mapped to KNX by providing either SOAP-based Web services or RESTful APIs in [94]. An oBIX representation is included in the payload for each interaction, which results in a tight coupling. oBIX is also at the core of IoTSys in [130, 131] where it is extended not only to KNX but to all devices present in the building. Contracts representing device functionalities are built on top of oBIX, SensorML and DomoML, and are further used by developers to interact with endpoints. These contracts are pushed down to IPv6 field devices becoming an integral part of the network, and are directly addressable without requiring gateways. HTTP and CoAP are used to transport the XML-encoded contracts between the field devices. Further, some performance measurements have shown that the most efficient way to interact with the IoTSys contracts was achieved by using CoAP and compressing the XML with EXI [132].

Perumal et al. [133] have developed a 3-level architecture to rapidly compose applications relying on `WS-*`. A first module is dedicated to the discovery where accessible resources are registered before being made available. The service abstraction layer provides access to devices functionalities through generic Web services. The last module aims to build something comparable to mashups by reusing available services to create higher-level functionalities dedicated to the control of the building. A SOAP-based self-organising and adapting building automation system was prototyped in [134]. They took a top-down approach by modelling the process of building control with strategies serving the devices to automatically coordinate their tasks. Familiar et al. [63] focus on the composition of mashups from heterogeneous devices for smart spaces by relying on semantics to describe contracts. RDF annotations are converted to JSON and ease the development of composite applications by end users.

A framework for smart city applications relying on heterogeneous sensors is presented in [135]. Agents offer higher level RESTful APIs by relying on a set of Contiki OS based gateways having access to the network infrastructure. The SOS agents collect the data from the gateways and formalise them according to SensorML. Smart City [37] is an event-driven architecture to monitor public spaces with heterogeneous sensors. Knowledge processors compose higher level abstraction rules that can be used to build mashups for monitoring applications. The homogenisation of the platform is achieved with the help of semantics allowing to formerly describe devices and rules.

A building management system decomposed in two levels is presented in [106]. The field level is composed of sensors and actuators that are connected to base stations according to their location within the building or user-defined parameters. Sensor information is relayed to the base stations with a proprietary protocol, and then transposed to the application level through a set of APIs.

Cooltown [136] is one of the early projects considering people, places and things as Web resources. A new interaction approach using HTTP `GET` and `POST` to manipulate things was introduced. Lately, with the evolution made in embedded systems, it was possible to include Web servers directly on sensors and actuators. The WebPlug [137] framework introduces so-called mashups, relying on the Web of Things paradigm, where sensors and actuators play a central role, being part of the Web.

2.5 Requirements for Open and Scalable Smart Buildings

In a world where building automation systems will be formed around thousands of devices from a variety of networks, having a common ground shared among all participants seems obvious. Only little research has focused on the large-scale deployment of heterogeneous networks for smart buildings. As these become more widespread, so does the motivation to supply application developers with a means of sharing information between these networks. To achieve this goal, an open and scalable middleware solution targeting integrability and simplicity is crucial.

An architecture to connect wireless sensor networks and legacy building networks will also need to scale and evolve besides decoupling as much as possible all the subsystems. The central research question addressed in this thesis is shaped by these observations: *"how to merge heterogeneous networks to form a unified building automation system envisioning a global control?"*.

As it was shown in this chapter, diverse approaches to integrate heterogeneous networks have been already studied in the literature. Nonetheless, most of these solutions do not address simplicity and interoperability at a large scale, resulting into isolated and incompatible islands of functionality. Because of the characteristics of smart buildings, the focus of this thesis is rather on lighter and simpler building automation systems, where concerns such as scalability, integrability and efficiency prime over raw performances aspects. Such kind of applications can be described with the term *serendipity*, which can be defined as *"serendipity is the best explanation for Web mashups, in which the capabilities of unrelated Web sites are combined to create new sites that provide benefits beyond those that the original developers had intended or even considered."* [138]. This principle is a central aspect of this thesis approach, which seeks to simplify the integration and the use of building networks in order to facilitate the development of ad-hoc applications on top of many heterogeneous systems. In our vision, we expect that future smart buildings deployments will be composed of a variety of devices and networks, which will require a larger degree of flexibility in order to be manageable. Future deployments should pay attention to the *participatory* aspect, meaning openness and accessibility for users who want to easily reuse the components of the system in their own applications. In the same way, users could add new devices and automation strategies with minimal efforts by relying on a vendor-neutral application-level interoperability, achieved by reusing common and open standards.

Based on existing building networks and wireless sensor networks deployments in the real-world [93, 139, 140] [303], one can observe that the application layer can act according to two distinct interaction models:

Request-response model: Basic commands are sent during operation to devices to read sensors, change the state of actuators, or to retrieve application states. This is the central pattern used by conventional control loops (polling-enabled), for actuation purposes or user operations. For example, a user requests the actual temperature and decides to turn the heating on. The *get/set* model is the most common used interaction pattern. It usually follows a client-initiated cycle, where a server executes a command (request), and eventually returns a value or an acknowledgement (response). This pattern is also the core model used by HTTP, where users send a read request to retrieve a Web page, and then write the content of a form into a database. The request-response model should be preferred when values must be retrieved at a specific moment, for example in a static GUI.

Event-based model: Events are generated occasionally by sensors when some predefined conditions are met (e.g. as soon as the temperature varies by 1°C). In building automation system applications, devices monitor the physical surroundings waiting for particular phenomenon to occur (e.g. detecting presence), and then react to this by undertaking actions. Devices will send their measurements only if a phenomenon occurs and this as quickly as possible. Control algorithms and especially machine learning applications require to collect historical data and to store it in a sink for further processing. In such cases, an event-based model should be preferred to avoid useless redundant data transfers on the network.

In addition of bearing the two aforementioned interaction styles, an open and shared protocol envisioning interoperability for smart buildings applications must also have the following properties:

Pervasiveness by being lightweight enough to run on tiny and constrained daily-life objects and things in general. This means technologies having small memory footprints and limited parsing complexity should be privileged.

Scalability as potentially thousands of devices will be forming one building automation system. The application layer along with its implementations should not be a limitation. The number of achievable parallel connections has to be sufficient to perform all the control related operations.

Loose coupling by ensuring minimal influence on the overall system if network components or devices should be added or removed. The underlying subcomponents should be exchangeable in a transparent manner in order to guarantee a long-term evolvability.

Cloud-likeness by building the system upon existing infrastructures and reusing computational capacities and memory storage of available devices such as sensors and gateways. Developers should be able to deploy higher level applications as a service into an in-network cloud, making abstraction of hardware and service distribution concerns.

Ease of use as applications developers will build unified control systems without specific knowledge of the underlying technologies and protocols. Composing applications from distinct systems should be kept as simple as composing Web 2.0 mashups [141].

Ease of provisioning by minimising the installation effort, following the plug-and-play concept and therefore augmenting the user experience. This results in making the solution accessible for a large variety of people. No special skill should be required to run smart buildings aside from basic configuration steps affordable by any user.

Adaptive automation by providing an architecture compatible with novel adaptive algorithms such as machine learning. These algorithms are more complex than simple threshold rules and typically rely on significant sets of historical data. They are able to discover how a building is used and to react to environmental changes. The training and runtime processes of adaptive automation should be intrinsic components of next generation building automation system.

Energy efficiency as the purpose of building automation systems is to save energy, the protocol must not go against this achievement. As energy consumption is closely related to the number of messages carried by networks, traffic load must be kept as low as possible.

As such a middleware is intended to be deployed as the common application level, it should assure relying on open and royalty-free use of protocols and implementations. Open-source projects like the Apache Web server or the Hadoop database are compelling examples of success stories where openness, code sharing and reuse are key features playing a central role in the wide adoption of any technology. The performance should be similar to other currently used approaches, or at least sufficient to cope with the increasing amount of heterogeneous devices.

In many situations, and especially in the field of smart buildings, applications can benefit from being part of the Web, for example to share data among devices or to access the properties of a device via a Web API. By naturally positioning the device API as full part of the Web, data can be easily reused into Web applications by relying on standard formats. Additionally, the devices become an extension of the Web, with as consequence their entire functionalities such as operations, sensing, actuating and descriptions being individually addressable in a uniform manner. The evolvability and interoperability of the resulting system will be greatly improved by using unified APIs instead of vendor-specific protocols.

Apart from these fundamental properties, various surrounding functions need to be supported to facilitate the development of scalable building automation systems: discovery of devices and resources description, service composition, standard data representation suited for traditional building networks as well as for novel IoT architectures, along with efficient eventing mechanisms. Finally, the whole middleware should be optimised in terms of communication overhead, therefore minimising its energy footprint.

2.6 Discussion and Summary

In this chapter, we outlined the requirements for a future open and scalable framework that fosters the integration of heterogeneous building networks (Section 2.5). We synthesised these requirements in the light of a review of emerging existing and standard technologies (Section 2.1, 2.2, 2.3, 2.4). In this section, we summarise our main findings and discuss the limitations that will be tackled in this thesis.

Early distributed applications relied on remote procedure calls (RPCs) to enable interactions between distant systems. Object oriented architectures are however not adapted to sensor networks because of their low flexibility having impacts on evolvability and scalability. Their tight interface contracts largely increase coupling and complexity, resulting in a strong binding between devices and specific application scenarios. Expensive protocol translations and data transformations are required to connect incompatible middleware, preventing the integration of sensors in various applications.

In the last decade, the Web has gained more importance whenever it comes to build applications on top of different technologies. In particular the `WS-*` stack comes with a variety of protocols ranging from the XML-encoded SOAP and WSDL up to security extensions. While they reduce the coupling between the components, these Web services middleware suffer from a particular drawback. HTTP is often only used as transport protocol in combination with custom application protocols (SOAP), while HTTP is an application protocol and does not require an additional layer. Moreover, most of the applications make use of a central repository where devices have to register before they can provide their services. The advent of resource oriented architectures (ROA) helped to solve some of these limitations. Devices are an inherent part of the Web and rely entirely on HTTP as application layer. However the presented middleware considering devices capabilities as Web resources tend to move away from this approach by using custom protocols on top of HTTP (for example oBIX), introducing a tighter coupling that prevents ad-hoc interaction with devices.

The Web not only consists of HTTP but encompasses a variety of fundamental components such as DNS, discovery services, and a set of data formats. While most projects use HTTP as federating protocol between heterogeneous networks, none of them questions the relevance and suitability of these orbiting technologies in smart buildings. The infrastructure supporting the Web is rather static and cannot evolve by itself without human intervention, whereas sensor networks have to deal with dynamic conditions. For example, an application-level infrastructure that envisions semantic discovery and search based on real-time sensor values has not been explored to the best of our knowledge. In the same perspective, the steps required to set up such an infrastructure within a building require expert-knowledge, preventing deployments in small-sized installations (i.e private households as apartments and houses). Approaches for incorporating the underlying infrastructure in a transparent manner have also been rarely addressed.

At first glance, the various projects and technologies available in the stae of research and presented here can appear as sufficient solutions for smart buildings. However, they only address a specific problematic of building automation (predominantly the field level) and neglect the other layers depicted in Section 2.2. No project is offering a global solution in a vertical manner that gives the possibility to build a unified control system starting with field devices up to control strategies and management tools. Moreover, the presented solutions propose only specific and closed implementations without taking a developer point of view. In this thesis we address these limitations, and propose the fundamental building blocks for a scalable and entirely Web-based infrastructure dedicated to heterogeneous building automation systems. In the next chapter, we discuss why the Web of Things architecture, leveraging on ROA, is an interesting candidate to create a participatory middleware.

Chapter 3

Towards Things-Oriented Building Networks

Contents

In the previous chapter, we have highlighted how smart buildings can benefit from integrating heterogeneous networks of sensors and actuators to build higher-level applications. Offering simple ways to share and to reuse devices and services at a global scale opens new dimensions where things[1] will play a central role. The Web is the most successful compelling example at building a large participatory network since now two decades, allowing users to retrieve and to share information. In this chapter we outline the reasons why it has been successful and how to transpose these benefits in the field of building automation systems that will be in the future and according to our hypothesis formed of Web-enabled things.

We start by motivating why the Web is an interesting paradigm to build scalable and distributed applications. We then outline the main contributions of the Web of Things and especially how

[1]In the rest of this thesis, we use the term *thing* to refer to Internet of Things devices.

the REST architectural style is used on smart things. In order to cope with the event interaction style of building networks, we present the main Web-based push techniques and discuss their use. Then, we introduce the emerging CoAP protocol along with its associated concepts gaining momentum in sensor networks. Finally, we conclude this chapter by discussing the implications and open challenges of a Web-based middleware for smart buildings that will be addressed in the next chapters of this work.

3.1 Relying on the Web

The Web has very rapidly gained importance when it has come to share information among users located all over the world. This is not a coincidence when we consider what are the technological foundations of the Web. It starts with a simple assumption that an easy-to-use protocol will be sufficient to carry hyper-linked documents, allowing to browse among them. These documents or resources can be hosted anywhere in the world and remain accessible using the same interaction style, independently from any hardware or software platforms. It is this loose coupling that made the Web so successful as the resources do not play a role on the integrity of the system. Resources can appear or disappear without the need to regenerate components before they can be consumed by Web browsers or other applications. Besides loose coupling and interoperability, the Web is a scalable architecture supporting billions of resources and keeps growing everyday as physical objects are being integrated. Additionally, the Web has a total independence to network topology and medium, which allows it to be ubiquitous.

The HTTP protocol lies at the core of the World Wide Web. It was originally designed as a substrate to build a distributed hypermedia system for linked multimedia documents [142]. As previously mentioned, scalability, robustness and evolvability are pushed at their highest level while the coupling is kept at its lowest. Historically, Web pages were used to present static content to the users. This content was usually managed by professional people such as webmasters directly in the source code. With technological evolutions, the way how the content is managed has evolved to a participatory Web, where the users and user-generated data have been at the centre of recent Web 2.0 applications. Many of them have become accessible through open Web APIs [143]. This opened new dimensions for developers to conceive Web mashups, which are applications combining content and functionality from various APIs. We can nowadays observe a clear trend to rely on Web technologies to build professional applications, even if they are not intended to be shared with the rest of the world.

The Web has been recently transformed in the most widespread and popular sharing platform and encompasses now other domains than the classical Web pages presentation. The five pillars presented below form the core foundation of the Web we know and use today. The Web of Things is at the intersection of those trends and positions itself as an evolution of the Web that reaches the physical world. We here summarise those five pillars according to the core foundation of modern Web applications as defined by Trifa [123] and discuss their implication in the context of smart buildings.

Social Web: The Web has evolved from offering only static content produced by webmasters towards a participatory network. Services such as blogs, forums and wikis make users supply the Web with new content and play a key role in the shift from a Web of pages to a Web of people. Tremendous various types of services are nowadays available on the Web to support virtual communities, as for example social networks like Facebook and Google+, up to authentication and identification tools as OAuth and OpenID.

This aspect is the less important in current smart building applications as no access to social networks or communities is required. However, we could imagine that in a near future, the management of a building could gather information from those services to offer even greater comfort. One could imagine to adapt the lighting according to the mood of the user.

Real-time Web: In a society where information is relayed rapidly, this trend does not pass besides the Web. Many domains such as finance, news or social networks rely on real-time systems to exchange large amounts of information in a robust and quick manner. Several tools and techniques have appeared offering ways to transmit timely information over the Web. However the traditional poll-based approach of HTTP being not adapted to deliver timely contents, other techniques solving this limitation have appeared and are further described in this chapter.

In a context of building automation, information has to be carried in a real-time manner in order to ensure a proper working of the system. As previously mentioned, most building automation systems rely on an event-based model to ensure the shortest possible delays for the interactions between sensors and actuators. The real-time Web has therefore its place in smart buildings.

Programmable Web: With the emergence of the concepts and technologies linked to the Web 2.0, the Web has become a novel development platform for applications. Web APIs open the access to raw data and services that can be reused in ad-hoc applications. Many companies offer a programmatic way to reuse components of an application instead of forcing the use of the finished product. We can cite as example the Google Maps API adopted on many websites to show mapping or geolocation content to users. These Web applications leveraging on services offered by various sources are called *mashups*.

Current automation strategies in form of simple rules (thresholds on devices) or complex control loops (centralised server-side) are rather hard to conceive and maintain. The programmable Web can open new doors by allowing rapid prototyping of new automation strategies. Furthermore, Web mashups lower the required knowledge so that even non-expert people can reuse existing components to build their own control scenarios.

Semantic Web: The Web relies on search engines crawling all the content to build indexes containing keywords. Although HTML provides tags to organise the text, it is rather difficult to retrieve a semantic meaning from the content of Web pages. To cope with this problem, machine-readable annotations embedded along or within the content are used. This metadata describes semantically the structure and the content of the documents, which further helps in the indexing and searching process. The resource description framework (RDF) [304] provides a set of elaborated semantic languages that not only allow to describe Web pages but any kind of resource.

In building automation systems, interoperability between manufacturers is often achieved by agreeing on data representations and exchange formats. The semantic Web can here contribute in this interoperability by describing this knowledge in a formalised manner around the notion of ontologies. One can imagine that datapoints and other functionalities would be standardised through semantic Web approaches.

Physical Web: The Web is no more confined to servers and desktop browsers. Smartphones and even things are able to interact with it. Plant sensors and fridges can push data into the Web (Twitter and other messaging platforms) to give insights about their states or what happens at the physical level. This shifts the content of the Web from human to machine-delivered informations. Furthermore, Web server software have now moved out of fully fledged computers (servers) to be embedded in smaller devices.

The physical Web is the most promising aspect of these pillars. Instead of field devices (sensors and actuators) communicating with dedicated protocols, they could rally around the protocol of the Web, namely HTTP. This would allow to achieve an interoperability between technologies as all would speak the same language. Moreover, applications could be developed more rapidly while avoiding to integrate each protocol separately.

3.2 The Web of Things

The Internet of Things allowing any kind of object to communicate from a network point of view (relying on IP), it does not specify the application layer. As a consequence, things remain incompatible to understand each other. The *Web of Things* (WoT) envisions to overcome this lack by pushing Web protocols at the application layer. As previously mentioned, it was initiated at the intersection of the current five trends of the Web. Guinard describes it as *"the overall goal of this architecture is to facilitate the integration of smart things with existing services on the Web and to facilitate the creation of Web applications using smart things"* [48]. To achieve this goal, they consider things such as sensors and actuators as core components of the Web by proposing the same interfaces that are used on classical Web servers. Things expose their capabilities (endpoints) in form of Web resources that seamlessly integrate into the Web. The HTTP protocol plays a key role by being shared as common ground protocol between all the participants, and acts as real application layer requiring no further specification to understand how to manipulate resources. Its widespread interaction style and all its related other tools and techniques used on the Internet such as caching, linking, search, authentication, or scripting among others are directly applicable to things. Thanks to its simplicity, HTTP is highly versatile and an omnipresent protocol with powerful and scalable server implementations that are often freely available as open-sources projects. Besides bringing some consistency at the application layer due to the loose coupling of stateless HTTP connections, it leverages most existing infrastructures that already support TCP/IP and lowers therefore the integration costs. This allows to push this paradigm down to devices and makes their interface available from all over the world instead of only the data they produce by means of sink applications.

The Web of Things praises Web scripting languages and frameworks as being the future of applications. Web applications are often simpler and faster to develop than classical desktop applications [123]. Current software approaches integrating the physical world and business applications are too rigid and closed to be tailored by end-users. A higher-level declarative approach rewiring and combining data could be used in the context of large ecosystems of networked devices, such as particular tools dedicated to the composition of Web mashups.

The Web of Things does not only describe how to use HTTP to make things Web-compatible, but is actually a set of decoupled and exchangeable building blocks to compose Web applications building on top of things. In this list, we present these blocks proposed in [48] by Guinard and discuss their suitability in the context of smart buildings:

Accessibility layer addresses the access to smart things from an application point of view. Things get integrated into the core of the Web and become first-class citizen of the Web like Web pages. The REST architectural style is at the heart of this approach and is used to expose the functionalities and services of smart things. This layer will be further described later in this section.

Transposing this layer in a building automation context would result in a standardisation of how the functionalities of sensors and actuators are offered. Instead of relying on dedicated protocols, all devices would share the same application protocols. It is however questionable how legacy devices already installed in buildings could participate in this Web network. As will be seen in this thesis, smart gateways are an interesting solution to include this kind of devices into the Web.

Findability layer proposes two methods to search for particular things or services among billions of them. It relies on a semantic description using the microformat formalism to describe properties and the concept of hyperlinks. First, things are made indexable by common search engines of the Web such as Google. Due to the rapidly changing dynamic context of things, schedule-based indexing of search engines is inefficient as the descriptions will be most of the time outdated in the index. In addition, as search engines follow links, new devices are not able to announce their existence. To solve this shortcomings, a lookup and registration infrastructure adapted to the particular needs of the Web of Things and building upon its description model is proposed.

Starting from a good intention, this layer proposed by Guinard does not meet the constraints of smart buildings as such. Due to the dynamic of sensors and actuators changing their context rapidly, search engines would have outdated representations in their indexes. Moreover, the proposed centralised repository would also require a frequent update of the descriptions, resulting in unnecessary traffic.

Sharing layer allows users to share their things among them and adds the notion of security. All this is performed by the social access controller (SAC) which allows users to advertise their things on social networks like Facebook or Twitter. They can manage to whom things should be accessible and fine-tune the nature of interaction (read-only or read-write). The authentication of users leverages existing infrastructures such as OAuth. This avoids managing access lists directly on devices. Only the SAC is authorised on the things by relying on the Digest authentication of HTTP along with SSL. The SAC acts therefore as proxy between users and things.

Although adding some security through access lists, this concept cannot be applied in smart buildings. The sharing layer indeed rely on social network to manage the available interactions between devices and users. Building automation should obviously not become dependent to such services. As interactions in building automation happen between machines, this would require that each thing is attached to a social network account.

Composition layer shows how smart things can be used to build Web mashups. A proposition relying on the Clickscript [305] graphical mashup editor is particularly detailed. Clickscript is adapted in order to directly talk with things and integrate their data and services as graphical components, easing the development of applications. Users simply link components together to create a higher-level logic that will be further executed by the mashup engine.

Being able to rapidly define new automation rules without particular knowledge is particularly interesting as current approaches require expert knowledge. This layer will be redesigned in the framework of this thesis in order to provide a simple graphical application enabling end-users to define their own control scenarios. We will however emphasise on the seamless integration of automation devices into the application.

In this section we describe the underlying concepts of the Web of Things paradigm by focusing on the accessibility layer which allows to homogenise the application level between devices coming from different backgrounds. As this paradigm solves many shortcomings when integrating heterogeneous devices in a global network, it will be the foundation of our proposition in this thesis. Meanwhile we will consider the Web of Things from a smart building point of view and adapt its components towards a Web of Buildings, especially tackling the vertical integration of building automation systems within the Web.

3.2.1 Representational State Transfer (REST)

In this section, we give a brief introductory overview on the main concepts of REST (Representational State Transfer). This architectural style was originally proposed by Roy Fielding in his PhD thesis [52]. His proposition does not focus on specific technologies or standards, but describes an architecture that aims at increasing interoperability and reducing the coupling between components of distributed applications. REST should be more considered as a set of constraints and guidelines facilitating the architectural composition of distributed applications. This is the reason why it is the underlying pattern of the Web, which is a word wide distributed application. Following the REST guidelines, HTTP has become the predominant protocol that lies at the heart of the Web. Web applications following the REST architectural style are called *RESTful* and potentially natively endow all the benefits of HTTP such as encryption, authorisation, authentication, caching or compression [123].

REST distinguishes from traditional `WS-*` Web services in different ways. First, it is characterised as a Resource Oriented Architecture (ROA) where endpoints are represented by resources rather than services. This aspect has a significant influence on the simplicity and fosters the ease of integration. It is this simplicity that avoids the necessity of additional description languages such as WSDL for generating stubs that would result in a more complicated structure and a tighter coupling. Then, HTTP is used as true application protocol and not only as transport protocol like with `WS-*`, where a marshalling is realised around the data with for example SOAP. This simplicity is also beneficial as clients can directly focus on the useful data rather than having to interpret the marshalling code. Relaying this in the context of building automation, the concept of representing functionalities as resources fits particularly well. Automation devices usually offer simple functionalities in a getter-setter manner that do not require any marshalling as the data (for example the state of an actuator) and not the service is the main concern.

The Web intends to create a large-scale distributed system of hypermedia documents, and Fielding describes it as *"REST provides a set of architectural constraints that, when applied as a whole, emphasises scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems"* [52]. This statement fits particularly well in the field of sensor networks where devices come from different manufacturers and target different functionalities. Moreover, the same applies in a broader manner to higher-level components that can be found in smart buildings such as control algorithms and management services.

As previously mentioned, REST does not specify how the components and protocols should by implemented, and only focus on the interactions among them. Furthermore, it provides guidelines on what are the various components of a distributed application, what are their roles and how they should behave with each other. For example, a broadly shared format should be used as interaction between resources by exchanging their representation. Fielding outlines that any distributed system claiming to be RESTful and therefore benefiting from its properties (performance, simplicity, scalability, portability and reliability) has to follow those five constraints [52]:

Client-server: A separation of concern is applied between clients and servers. They communicate over uniform interfaces allowing to reduce the coupling between the roles. A client will have no clue about the internal representation of the server-side that will hide the storing and processing mechanisms with data. In the same way, a server will not know the purpose of a request and what the client will perform with the data, as this could be used for a user interface or other purposes. Such a separation between data and control logic has a great influence on scalability and portability on the client code. This loose coupling enables the development of independent components, which is advantageous for heterogeneous building networks.

Stateless: Each request issued by a client to a server must contain all the necessary information to understand the request and perform the operation. The context of the client should not be kept on the server. Visibility, reliability and scalability are improved due to the fact that no context information are stored on the server. It can therefore execute operations more rapidly and can recover with no need to determine its previous state.

Cacheable: Being able to cache data improves network efficiency as it reduces the number of interactions. Clients or proxies will locally store the current representation of a resource. Servers can define policies when data expire and should be reloaded into the cache. This results in higher performance and increases the user experience by lowering the average latency.

Uniform interface: Loosely coupled interfaces are the essence of REST that emphasises on evolvability through the use of uniform interfaces between components. Clarity, simplicity and standard interfaces that can be easily extended to several components and various contexts have significantly influenced the adoption of the Web as a participatory architecture. This is a central aspect of the Web of Buildings as devices can be added and removed dynamically at any time without compromising the overall system stability.

Layered system: Component behaviour is restrained such that each component can only see its direct remote partner but has no knowledge beyond the immediate layer it is interacting with, promoting substrate independence. Layers can be built to encapsulate legacy services and to protect new components from legacy clients, as for example gateways to proprietary protocols. By following this concept, intermediary servers improve scalability and response time via load balancing services and shared caches or distributed content delivery networks.

The uniform interface is the most important of these constraints again according to Fielding [52] and should follow four principles that any REST interface has to implement:

1. **Resource identification:** Every component such as data or service that is relevant enough to be referenced in an application is referred as a resource. Each resource must have a unique identifier and has to be addressable separately through a unique referencing mechanism. Relying on unique identification and addressability increases simplicity, reusability and visibility. The Web uses the URI specification to comply with this requirement.

2. **Self-describing messages:** Each request or response is considered as a message and REST expects them to be self-descriptive. This means that every message holds all the required information to process the task. A constrained set of messages should be fully understood by the client and the server. It refers to the allowed operations (HTTP methods) on a resource. The name and meaning of the messages' metadata elements (HTTP headers) are also well-described and have to be understood by all parties.

3. **Manipulation of resource through representations:** REST decouples the representation of a resource from its identification. A resource that is identified by a unique URI can have many representations like HTML or XML. The client and server can negotiate the most appropriate representation. The specification of HTTP defines a set of media types. A client will inform the server about which types of representations it can understand by putting the list of media types in the *Accept* header.

4. **Hypermedia driving application state:** Clients can benefit from resource identification to link resources to each other. This allows following links contained in resources to interact with services. Every application state has to be addressable through its URI, and each resource should contain links to describe which operations are possible in each state and

how to navigate across them.

Simplicity, uniform interfaces, and widely available server libraries, as well as ubiquitous clients make out of RESTful services a truly loosely coupled architecture. No prior knowledge about how to use a resource is required as services based on RESTful APIs are reusable in an uncomplicated way. Using the REST architectural style on building automation devices and services provides many advantages, especially due to the native simplicity of this architecture. First, more scalable and robust applications can be designed due to the loose coupling. Next, one can use all the Web-related tools and underlying technologies (scripting language, event notifications, markup languages, URIs, etc.) to rapidly prototype applications without having to define all the orbiting concepts. Finally, the resource oriented approach of REST naturally fits with the representation of functionalities of building automation devices like sensors and actuators where current values can be read or statutes updated.

3.2.2 Addressing Resources

Every element of an application offering functionalities has to be explicitly addressed (for example the temperature measured by a sensor, a stored data or a computing service) and is represented as a resource addressable using a globally unique identifier. This is a particular aspect of resource oriented architectures combined with data-centric applications. The well-known URI [144] standard scheme is used on the Web to offer a simple manner to identify resources. Using the same addressing scheme on things like all other resources accessible on the Web opens the door to seamlessly make device functionalities such as sensing and actuation be part of this big family. They can therefore be linked and shared as other traditional Web resources.

```
# The temperature in Celsius of a device
http://device.com/sensors/temperature/celsius

# The measure number 8360 from 2014-10-29
http://storage.com/measures/2014/10/29/8360

# The access authorisation of employee number 45
http://access.com/employees/45
```

Listing 3.1: URIs point to device endpoints represented as Web resources.

The examples outlined in Listing 3.1 show how final resources can be addressed (resources that have no children). However non-final resources having children can also be addressed. These type of resources are known as collections and visible in Listing 3.2.

```
# A list of the available sensing capabilities of the device
http://device.com/sensors

# All the measures from 2014-10-29
http://storage.com/measures/2014/10/29

# A list of all the employees
http://access.com/employees
```

Listing 3.2: Collection resources group several children of the same type.

This principle of mapping functionalities to URIs is further discussed in Section 4.2. In order to seamlessly integrate into the Web, each functionality of a sensor or actuator is represented by its own unique URI and becomes accessible via its RESTful API.

3.2.3 Exchanging Information

One constraint of REST is to decouple the resource from its representation. The abstract resources are therefore not bound to a particular representation. Moreover a resource exposed by a smart thing can be represented according to several formats. However defining a common agreed format can significantly reduce the complexity of distributed applications as no single content negotiations will be required. For this HTTP defines a simple mechanism allowing content negotiation. Clients can specify which representation of the resource they prefer. Using the *Accept* header field, a client can announce the desired format. This field is not restrained to contain only one format, but can hold several ones if specified by the client. In the same way, a server will announce the type of data contained in the payload with the *Content-Type* header field. We outline this behaviour in Listings 3.3 and 3.4 where these fields are used to drive the content format.

```
GET /measures/45 HTTP/1.1
Host: storage.com
Accept: application/xml

200 OK HTTP/1.1
Content-Type: application/xml

<measure>
   <value>23.1</value>
   <unit>Celsius</unit>
   <date>2014-10-29</date>
</measure>
```

Listing 3.3: Response with XML payload.

```
GET /measures/45 HTTP/1.1
Host: storage.com
Accept: application/json

200 OK HTTP/1.1
Content-Type: application/json

{
   "value":23.1,
   "unit":"Celsius",
   "date":"2014-10-29"
}
```

Listing 3.4: Response with JSON payload.

The Hypertext Markup Language (HTML) is currently the most used representation format on the Web for exchanging information. It defines a set of tags that are used by the Web browsers to show the content in a graphical manner to end users. Further, it allows users to navigate amongst resources by clicking on hyperlinks. Although being a well-defined and wide-accepted format, HTML is not suited for machine interactions, as many of its semantics are related to the presentation.

For machine-to-machine (M2M) communications, other formats are better adapted, such as XML and JSON, which are widespread in service communications between enterprises. While XML is largely adopted as the common format for `WS-*` Web services, it presents a few drawbacks with regards to verbosity and integrability. As JSON is more lightweight than XML in terms of message size and parsing time [145, 146], it is recommended as shared format for smart things having limited memory and processing capabilities. The XML payload in the above example is already 60% more verbose than the JSON representation, and this without using arrays that could further reduce the size. Another argument in favour of JSON is that it can directly be transformed in JavaScript objects in Web applications [147], which makes it an ideal candidate to compose Web mashups including physical things. More recently, compression techniques like EXI (Efficient XML Interchange) have appeared in order to reduce the overhead footprint of XML by encoding data in binary format [148]. This requires however an additional step of processing before data can be shared or interpreted, and is therefore not adapted for smart things.

A large variety of clients and applications can interact with services in a consistent way when a flexible control of data formats is provided, which has a positive influence on the coupling. Rigidly-specified or newly-invented formats should be strongly avoided. Low-semantic data formats on the contrary should be favoured as they increase scalability and loose coupling.

3.2.4 Interacting with Resources

Uniform interfaces exposed to clients reduce the coupling between operations and their implementations. They specify contracts between the client and the server, which is based on the identification of a resource and the operation to execute. Each resource must therefore support a common set of operations specified in terms of semantics and behaviour.

Operations

HTTP provides a set of operations (also called verbs or methods) that can be executed on resources. Only five of them are mostly used on the Web and follow the CRUD (Create, Read, Update and Delete) interaction style. These interactions are often used in resource oriented applications, as for example when interacting with databases where SQL provides also the same set of operations (Insert, Select, Update and Delete). Some of those HTTP verbs are qualified as *idempotent* when performing several times exactly the same operation does not have an unexpected consequence and the result always stays the same. Other methods can be considered as *safe* if the representation of the resource is not altered by the execution of the operation. This applies especially to read-only interactions. We here describe those five main operations offered by HTTP:

OPTIONS is probably the less known operation specified by HTTP and is implemented by almost all servers. It can be used to retrieve the list of operations that are supported by a particular resource. This functionality is very useful in dynamic ad-hoc applications as clients can discover on the fly how they can actually interact with a resource. It is idempotent and safe as well.

Example: discover the list of supported operations on a light switch (`http://light.com/switch`).

```
OPTIONS /switch HTTP/1.1
Host: light.com

200 OK HTTP/1.1
Allow: GET, PUT, OPTIONS
```

GET is a read-only operation used to retrieve the current representation of a resource. It is both idempotent and safe.

Example: read the current temperature of a sensor (`http://sensor.com/temperature/celsius`).

```
GET /temperature/celsius HTTP/1.1
Host: sensor.com

200 OK HTTP/1.1
Content-Type: text/plain
Content-Length: 5
21.45
```

POST enables to create new resources whereas the URI of the newly created resource cannot be determined in advance. The server will define itself a URI for the resource and returns it in the response header. This method is non-idempotent and unsafe.

Example: add a new temperature measurement in a database. This results in the creation of a new resource identified by the *Location* field in the response.

```
POST /temperatures HTTP/1.1
Host: database.com
Content-Type: text/plain
21.45

201 Created HTTP/1.1
Location: database.com/temperatures/2701
```

PUT can be used to update an existing resource or to create a new one. The client knows in advance the URI of the resource it wants to interact with (update or create). This method is considered as unsafe but idempotent. Sending the same `PUT` message several times will not have any effect on the underlying service (the resource will only be created once or updated with the same value).

Example: update a resource that will have as consequence to switch on a light (`http://light.com/switch`).

```
PUT /switch HTTP/1.1
Host: light.com
Content-Type: text/plain
ON

200 OK HTTP/1.1
```

DELETE allows to remove a resource while being an unsafe but idempotent method. Sending several identical messages will not have any consequence as the resource will already have been deleted.

Example: delete a temperature measurement in a database (`http://database.com/temperatures/2701`).

```
DELETE /temperatures/2701 HTTP/1.1
Host: database.com

204 No Content HTTP/1.1
```

The fact of restraining the possible operations to atomic ones greatly simplifies the system architecture besides increasing the comprehension of the interactions. This is however a limitation for complex workflows of business distributed applications usually working with more advanced operations. In the context of sensor networks, it is actually an advantage as devices are proposed by a large variety of manufacturers. The limited set of operations improves the interoperability as only few interactions must be implemented to guarantee compatibility. HTTP contributes in this direction by being well-described and due to the important number of free implementations that are available. Developers can focus on the data rather than spending a lot of time on programming the interactions. This is a compelling example for a loosely coupled system. Scalability is improved as the interactions are predictable and new devices can rapidly integrate the distributed application.

Status Codes

HTTP also defines status codes giving insights on the result of the operation such as success, errors or exceptions. Each response contains in the header a field indicating the status of the operation according to well-defined codes. Many of such codes are standardised and have well-known meanings for HTTP clients as described in the HTTP 1.1 specification [149]. The use of those status codes in the context of RESTful applications has been covered in [53]. The most common status codes used with HTTP are:

200 OK The request has succeeded.

201 Created The new resource has been created.

204 No Content The request was successfully executed but no information has to be returned.

401 Unauthorized The resource requires user authentication or the provided credentials failed in authorisation.

404 Not Found There is no resource associated with the provided URI.

405 Method Not Allowed The resource does no support the expected operation.

500 Internal Server Error The server has encountered an internal error preventing from executing the operation entirely.

3.3 Pushing Data from Things

REST and particularly HTTP are intended for client-server communications, where clients explicitly request a resource (pull) and receive a response containing its representation. While this interaction style is well suited to control things, this client-initiated approach is not adapted to the dynamic of smart buildings where sensor measurements drive actuators in an asynchronous manner.

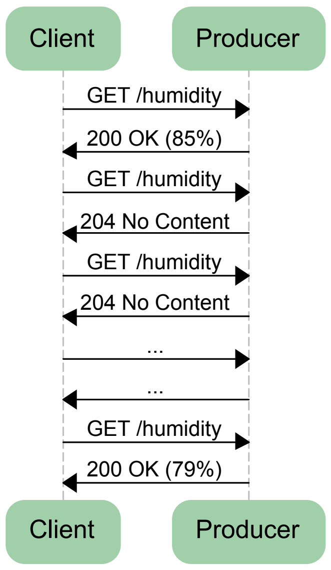


Figure 3.1: Sequence diagram of notifications between a client and a smart thing producing data. It follows the traditional HTTP pull interaction where the client constantly sends a request to gather the latest measurement.

As presented in Section 2.2, building automation systems use mostly event-based notification mechanisms to exchange measurements between sensors (producers) and actuators (clients). If we consider a HVAC controller using a humidity sensor to optimise the air quality, with HTTP 1.1 the client will have constantly to send requests to the producer (polling), as shown in Figure 3.1. In the best case, the producer will after a while respond with *204 No Content* responses if supported. In most cases the producer will however respond with the full representation of the resource. Such a behaviour is critical for smart building applications: first it can cause a network congestion as a tremendous amount of requests and responses will be carried on the network, knowing that the majority of these messages will be useless. This is even more problematic if considering a large-scale installation of thousands of devices. Besides being poorly scalable, such an interaction style is not suited to control a building as some measurements could be missed if the polling frequency is badly chosen (relatively low to reduce the network footprint). It will potentially cause to miss some short-lived critical conditions that should have generated alarms. On top of that, sending a large quantity of HTTP messages has a significant consequence on the energy consumption of devices that are potentially battery-powered [150] [15, 16].

Although HTTP allows to simulate real-time notifications by polling resources at high frequency, it is not adapted to the nature of building automation systems where a protocol to send asynchronous notifications as soon as a sensed value changes is a must. In this section we present the currently most used techniques to make out of HTTP an event-capable protocol along with the MQTT mechanism and discuss their application in our context. A detailed study focusing on the energy consumption aspect of several Web notification techniques is presented in Section 5.2.

3.3.1 HTTP Long Polling

A first technique to simulate real-time notifications with HTTP is called *Long Polling* and relies on delayed responses from the producer, as shown in the left part of Figure 3.2. It is not a true push but a variation of the traditional polling [151]. The client will issue requests as it would do in a classical polling approach but at a much lower frequency. If the producer has no new value to send to the client, it will keep the connection open until a new information is available. When this event arrives, the producer immediately sends a HTTP response to the client and closes the connection. After having received the response, the client restarts the procedure by sending a new request.

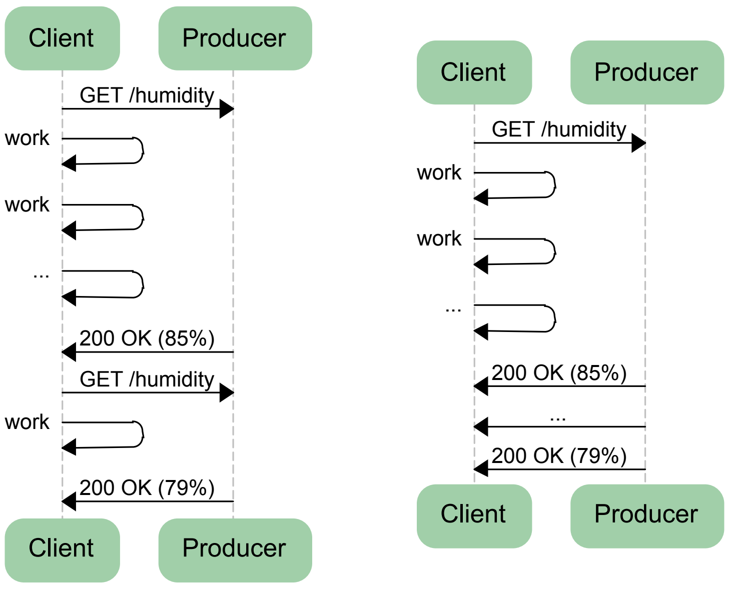


Figure 3.2: Sequence diagram of push techniques using HTTP. On the left *Long Polling* delays the HTTP response while there is no new value, therefore decreasing the polling frequency. On the right HTTP Stream keeps the connection open as long as possible, allowing to send HTTP responses only when new data is available.

HTTP Long Polling however induces the most drawbacks of all techniques:

Header overhead: Every poll request and response is an entire HTTP message including all the related headers in the message framing. For messages containing only few payload amount like it is the case for sensors (only one measurement representing a few bytes), the headers can represent a large percentage of the overall message size, resulting in inefficient communications.

Maximal latency: After having responded to a poll request with a new value, the producer has to wait for the next poll request before sending a new value to the client again. The maximal latency is therefore equivalent to three messages (poll response, next poll request and response). During this time, it could happen that a new value is available that will not be sent to the client. This can cause the control system to delay or to miss critical values that would have triggered an alarm or a specific reaction.

Timeouts: Responses are delayed until a new value is available and impose the connection to remain open. As HTTP uses TCP as underlying transport protocol, the maximal time a connection can be kept open cannot be known in advance as this depends on the client and server implementations as well as on the network equipment.

3.3.2 HTTP Streaming

The right part of Figure 3.2 shows the *HTTP Streaming* technique. It is an evolution of the Long Polling and can be considered as true push. The client sends as usually a HTTP request to the resource that is observed and the producer responds with the current value except that, in this case, it will not close the connection [151]. The producer keeps the connection open so that it can immediately send out HTTP partial responses as soon as the value changes. Two different ways allow a producer to signify that the connection will not terminate. The first one is by using the chunked transfer encoding. The second one uses a special MIME type (`multipart/x-mixed-replace`) which tells a client that updates will be sent using the same connection. No method is more advisable than the other.

Although reducing the network traffic with regards to Long Polling, this technique is also susceptible to timeouts with these additional issues:

Network intermediaries: Intermediaries such as proxies and gateways can be involved in the transmission of a HTTP response to the client. They are not forced to immediately forward a partial response and it is legal for them to buffer the entire responses before sending data to the client. HTTP Streaming will therefore not work in environments where such equipments are present.

Client buffering: The existing HTTP 1.1 specifications do not require that partial responses are immediately delivered to the application by the client library. Some implementations require a buffer overflow before partial responses are forwarded to the application in order to empty the buffer. Such behaviour is therefore not predictable and depends on the client implementations.

3.3.3 HTTP Callbacks

HTTP Callback (also called Webhook) is the most straightforward technique for machine-to-machine communications. Each participant acts as a REST server, which is by nature not compatible with client applications (Web browsers) unless using dedicated libraries. The client subscribes to a resource by sending a `POST` request to the `/subscribers` sub-resource in order to add itself as a subscriber, as shown in the left part of Figure 3.3. The callback URL that will be used by the producer is contained in the payload of the request. Each time the value changes, the producer will send a `PUT` request to the client's callback resource. The client will acknowledge the reception of the new value with a `200OK` status code response. From a broader view, this mechanism implements the event-based model by diverting the original use of the request-response model.

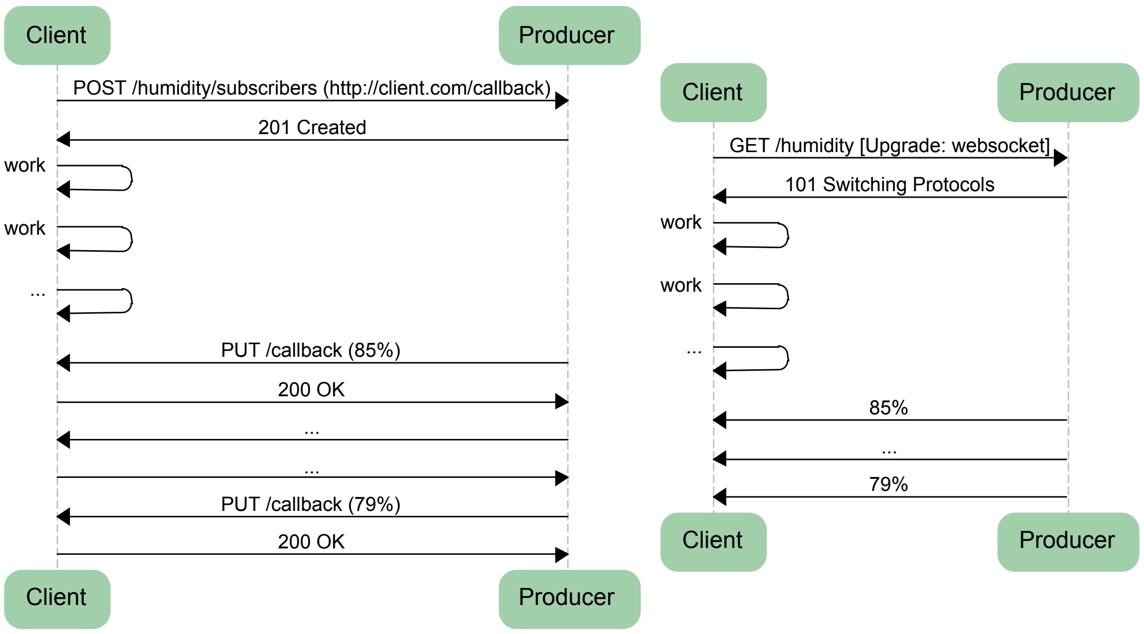


Figure 3.3: Sequence diagram of push techniques using HTTP. On the left the *HTTP Callback* transforms the client into a server exposing a callback resource to the producer. On the right HTTP is only used to initiate a *WebSocket* that will further switch to a bidirectional TCP connection.

Although it solves the problem regarding the timeouts of non-closed connections, it comes however with several other concerns:

Header overhead: Although the client does not make any polling and registers only once, each notification is composed of a HTTP request and a response. When looking at the few bytes (value of the measurement) that are useful data in a notification with regards to the overall message size, the communication can therefore be considered as inefficient.

Tight coupling: During the subscription process, the client has to address a specific sub-resource dedicated to the subscription of clients. The name of this sub-resource being not defined in any specification, this naming is left to the developers' discretion. While an agreement about this name is simple to achieve in small-scale applications, it becomes quite hard to accord devices coming from different manufacturers. It results in a tighter coupling between the devices.

Network intermediaries: As every client becomes a server, connections have to be opened in the reverse direction, from the producer to the client. This can be problematic if clients are located behind some firewalls blocking the callback requests.

3.3.4 WebSockets

The *WebSocket* protocol [152] is standardised by the IETF and enables two-way communications between clients and producers. It consists of an opening handshake relying on HTTP, as shown in the right part of Figure 3.3. The client sends a `GET` request to the resource and specifies that it wants to upgrade to WebSocket in the header (`Upgrade:websocket`). The use of HTTP ends at this point and the remaining of the communication is realised with the WebSocket protocol having a less smaller header. Each time a new value is available, the producer will encompass it in a WebSocket message. WebSockets are an inherent part of *HTML 5* and are supported by a large variety of Web browsers, allowing to create dynamic applications using duplex communications, increasing the user experience.

Although this technique reduces the amount of exchanged payload, it also comes with several drawbacks:

Tight coupling: Depending on another protocol for event-based communication, WebSocket in this case, devices are required to implement more than one application protocol. With regards to the REST definition, the application protocol should be self-sufficient to realise all the related application tasks. Besides offering loosely-coupled interactions, this increases the interoperability between components as the probability of using the same protocol is higher.

Timeouts: In the same manner than other techniques keep the TCP connection open, timeouts can occur at several locations of the network. To prevent unexpected closings, WebSocket uses a ping mechanism which has as side-effect to generate traffic not carrying useful payload.

3.3.5 MQTT

Unlike the other mechanisms previously presented, *MQTT* (Message Queuing Telemetry Transport) is a message delivery protocol enabling a publish/subscribe messaging model in a lightweight manner. The version 3.1 has been submitted by IBM to the OASIS specification [306]. It follows a one-to-many message distribution approach where the messages are agnostic to the content of the payload. The overhead of the protocol is kept as low as possible with only two bytes of fixed header. Publishers will feed a topic (a kind of data flow) on a broker that will further forward the notifications to the subscribers. To do so, a subscriber has first to register on the topic it wants to follow, as illustrated in Figure 3.4.

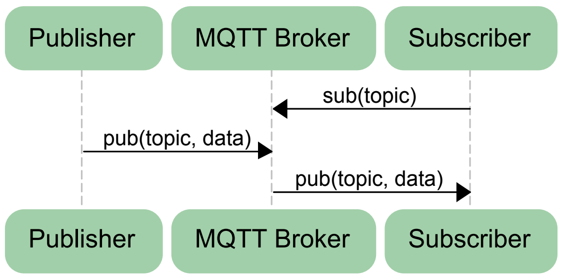


Figure 3.4: Sequence diagram of the MQTT notification technique. Subscribers announce their interest to a broker that will forward the notifications sent by publishers.

MQTT uses TCP as underlying transport protocol and requires therefore one connection per client, which can be either publishers or subscribers. The clients hold the connections to the

MQTT broker open at all times. As this is not compatible with constrained devices residing on sensor networks, an even more constrained version has been developed (*MQTT-SN*). It differs in several points to the traditional MQTT. First, it relies on UDP (instead of TCP) which is better suited for lossy networks. The topic names that can be rather long strings are replaced by a short 2-bytes long so-called topic id. Further, clients having not been configured with a gateway can discover the network for available ones.

As MQTT and its sensor network version are naturally not compatible with each other, gateways are required to translate the protocols. As shown in Figure 3.5, two different gateway approaches are offered. The transparent gateway will only perform syntax translation between the protocols. As all interactions are end-to-end between the MQTT-SN clients and the broker, all the features of MQTT can be offered to the clients. This solution meanwhile requires the gateway to manage a separate TCP connection for each client on the sensor network, which can badly influence performance. On the other hand, the aggregating gateway only handles one TCP connection to the broker. All messages sent by constrained devices end at the gateway. The gateway then takes decisions about which messages should be forwarded. This solution being much more complex to implement is however more adapted to networks with a large number of devices.

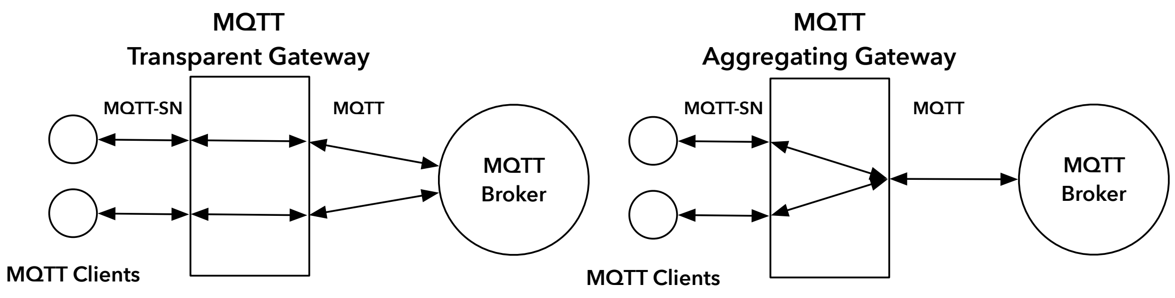


Figure 3.5: Gateways are required to interconnect the classical MQTT protocol with its sensor network optimised version. On the left the transparent gateway translates the protocol allowing end-to-end interactions. On the right the aggregating gateway only uses one connection to the broker and reformats the messages.

As MQTT uses TCP as underlying protocol, it comes with the same drawbacks as WebSockets. However MQTT has also other issues than timeouts:

Tight coupling: MQTT is not an application protocol but dedicated to event messaging. It therefore tightens the coupling in the same manner as WebSockets. Additionally the different versions of MQTT require gateways to achieve interoperability.

Maximal latency: As publications (notifications) are first sent to a broker that will relay them to subscribers, the maximal latency is increased. This latency increases even more when using a gateway that has to translate the protocols before the broker receives the notifications.

Centralisation: The mandatory brokers handling the forwarding of notifications to subscribers represent a single point of failure and are not compatible with the distributed approach praised by the Web.

3.3.6 Discussion

We have presented several mechanisms allowing HTTP to push data from things. As HTTP is a pull-based protocol following the request-response model, it does not offer a native push functionality. In order to be able to push data, its initial use has to be adapted by introducing techniques avoiding traditional polling. The first technique is to keep the TCP connection open as long as possible. HTTP responses are sent over the connection as soon as the value changes. Meanwhile reducing the network traffic and potential congestions, this technique is not recommended for sensor networks, particularly if devices are connected to mesh networks such as 6LoWPAN [153, 154]. This is the main argument why we will not further consider HTTP Long Polling and HTTP Streaming in the context of this thesis.

HTTP Callbacks (Webhooks) avoid the timeout problematic by transforming notifications in pull requests. Clients also need to act as servers by exposing callback resources. As this constraint is not an issue in machine-to-machine applications, callbacks appear as an interesting mechanism to send notifications with HTTP. The main drawback is related to the name of the sub-resource (for example `/subscribers`) that can differ for every device and prevents clients from subscribing dynamically when having not been preconfigured. In Chapter 4.1 we will present a Web semantic based technique overcoming this limitation by allowing clients to dynamically discover the subscription sub-resource.

HTTP being not natively conceived for notifications, other protocol targeting a better integration have emerged in recent years. Websockets use HTTP only for handshaking and then switch to their own message protocol on a bidirectional TCP socket. While presenting the same timeout issues as other techniques, they can be natively integrated in HTML 5 Web applications and are therefore recommended for things that are intended to be directly accessible from end-user Web applications. Both MQTT and its optimised version for sensor networks MQTT-SN require brokers between publishers and subscribers, preventing direct communications between the parties, in addition to be not resource oriented. The failure of such a broker will prevent things to receive notifications and therefore risks to paralyse the building automation system. As this issue has a large consequence on building control, MQTT will not be considered in this thesis.

The aforementioned techniques to push data from things all comes with a certain number of issues either related to timeouts, header overhead or centralisation. In the next section we will present the more elegant push model specified by CoAP that is especially tailored for sensor networks.

3.4 The Constrained Application Protocol (CoAP)

After having described the core design of the Web of Things and its related notification mechanisms, we now show how these concepts can be relayed at the smart building level composed of native IP devices. Despite the fact that HTTP is an example of REST protocol having great success on the Web, it was not designed for sensor networks and particularly not for constrained devices composing them. This has mainly two backgrounds: first the underlying TCP protocol is connection-oriented and therefore not recommended on mesh networks having generally a non-negligible amount of packet loss. The retransmissions will increase the traffic while also augmenting the maximal latency of the network. Second, the HTTP protocol is itself not optimised for constrained networks as all the headers are text-encoded. Due to these shortcomings, HTTP is not a good candidate for sensor networks. This is why we present CoAP, which is a HTTP-similar protocol but conceived for constrained environments. We then conclude this chapter by discussing the actual limitations of the Web of Things when used to build scalable and sustainable smart buildings applications.

The Constrained Application Protocol (CoAP) has recently become a RFC managed by the IETF [155]. It is a specialised Web application protocol to be used with constrained nodes and lossy networks, especially for machine-to-machine applications. Nodes with 8-bit microcontrollers and limited amount of memory, as well as lossy and low-throughput wireless sensor networks (for example 6LoWPAN) are particularly addressed by CoAP [156, 157]. CoAP is very similar to HTTP and provides the same set of key-concepts of the Web like a request/response interaction model, resource orientation, URIs and media types. It can also very easily be interfaced with HTTP for integration with classical Web applications while conforming with specific requirements of constrained environments such as multicast, very low overhead and simplicity. Moreover, CoAP allows to realise a REST architecture with only minimal efforts.

Unlike HTTP, CoAP relies on UDP as underlying transport protocol that is best suited for lossy networks, but with the trade-off to offer no reliability in terms of TCP acknowledgements. To circumvent this limitation, CoAP includes its own application-level reliability. Messages can be set as confirmable or non-confirmable and will require an acknowledgement in the first case. The notion of reliability is decoupled from the request/response model as a server receiving a non-confirmable request can decide to reply with a confirmable response that will have to be acknowledged by the client. Acknowledgements must not necessarily be sent as separate messages. If a server receives a confirmable request while the data is immediately available, it can piggyback the acknowledgement with the response. This principle allows to lower network traffic.

The CoAP message format is binary-encoded for most headers and options excepted the user-specified ones like the URI. This allows to be very constrained with a minimal achievable message length of four bytes, as shown in Figure 3.6. Every field that is not mandatory is considered as an option. The most used options are defined in the specification and are given a standard number. For example the `Uri-Path` is defined as option number `11` and encodes each segment of the URI individually so that the server does not need to parse it. This list can be completed with specific options according to developers' needs and submitted for approbation to figure in the standardisation.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver| T |  TKL  |      Code     |          Message ID           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Token (if any, TKL bytes) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Options (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 1 1 1 1 1|    Payload (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3.6: A CoAP message can be in the best case only four bytes long, as represented in [155].

The mandatory headers along with the token are the followings: *Ver* indicates the CoAP version for compliance. Actually only version one is defined. *T* defines the type of message that can be either confirmable, non-confirmable, acknowledgement or reset. *TKL* specifies if a token is present in the message and its length. *Code* indicates the type of method in the case of a request. CoAP only implements a limited set of operations that are the most used on the Web (`GET`, `POST`, `PUT` and `DELETE`). In the case of a response, it contains the response code specifying whereas the request succeeded or if an error occurred. Like for options, the semantics of the value is described in the specification and one can also add its own method or response codes. These self-defined properties will however only be valid in the scope of the application and not understood by other devices. *Message ID* allows to detect message duplications. *Token (optional)* is used to correlate requests and responses.

3.4.1 Observer Pattern

A method for pushing data from devices is specified in an extension of CoAP named *Observe* [158]. It allows to observe resources in a very similar way to HTTP Streaming. Its main difference lies in the transport protocol. As CoAP uses UDP, there are no connection timeout concerns.

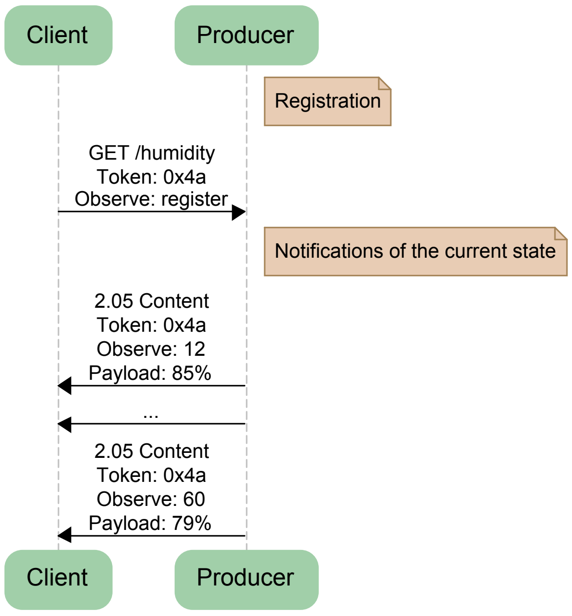


Figure 3.7: Clients can observe resources and be notified with a CoAP response containing the new value as soon as an event occurs.

A client starts by announcing its intention to observe a resource. As visible in Figure 3.7, this is realised by sending a `GET` request to the resource with the specific *Observe* option. The value of the option can have several meanings as follows: `0` adds the client as observer, `1` removes the client and any other value is used to reorder the notifications (sequence number). This requires the value to be incremented for each notification. The token is very important as it allows a client to match the notifications with the initial request, and therefore associating the value with the resource.

The producer will respond to the initial request with the current value of the resource. If it supports observation, then it will include the *Observe* option in the response with an appropriate value. However if it does not support observation, it will signify this to the client by omitting the option. Every time the value changes, the producer will send a CoAP response message to the client. These responses can be set as confirmable to enable reliability and to ensure that the client has received the new representation.

This mechanism presents several improvements over the other Web push techniques. First, unlike MQTT, the notifications are directly sent to the clients, which decreases the maximal latency. Not relying on TCP eliminates the timeout issue and is compliant with lossy wireless sensors networks like 6LoWPAN. Notifications being response messages of only a few bytes increase the overall network efficiency. Finally, the observer pattern is natively included with the CoAP protocol which lowers the coupling of the architecture, whereas the HTTP techniques are only propositions and not standardised.

3.4.2 Group Communication

As devices often naturally operate in group formations, the CoAP *Group Communication* specification [159] defines how CoAP should be used in a group communication context. IP multicast is used to communicate simultaneously with several endpoints (one-to-many). This allows a CoAP client to read or update resources with a single message instead of addressing each server separately (unicast). In the context of building automation, this approach can be used to drive actuators simultaneously (for example a light switch controlling several lights in a room), as shown in Figure 3.8.

Multicast becomes interesting to use even with Wi-Fi and 6LoWPAN mesh networks since dedicated routing protocols optimise the forwarding only to devices listening to multicast groups. This significantly reduces the network traffic and energy impact of multicast. For example, the Internet Group Management Protocol (IGMP) [160], the Multicast Listener Discovery (MLD) [161] and the Multicast Group Membership Discovery (MGMD) are commonly adopted protocols. Primarily used in mesh networks, the Protocol Independent Multicast (PIM) [162, 163] is a collection of multicast routing protocols, each optimised for a different environment relying on the existing network topology to take routing decisions. By relying on such protocols, multicast messages will only be forwarded to devices having previously announced their belonging to a group.

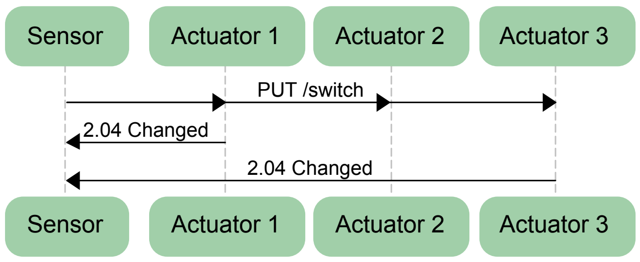


Figure 3.8: With CoAP group communication a client can simultaneously interact with remote endpoints. The sensor will be able to drive the actuators number 1 and 3 as they are in the same group. Devices that are not participating in this group will discard the request and do not respond (actuator 2).

A group is defined as a set of endpoints (resources) that are enabled to receive group communication requests sent to the group's associated multicast IP address. The responses to multicast messages are always sent back in unicast mode to the originating IP address. An endpoint is not limited to one group but can be part of several, whereas group membership can dynamically change over time.

Group communication requires to follow certain rules in order to work properly. The specification recommends to always use the same port for groups (CoAP default port) and to vary the IP multicast address to distinguish groups. The most delicate constraint is related to the URI-path of the resources that has to be the same in one group. For example group communication will not work in a group where one device has a `/switch` URI-path while a second one has `/change`.

The interaction model must be adapted to the context of group communication. First, only the idempotent methods are recommended for use (`GET`, `PUT` and `DELETE`). The problem with `POST` is related to the lossy nature of wireless networks where some servers may not receive the request. The client can obviously send several times the request to ensure its reception by all participants of the group. Meanwhile this will have as side-effect a probable inconsistency as the servers having received twice the request will have created two new resources instead of one. Secondly, as a client not necessarily knows the group members, it is not possible to include reliability in group communication. A client can indeed not predict from which servers it should receive an acknowledgement or a response, as using a multicast address impersonates the interaction. Moreover, group communication and observation are mutually exclusive according to the specification because of this limitation.

While CoAP Group Communication makes a step towards the interaction model used in building automation systems, it is currently not deployable as such due to the following limitations:

Poor interoperability: The URI-path of the resources participating in group communications have to be pre-configured, which avoids using devices coming from different manufacturers. This is the most serious limitation as it is very unlikely that manufacturers will agree on URI-paths. Group communication is therefore only adapted to infrastructures where devices can be pre-configured by a single party.

No reliability: As group participants can be added or removed dynamically, a client does not have any clue about which servers are currently participating in a group communication. Therefore it is unable to determine from which server it should receive a response and cannot ensure proper reception of requests. This is particularly disadvantageous in building control as a command from a sensor can potentially never arrive at the actuator due to lossy network conditions.

No notification capability: Being able to notify several observers simultaneously can be advantageous. However due to the reliability problem, observation and group communication are mutually not compatible.

Due to the numerous advantages offered by group communications in the context of building automation, we will rely on this mechanism throughout this thesis. Nevertheless, we address the aforementioned issues in Section 5.2.2 by proposing some adaptations to CoAP and its extensions.

3.5 Discussion and Summary

In this chapter, we have presented the current modern architecture of the Web. Following this trend, the Web of Things pushes Web technologies to the real world by enabling native Web interactions with physical objects and constrained devices. This largely simplifies the development of Web-based distributed applications. In practice, the Web of Things relation to the physical world only concerns the possibility to retrieve and update values. The other layers of the architecture are more social-centric where the human is the principal aspect. In particular the centralised sharing and authorisation mechanisms relying on social profiles are not adapted to smart buildings. Besides, various practical issues related to the context of building automation need to be addressed before achieving homogeneity at all levels in a participatory manner.

The issues that we explicitly tackle in this thesis towards the development of an homogenised architecture are the followings:

Modular framework for smart buildings: Current proposed implementations integrate the Web patterns directly in a rigid manner without taking a developers point of view to facilitate their integration. Moreover, while the projects state themselves as Web of Things applications, they in fact solely address the field level where Web services allow to interact with the physical functionalities of a device. Considering the problem as a whole with all the underlying components can contribute in a broader acceptance of Web-oriented building automation systems. We therefore propose in this thesis to build a Web-based framework that encompasses all the features of smart buildings, called the *Web of Buildings*. We take a modular approach where the separation of concerns leads to reusable building-blocks increasing interoperability, as shown in Chapter 4.

Scalable architecture: While building automation systems were traditionally installed and maintained by experts, the advent of new technologies questions this practice. Moreover, future deployments will be more dynamic as devices can be easily added or removed by common users with no particular training. This changes the way building networks have to deal with such situations as they must support additional devices in an off-the-shelf manner. The underlying components must therefore accept additional devices in terms of network size scalability, but also from a functional point of view. This property will be at the heart of our architectural and technological decisions throughout this thesis.

Energy-efficient interactions: Building automation systems mostly follow an event-driven interaction model to exchange values between participants in a building network. While these communications represent the large majority of communications in buildings, it is unclear which Web notification mechanism has the lowest footprint with regards to energy, as this has never been an issue so far. Furthermore, the notification techniques have not yet been characterised in terms of energy consumption and which should be preferred to others according to eventing conditions. In Section 5.2 we take a pragmatic approach by measuring the energy footprint of Web push methods and propose an optimisation module selecting the most appropriate method.

Seamless integration of legacy BAS: Current approaches pay only little attention to legacy building automation systems and are usually excluded from the Web-based proposition. Their integration is meanwhile crucial as they will have to coexist with novel IP-based solutions. In Section 5.1 we present how this can be achieved through the use of *smart gateways* mapping native field level functionalities to RESTful interfaces.

In the next chapter, we expose our main contribution of the Web of Buildings, which is an entirely Web-based middleware especially tailored for smart buildings.

Chapter 4

The Web of Buildings

Contents

In this chapter, we present our vision of the Web of Buildings. We describe the four layers it is based on: the Field Layer, Automation Layer, Intelligent Adaptation Layer and the Management Layer, as shown in Figure 4.1. We voluntarily decided to expose our main contribution in a single chapter in order to emphasise this vision of a unified architecture around several building blocks.

Our proposition questions the relevance of the current classic building automation layered architecture. The separation of concerns was defined according to the levels of implementation of each layer, which should decouple the architecture [92, 28]. In the classic architecture, each layer targets specific functionalities within its own borders. For example, the Field Level is limited to network topologies and to the access to basic functionalities with the devices. The complexity of the featured functionalities increases as one goes up the layers, meaning that the Field Level is considered simple while the Management Level brings the greatest value at the organisational level. Although starting from a good intention, the classic model is however too rigid and prevents applications to be easily prototyped as it requires a substantial amount of domain-knowledge.

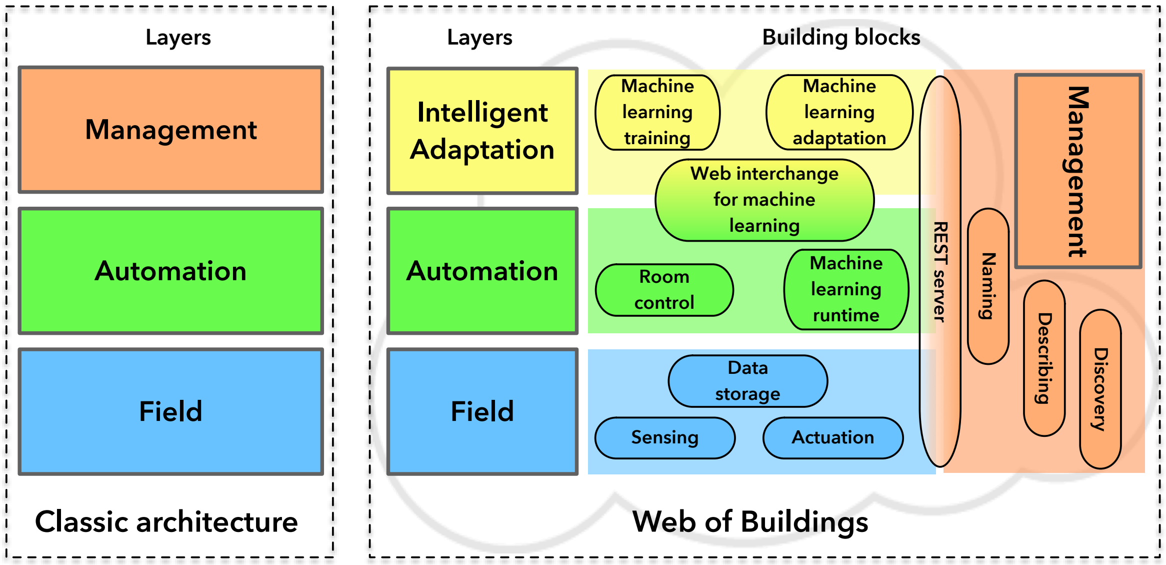


Figure 4.1: Comparison of the classic building automation layered architecture to our Web of Buildings contribution. Each layer can run separately and is composed of a set of building blocks facilitating the development of smart buildings.

By *Intelligent Adaptation*, we mean the automatic elaboration or adaptation of automation rules by using data-driven approaches such as the ones proposed in the machine learning domain. While machine learning is gaining momentum in the optimisation of a building according to user behaviour or other pattern-based parameters, its positioning in the classical architecture has never been discussed yet. Before determining its position, it is necessary to understand its purpose and overall processes as outlined in Section 2.1.3. Models have first to be trained by using historical data. Instead of placing this stage in the automation or management layer (both could be potential candidates), we propose to create a dedicated layer targeting an intelligent adaptation of the building according to its surrounding context. As the machine learning training task is not comparable with any other functionality of the classic architecture, the separation of concerns motivates our decision. Adding a new layer to the architecture avoids increasing the complexity of the existing ones and decouples functionalities, while augmenting visibility and interoperability. Another motivation to add this layer is grounded to the fact that adaptive algorithms are typically CPU and memory consuming during the training time. Their execution will require different solutions as for the other layers. Regarding their runtime stage, it can be seamlessly integrated in the automation layer as it targets the same objective: to exploit the data from the field level in order to take decisions.

In our point of view, the management layer is especially noteworthy and needs to be redesigned, as it does not fit with modern Web architectures. It was originally intended for two purposes:

vertical configuration of the underlying layers, as well as sink for all building related data (storage and analysis). These tasks are rather compatible with a centralised approach where management is performed by a single entity. Moreover, the vertical top-down configuration through the layers makes implementations dependent to each other, which is not compatible with loosely coupled architectures. We therefore reconsider the usefulness of this layer by attributing another purpose. We advocate that the management layer should in a Web architecture become a common ground for general purpose services shared by all the layers. In a Web context, this encompasses RESTful interfaces addressable through a unique naming system. Furthermore, each service or functionality offered by the layers should be made discoverable using the same formalism. The other layers will therefore rely on the services offered by the management level to publish their features and to discover others. Being common to all other layers, it should therefore be positioned vertically so that each layer has access to these functionalities.

Rather than defining the layers according to software functionalities or to their location of execution (for example the management level is dedicated to administrators running it from control workstations), we opted for another approach, defining the layers according to the services and by applying a separation of data. Data should be made available where it is produced and not where it is consumed, like in central sinks specific to applications. This would cause duplication and potentially some inconsistency, in addition to be not cloud-oriented. Storing data where it is produced increases reusability as ad-hoc applications can incorporate them without the need to manage their own storage sinks.

Unlike traditional layered architectures such as the OSI model, the layers that are part of the Web of Buildings are not strictly defined and do not hide the previous ones. The WoB architecture should be considered as an ecosystem of services that eases, step by step, the creation of unified smart buildings applications. These applications can be built on top of the services offered by the implementation of each layer, or by combining several of them depending on the specific requirements of the building control. The layers also reuse services among them to provide higher-level functionalities to application developers, as shown in Figure 4.2. The Intelligent Adaptation Level can for example directly use the storage services offered by the Field Level to retrieve the historical data it needs, and this without going through the automation layer.

As shown in Figure 4.1, each layer is composed of a set of subcomponents considered as building blocks. We have designed these building blocks for an infrastructure that allows to develop complete smart buildings applications solely using Web standards. Web services are spread over devices according to their roles and they all together form a bloc reusable by application developers. From a developer point of view, it lowers the entry barrier and fosters rapid prototyping of opportunistic applications. Moreover, this direct access to global functionalities eliminates the needs for dedicated software as everything is accessible through a well-defined, open and standard API.

The Web lies at the heart of our architecture and is not only used at the Field Level offering common interfaces to devices functionalities like in most projects. We increase reusability and interoperability by relying on loose coupled RESTful interfaces in the entire architecture. Every building block including the underlying supporting components offers uniform interfaces easing the development of control applications. The motivation behind using Web standards as federating protocol to interact with building blocks is also to benefit from the experience and technologies from the last two decades around the development of scalable applications accessible by any kind of device. Furthermore, the openness and simplicity of REST naturally fit with the requirements of heterogeneous building automation systems.

In this chapter, we outline each layer composing the Web of Buildings. We start by describing the general architecture of the components in these layers. Then, we look at the services and

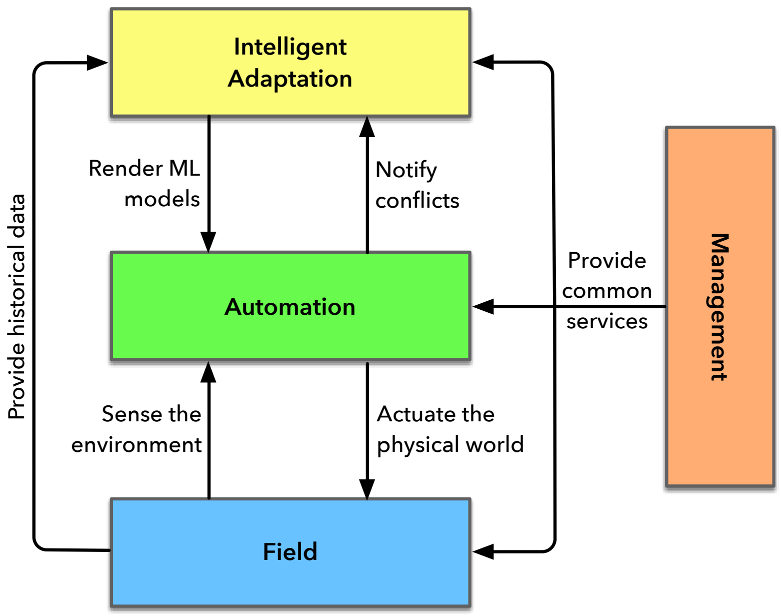
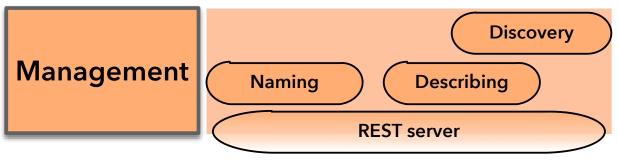


Figure 4.2: The interaction is not necessarily achieved by traversing all layers but must be seen as a set of services offering functionalities that can be easily reused among the other layers. The Management Level is positioned vertically as it offers essential common services such as naming or discovery.

APIs they offer and propose implementations of these building blocks. The concrete building blocks are evaluated in terms of performance while the more generic concepts are applied to specific scenarios and evaluated in Chapter 5.

4.1 Management Level



In the first layer of our Web of Buildings architecture, we address the common functionalities that will be shared and reused by the other layers. This mandatory layer represents the minimal requirement that has to be implemented on devices. It is similar to the "device object" in BACnet that has to be present on each device. However, we go further in the minimal functionalities that are to be provided by including generic services, which will be specialised by other layers.

Our proposal is to offer a generic layer that will help to expose different kind of functionalities and capabilities in the core of the Web. For this, we offer a reusable shared REST server that will allow other layers running on a device to expose their services to the Web by registering them in a simple manner. This approach lessens the complexity of implementation and avoids each layer to implement its own server, which would cause numerous issues (memory and CPU limitation, port configuration, etc.). Moreover, the Management Level also defines a common naming to address machines and resources, which along with the REST server allows a unified and global access to things.

Then, we illustrate how the REST server can be the bearing component for the dynamic discovery of capabilities. The semantic Web comes with powerful concepts and languages giving a strong semantic meaning to descriptions that become understandable by all participants. This paves the way for machine-to-machine direct discovery, without requiring any human in the loop. Furthermore, this scenario can be applied to the context of smart buildings where devices can automatically compose mashups according to their description and other properties.

4.1.1 Hybrid REST Library

Resources exposed through RESTful interfaces lie at the heart of Web-oriented architectures and do so in the Web of Buildings. Any kind of capability or functionality can be represented in terms of Web resources that are manipulated through requests according to the REST architectural style. This interaction principle is the basic common ground to ensure compatibility between devices that potentially come from different manufacturers. Offering a straightforward way to expose RESTful Web services to the building network is a key constraint. From a developer's perspective, writing RESTful APIs can represent a daunting task as the underlying HTTP and/or CoAP protocols have first to be implemented in order to provide a REST engine.

Some libraries allow to rapidly create RESTful applications not only by already encompassing the application protocol stack but also by providing simple-to-use programmatic ways to define service endpoints. We can for example mention the Jersey [307] library (HTTP) for Java, the Windows Communication Foundation [308] (HTTP) for .NET or Californium [309] (CoAP) for Java. Developers register their services within the REST engine by indicating their properties such as path, method and the callback function that has to be executed upon receiving a request. Although easing developer's work and hiding the protocol specificities, these libraries are rather tailored for server-side applications than for constrained devices having a limited amount of memory.

However, these libraries show a set of limitations. First, they only support a single application protocol (HTTP or CoAP) while both can coexist in RESTful environments. This strongly limits the interoperability between devices and requires a large amount of gateways, increasing complexity and latency. Next, as only CoAP defines a notification mechanism, the HTTP-based REST libraries are not able to inform clients about events and changes of states, and impose therefore to rely on a polling strategy.

In this section we propose the elaboration of a hybrid REST server library overcoming the aforementioned limitations. This library is the underlying ground and will be shared and reused by the other layers and building blocks to expose their services, as shown in Figure 4.1. This motivates its vertical positioning rather than being horizontal on top of the architecture's stack. The hybrid library will be used throughout this thesis as base to expose any kind of service on constrained devices.

Requirements

Although being both RESTful protocols, HTTP and CoAP differ from several points, complicating their merging into one hybrid library. First, they do not use the same transport protocol and they therefore require being able to handle TCP and UDP sockets separately, which is not obvious on constrained devices. They indeed limit the number of sockets due to memory restrictions. The possibility to run in parallel TCP and UDP depends on the hardware specification, as well as on the number of concurrent sockets that can be opened simultaneously.

While CoAP defines a notification mechanism based on asynchronous responses [158] as outlined in Section 3.4.1, several techniques can be used with HTTP. As explained in Section 3.3.6, we will only consider HTTP callbacks and Websockets during this thesis. Each mechanism meanwhile works differently and requires the server being able to handle each of them independently. This disparity largely complicates the management of notifications and would ask that the developers have an extensive knowledge of the mechanisms.

Regarding the security aspect, both protocols are also relatively different. HTTP comes with a set of well-defined techniques to secure resources. The Basic Access Authentication [164] that the client (often a Web browser) provides credentials (user and password) in a request to be granted with access. It is generally coupled with an encryption protocol such as SSL/TLS to ensure that critical values such as the passwords are not transmitted in clear text. Due to the large number of things that can be found in a smart building, it becomes rapidly unmanageable to create and share credentials for each one and to distribute these among things that have to communicate with each other. The Basic Access Authentication of HTTP is therefore not suited for the IoT and smart buildings. CoAP for its part has only little be enhanced with security yet. As TLS is connection-oriented, it cannot be applied as such to CoAP relying on UDP. It has therefore been adapted for datagram communications which led to DTLS [165]. Several versions especially optimised for the Internet of Things were proposed with regards to timeout and certificate exchanges among others [166, 167, 168, 169]. These works are currently assembled in a working draft [170] that will especially address the concerns of DTLS in the IoT. The use of DTLS on things with CoAP significantly increases the round trip time for requests as well as the energy consumption. Measurements showed that the initial negotiation phase can reach up to 800ms on constrained devices [171, 172]. The *Authentication and Authorization for Constrained Environments (ACE)* IETF working group is currently very active in defining use cases for IoT that encompass building automation [173, 174]. Several exploratory works using either OAuth [175, 176, 177] or EAP [178] as authentication procedure have been published as drafts. These technical solutions are at preliminary stages and none of them has yet been adopted as RFC nor are predestined to become a standard. Because of this lack of clarity in authentication mechanisms and the impacting delays induced by a security layer, we will not further focus on security constraints in this thesis. Authentication can however be achieved either with access lists [179, 180] or authentication servers [181, 48, 182, 183].

In general, we aim to satisfy the following requirements for a global reusable hybrid REST server that will expose all kinds of services in the WoB:

Ease of use: The library must be easy to use from a developer's point of view by offering a limited set of comprehensible functions. Developers should be able to add new services rapidly in order to lessen the time required for prototyping.

Eventing: Notifications are a key concept of building automation and need to be transposed to the Web. In order to simplify developers work, they must be able to generate events that will transparently result in notifications using a format that is understood by the client according to the protocol used during registration.

Genericity: Although HTTP and CoAP differ in several aspects, the API of the library should hide these differences and optimise code reusability for user functions. In other words, developers should be able to write only one service regardless from the application protocol.

Lightweight: The memory footprint of the library must be kept as low as possible to leave most of the space for user applications.

Integrability: The integration on constrained devices should be as transparent and seamless as possible. Developers must be able to decide which protocols and functionalities they want to run on the server. They can adapt and select which components of the server are enabled or those which are not required. A very constrained device for example will only be using CoAP as application protocol and dismiss HTTP requests.

Another important aspect of the hybrid REST server is the ability to handle CoAP multicast requests according to the group communication specification [159]. Application developers should be able to define resources that are shared among a group identified by its IP address and port. HTTP is for its part not multicast-compatible as it uses TCP as transport protocol which is connection-oriented.

Mapping CoAP and HTTP

CoAP and HTTP are both RESTful application protocols following a request-response model and rely on the same concepts of URI and methods encoded in the header. They however differ in the range of options they provide, which induces some incompatibilities. This is especially problematic when envisioning a higher level abstraction that encompasses both protocols. Such a higher level abstraction model is necessary to hide the protocols specificities and to provide an easy handling of the library. We outline the main differences between both protocols and provide some solutions to overcome these limitations.

A first disparity concerns the response codes that are used to indicate to the client the result of the request (successful, request not valid or an error). HTTP defines a substantial set of status codes that are identified by a code number and a string in the header. For its part, CoAP defines even finer grained response codes to inform the client more precisely about the result of the request. For example, it explicitly notices if a resource has been successfully deleted or updated, which is not the case with HTTP as only a 200 OK will be returned in both cases. In our library, we opted to use the CoAP response codes as the ones that will be offered to users when replying to requests. This choice is relatively obvious as it allows to simply match them to HTTP status codes while the opposite would lead in uncertainty. For example the HTTP 200 OK can potentially be mapped to several CoAP response codes (2.02 Deleted, 2.04 Changed and 2.05 Content). By proceeding from CoAP to HTTP, we can ensure the semantic correctness of the response as the REST requirements are followed. If the user is responding to a HTTP request, the library will automatically map the user selected response code to the equivalent HTTP status code. In Table 4.1 we outline the mappings that are implemented in the library. Only two CoAP response codes can potentially be mapped to several HTTP status codes. The rule to decide which status code is appropriate is very simple as it depends on whether a payload is returned or not, and can be implemented in a straightforward manner.

Another difference between CoAP and HTTP regards the available methods. CoAP indeed does not include the entire set of methods that is proposed by HTTP. Only the methods allowing to manipulate resources can be used, which are `GET`, `PUT`, `POST` and `DELETE`. The `HEAD` and `OPTION` methods are missing in CoAP. While the former is only seldom used and not very important for RESTful interfaces, the second one can be relatively limiting in some scenarios. The `OPTION` method allows indeed to retrieve some properties about the resource, which can be used to automate service consumption. Its importance in the context of semantic discovery is outlined in Section 4.1.3. As its presence is required for such purposes, a way to simulate the `OPTION` verb with CoAP is necessary. Three fundamentally different approaches may be considered:

Method: Methods are identified in the CoAP header by an integer that according to the specification identifies the method. The `OPTION` method could therefore be implemented by considering it as a special number. This can however cause some troubles and can result in incompatibilities if the specification changes. It could be possible that a new version either

CoAP Response Code	HTTP Status Code	Remark
2.01 Created	201 Created	
2.02 Deleted	200 OK 204 No Content	A payload is returned No payload is returned
2.03 Valid	304 Not Modified	
2.04 Changed	200 OK 204 No Content	A payload is returned No payload is returned
2.05 Content	200 OK	
4.00 Bad Request	400 Bad Request	
4.01 Unauthorized	400 Bad Request	
4.02 Bad Option	400 Bad Request	
4.03 Forbidden	403 Forbidden	
4.04 Not Found	404 Not Found	
4.05 Not Allowed	400 Bad Request	
4.06 Not Acceptable	406 Not Acceptable	
4.12 Precondition Failed	412 Precondition Failed	
4.13 Request Entity Too Large	413 Request Representation Too Large	
4.15 Unsupported Content-Format	415 Unsupported Media Type	
5.00 Internal Server Error	500 Internal Server Error	
5.01 Not Implemented	501 Not Implemented	
5.02 Bad Gateway	502 Bad Gateway	
5.03 Service Unavailable	503 Service Unavailable	
5.04 Gateway Timeout	504 Gateway Timeout	
5.05 Proxying Not Supported	502 Bad Gateway	

Table 4.1: Mapping between CoAP response codes and HTTP status codes.

reuses the number we have attributed for another verb, or that the working group decides to add the `OPTION` method with a different number than ours.

Option: CoAP defines options in the same way as methods by using integers referring to the option meaning in the specification. Some of them can be followed by values if required. A specific option number could be used to indicate that the request actually targets to retrieve the resource description. Obviously this request would be of type `GET` as the client wants to read a representation. This solution has the same drawbacks as the previous approach.

Placeholder: Gathering a resource description can be considered as reading a sub-resource. As resources are hierarchically organised, a special placeholder located right after the resource path can indicate that the client wants to retrieve its description. For example, the `coap://host/temperature/.{description-format}` URI could point to the description of the temperature resource. The special `/.{description-format}` placeholder at the end of a path can be interpreted as the `OPTION` method by servers. The drawback is that systems have to agree to use the same placeholder and that the specific path will be dedicated as `OPTION` method.

With regards to the REST architectural style, the first solution appears to be the most appropriate as it follows strictly the guidelines. However, we do not believe that it should be implemented currently as the CoAP protocol is relatively young and it may change to include the `OPTION` method in a near future. This will result in incompatibilities between the specification and our solution if the chosen integers are not identical. We therefore opted for the placeholder approach as it is the most independent to the potential evolutions of CoAP. The coupling is not tighter than the other solutions and offers even more flexibility as the transition to a newer specification including an `OPTION` method will require minimal changes. We will therefore use the `/.{description-format}` placeholder in the WoB architecture to access resource descriptions with CoAP. More information on how to retrieve resource descriptions is provided in Section 4.1.3.

Managing Notifications

Notification principles are well specified for both CoAP with the observer pattern and for WebSocket that upgrades the connection to its lightweight socket-based protocol. HTTP callbacks lack of standardisation as only the concept is well-known but no specification describes the technical points and how it should be implemented. This lack of rigour is reflected in the implementations of REST server libraries that do not offer any notification mechanism. The developer is responsible to implement its own notification scheme by defining resources for registering and unregistering clients, and this for each available resource on its RESTful API.

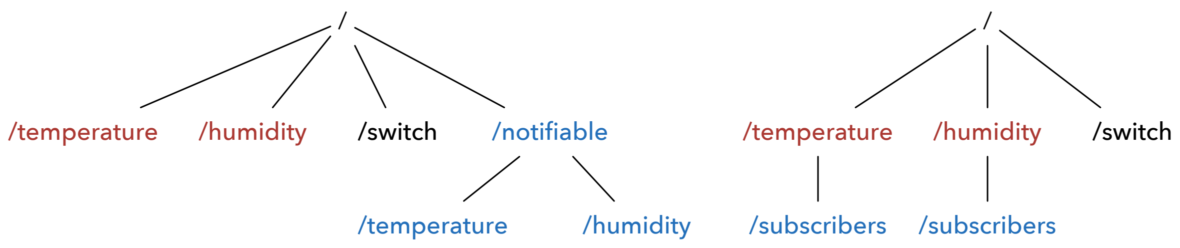


Figure 4.3: Two potential ways are available for a client to register for notifications. On the left part, a common resource (`/notifiable`) regroups all the resources offering notifications. Client will interact with these during registration. On the right part, each resource that supports notifications exposes a sub-resource (`/subscribers`) dedicated to registration. This latter way follows better the ROA style and is preferred.

In order to provide a standard way for things to exchange notifications with HTTP, some properties have to be fixed. Clients need to address a particular resource in order to register themselves at the producer. This can be realised in two ways, as shown in Figure 4.3. First, a common resource can be defined to manage the registration of resources supporting notifications. In the second approach, each resource offering notifications will expose a sub-resource dedicated to registration. Both approaches could be easily implemented for registration. The first one however does not follow rigorously the ROA style as dependent resources are not hierarchically organised in a parent-children manner. The registration resource can be considered as a sub-functionality of the sensor resources and needs therefore to be exposed as a child. Clients will `POST` to the `/subscribers` sub-resource by indicating their callback URI in the payload as text. The producer will add the client to the list of subscribers for the resource. Finally, it responds with the `Location` option in the header containing the URI of the newly created resource. Each time the resource changes, the producer will issue `PUT` requests to all clients with the new representation contained in the payload. When a client wants to unregister, it simply sends a `DELETE` request to the URI that was specified in the `Location` option during the registration phase.

Our REST server library includes the mechanism shown in Figure 4.4 at its core and handles all the steps of registration, notifications and unregistration transparently to users. They only have to mention during configuration that a resource can generate events. Depending on the type of clients, the library will push the new value to clients by using the appropriate protocol that was used during registration (CoAP observe, WebSocket or HTTP callback).

Architecture

In this section, we present the software architecture of our hybrid REST server library that will be used throughout this thesis as ground for exposing services.

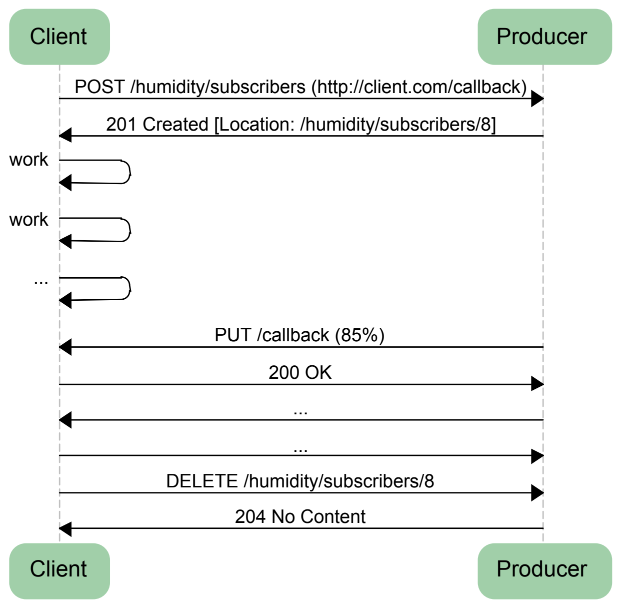

Client
Producer
POST /humidity/subscribers (http://client.com/callback)
201 Created [Location: /humidity/subscribers/8]
work
work
...
PUT /callback (85%)
200 OK
...
...
DELETE /humidity/subscribers/8
204 No Content
Client
Producer

Figure 4.4: During registration, a client issues a `POST` request to the `/subscribers` sub-resource by including its callback in the payload. The producer responds with the `Location` option indicating the URI available for unregistration.

Our architecture is based on a modular approach to achieve a loose coupling between the components. Using such a modular style not only improves code comprehension but also has a positive impact on the reusability and maintenance. We opted for the C language as it is the most widespread one for constrained devices often relying on a 16-bit micro-controller architecture. The general architecture is composed of modules which are themselves regrouped according to their functionalities, as shown in Figure 4.5. We present in more details the internal composition of modules and their interactions.

Application Protocols The application protocols are implemented independently and each implementation handles its own set of sockets (UDP for CoAP and TCP for HTTP). Their roles are to parse the incoming binary frames according to the protocol specifications and to delimit the payload from the header. The next step consists to parse the header and to fill a generic structure that will be passed to the REST server. This structure holds not only the payload data, but also fields for the method, path, content-type, message ID (CoAP), token (CoAP), observe option (CoAP) and a handle to the socket as shown in Listing 4.1. The addition of a new protocol will not affect the currently implemented ones as only the structure will be extended with new properties. The REST server will for its part be adapted to consider these properties. Last, the application protocols take care of building the responses coming from the REST server and will reply to clients. We briefly describe the most important characteristics of each protocol implementation:

```c
typedef struct Frame_t {
  Method_t method;
  char** path;
  char* payload;
  ContentType_t content;
  int socket_handle;
  union{
    int message_id;
    char* token;
```

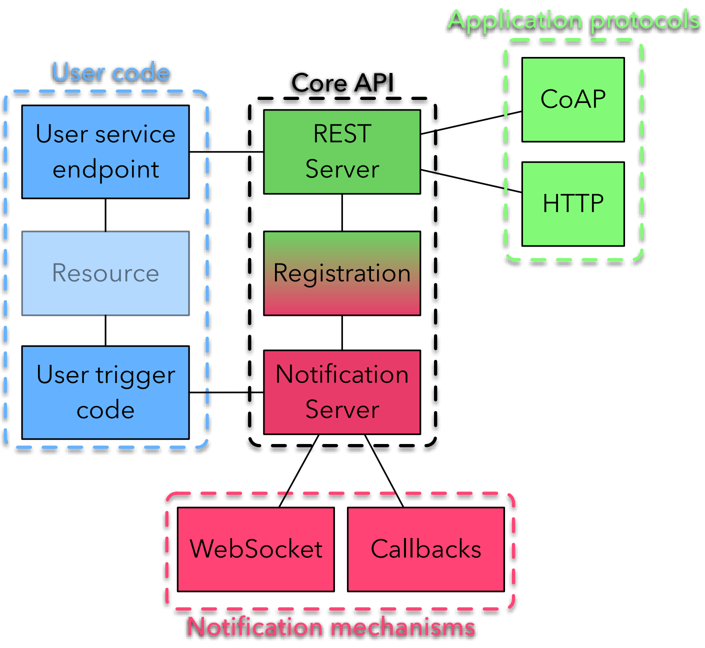


Figure 4.5: The hybrid REST server library is composed of several modules performing dedicated tasks in the overall architecture. This loosely coupled decomposition allows straightforward maintenance and also eases the addition of new modules.

```
    byte observe;
  } CoAP;
  union{
    byte upgrade;
    char* key;
  } HTTP;
} Frame_t;
```

Listing 4.1: Generic structure used for forwarding requests to the REST server. It can be seamlessly extended with new protocols.

CoAP: We base our CoAP implementation on the microcoap [310] library that is compliant with draft version 18 (last draft before RFC). It has the most lightweight memory footprint comparing to other libraries. The observe and group communication extensions that we intend to use in our REST server are unfortunately not integrated in this library. We therefore decided to modify microcoap in order to include the observer pattern and the support for multicast group communications.

HTTP: Many HTTP server implementations like Wt [311], onion [312] and libmicrohttpd [313] are freely available on the Web. As they are entirely implementing the HTTP specification and for some also including SSL security, their use on constrained devices is relatively difficult to achieve. Indeed their memory footprint would be too heavy and most of them use pipes to achieve multi-threading, which are quite rarely available on constrained devices. Instead of modifying one of those, we opted to build our own HTTP implementation from scratch by restraining the HTTP functionalities to those needed in RESTful environments. This allows to simplify the implementation and to limit the memory footprint. The consequence of this choice is obviously that our implementation will not fully be compatible with the HTTP specification.

Core API The Core API is the key engine of our REST server library. Its role is to provide simple functions to developers who want to expose Web services. We here tried to reproduce the behaviour and service declaration of the Java Jersey REST library [307]. Due to the limitations of the C language in comparison to Java, we were not able to make users simply declare their services by following a predefined function prototype along with meta-data using Java annotation. Users have to explicitly register their services on the server. This does not represent a limitation and only slightly complicates the development process. We give further insights on the role of each module:

REST Server: The REST server receives requests from the application protocols and will determine their nature. Those that are related to registration or unregistration of notifications will be forwarded to the Registration module. For all other requests, the REST server will determine if services that were registered are matching, basing on the method and path of the resources. If it has found a positive match, it will call the user-defined callback function. Otherwise, a `404 Not Found` response will be sent to the client. When the callback function is returning, it will forward the response structure to the appropriate application protocol. It will build the response frame and will send it to the client.

Registration: The list of clients observing resources is managed by the Registration module. According to the requests provided by the REST server, it deletes entries or adds new ones by keeping track of the client's properties and the used protocol. It serves as interface between the REST server and the Notification server.

Notification Server: When a new value for a resource is available, the Notification server is informed by the user code in order to dispatch it to the clients. It first retrieves the list of clients and according to the notification mechanism associated with the client, decides which method to use. It either forwards the value to the WebSocket or Callback implementations, or to the REST server in the case of CoAP.

Notification Mechanisms The notification mechanisms include WebSocket and HTTP Callbacks. The CoAP observer pattern being an inherent part of CoAP is thereby directly implemented with the protocol itself. Each notification mechanism handles independently its clients and sockets according either to the specification for WebSockets [152] or to our defined HTTP callback scheme. The WebSocket implementation relies on the libwebsockets [314] library being especially lightweight. Once the TCP connection has been closed by the client, the Registration module will remove it from its list. Regarding the HTTP callbacks, we developed our own implementation of an HTTP client that will send the requests to the client's callback URI.

User Code The user-related part of the architecture is first of all composed of the implementation of a resource that will be accessed through a service and will potentially generate events. This resource can either be a physical value representing the state of a sensor or an actuator, as well as any other kind of virtual value such as a configuration or an operation. The internal representation of the resource, which is mainly a variable that can be of different types according to the nature of the resource (integer, float or string) will be used by the modules as follows:

User service endpoint: The functionality of a service is implemented by the client within a certain function that was previously registered within the REST server. Using a request-response model, long processing within the service implementation should be avoided as much as possible in order to increase responsiveness and to avoid timeouts. The function should perform few computations and directly return the state of the resource or temporarily store the provided payload so that it can be further processed outside of the service implementation, as for example in the main loop.

User trigger code: The user has continuously to watch if an internal resource representation has changed. This is mainly realised either through interrupts for physical values or by comparing the actual value to the previous one in the main loop for other types. Every time a value changes, the user code dispatches this event to the Notification manager that will propagate this to the clients having previously registered.

Usage

The REST server allows to expose a REST API either accessible through HTTP or CoAP. We here attempt to reproduce the Java Jersey API in a lightweight manner for constrained devices. The developer's tasks are kept as simple as possible by registering services and handling requests. Setting up the REST server only requires a few steps, which starts by including the necessary library .h and .c files in the user's project. The minimal code to configure and to run the server is very simple and consists of only four steps, as shown in Listing 4.2. We here depict those steps:

1. Declare a RestServer type variable (line 1).
2. Initialise the server by providing the ports for the HTTP and CoAP interfaces (line 5). A value of `NULL` for a port will disable the application protocol. The developer can in this way select which protocols will be enabled.
3. Define services for resources (lines 7-10). The exposed API is defined during this step. The user will provide the method (`POST`, `PUT`, `GET` or `DELETE`), the path to the resource and to which application protocol it should respond. The path can be dynamic by defining attributes enclosed inside brackets. For example, `/temp/history/{id}` would allow to define a generic path for a history storage service. Any request starting with `/temp/history/` will be dispatched to the corresponding service independently on whatever follows.
4. The services are registered on the REST server (line 12) by providing a pointer to a callback function (line 20) that will have to handle the requests.
5. As most socket libraries do not generate events when new data is available, we have to continuously check for incoming connections. This is achieved by calling the RestServer_do function inside the main loop of the program (lines 15-17).

```
1  RestServer restServer;
2  
3  void main(){
4    ...
5    RestServer_init(&restServer, 80, 5683);
6  
7    struct RestService svc_temp;
8    svc_temp.method = GET;
9    svc_temp.protocol = HTTP|CoAP;
10   svc_temp.path = "/temperature";
11 
12   RestServer_register_service(&restServer, &svc_temp, temp);
13   ...
14 
15   while(1){
16     RestServer_do(&restServer);
17   }
18 }
19 
```

```
void temp (Dictionary* uri_params, Dictionary* params, char* data,
    Response* resp){...}
```

Listing 4.2: Code example showing how to set up the hybrid REST server and registering a service.

In order to be able to handle incoming requests and to assign them to registered resources, users have to provide callbacks. The callback must respect a function prototype as follows: `void callback_name (Dictionary* uri_params, Dictionary* params,char* data, Response* resp)`. The parameters contain various useful information to handle the requests:

Uri_params: Dictionary of dynamic attributes inside the location-path of the URI that are defined in brackets during registration.

Params: Dictionary of parameters contained inside the URI, as for example `?key1=value1&key2=value2`.

Data: The raw payload data contained in a string.

Resp: Structure used to indicate response information. The user specifies the response code, and if necessary the content type and payload.

The response is formatted according to the HTTP or CoAP specifications and automatically sent to the client after the code in the callback function has been executed. The user has only to provide the response code and the potential payload.

Initialising the Notification server is very simple too. First, the .h and .c files have to be included in the project. The required steps in order to run the Notification server which are shown in Listing 4.3 are the followings:

1. Declare a NotificationServer type variable (line 1).
2. Initialise the server (line 6).
3. Register the resources that will generate events (line 11). The same structure references as for the REST server have to be used (line 8).

Once those steps are done, the Notification server is ready to receive client registrations and to generate events. Internally, the user has to check for changes on the physical or virtual representations (temperature, switch, configuration, etc.). When a change of state arises, the user will have to dispatch this notification. To do so, the user has to call the `NotificationServer_dispatch_event` function, by indicating which resource is concerned, as well as the payload data and its content-type.

```
1  NotificationServer notificationServer;
2
3  void main(){
4    ...
5    //REST server initialisation and resource definition
6    RestServer_init(&restServer, 80, 5683);
7
8    struct RestService svc_temp;
9    ...
10   NotificationServer_init(&notificationServer);
11   NotificationServer_enable_notifications(&notificationServer, &
       svc_temp);
12   ...
13
14   while(1){
15     ...
```

```
    if(temp_value_changed)
      NotificationServer_dispatch_event(&notificationServer, &
          svc_temp, temp_value, TEXT_PLAIN);
  }
}
```

Listing 4.3: Code example showing how to set up the notification REST server and enabling services for accepting client registration. The final step is to dispatch an event when a change of state (physical or virtual value) arises.

Developers can control which application protocol should be supported and whether resources offer notifications or not. This allows to fit the library according to the available computational power and memory of devices. Besides, the API of the REST and Notification servers are simple to use and limited to a small set of functions. The general API is shown in Figure 4.6 with the most important functions highlighted in bold.

Evaluation

In this section, we assess the performance of our hybrid REST server. As being the only one of its kind at the time of writing this thesis, it is unfortunately not possible to compare the results with similar libraries. Indeed, no other library for smart things combines CoAP and HTTP like in our architecture. We however provide some measurements of the performance of our solution allowing to compare the differences between CoAP and HTTP.

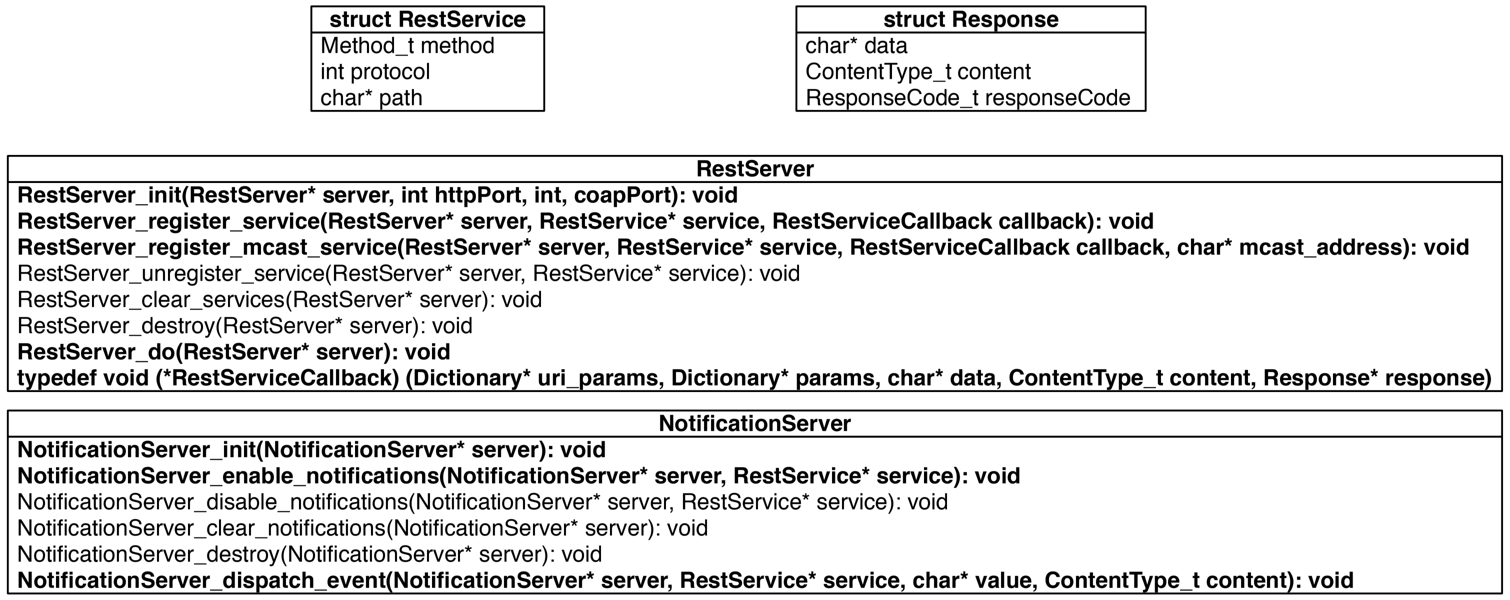
Our experimental setup was composed of a single OpenPicus Flyport Wi-Fi node running our Hybrid REST server. The REST server was configured with a single service accepting both CoAP and HTTP requests. In order to evaluate the performance of the server only, the service was not performing any computational task and was directly returning a single byte of data.

As previously mentioned, we opted for our own implementation of HTTP to reduce the memory footprint. To be able to position the performance of this implementation compared to existing ones, we decided to perform the same evaluation using the embedded Web server of the OpenPicus. The device provides indeed a simple Web server returning Web pages. It can however not be used as reference implementation within our library because no API is offered to developers for catching and handling requests. It is therefore not feasible to expose Web services as solely Web pages are supported.

CoAP vs. HTTP Albeit both protocols are RESTful and dedicated to interact with Web resources, they largely differ in their specifications. Differences are not only observed in the complexity of implementation [184], but also in the achievable performances [185]. The latency of responses is essential in real-time applications such as smart buildings where even small perceivable delays will be considered as a bad user experience.

The round-trip time of 500 consecutive requests on a dummy Web service is shown in Figure 4.7. As expected, CoAP outperforms both implementations of HTTP by far. Avoiding the TCP connection negotiation contributes in lowering the round-trip time as a single packet carries the request. The average round-trip time for CoAP is 11 milliseconds. From the cumulative density function (CDF) shown in Figure 4.8, one can see that all requests are answered in less than 23 milliseconds.

For both HTTP implementations, the results are not as good as for CoAP. This difference is due to the TCP protocol used by HTTP requiring connection negotiation (opening, sliding window and closing) that is time consuming. In addition, HTTP carries all the header data in ASCII, slowing



```
struct RestService
Method_t method
int protocol
char* path

struct Response
char* data
ContentType_t content
ResponseCode_t responseCode

RestServer
RestServer_init(RestServer* server, int httpPort, int, coapPort): void
RestServer_register_service(RestServer* server, RestService* service, RestServiceCallback callback): void
RestServer_register_mcast_service(RestServer* server, RestService* service, RestServiceCallback callback, char* mcast_address): void
RestServer_unregister_service(RestServer* server, RestService* service): void
RestServer_clear_services(RestServer* server): void
RestServer_destroy(RestServer* server): void
RestServer_do(RestServer* server): void
typedef void (*RestServiceCallback) (Dictionary* uri_params, Dictionary* params, char* data, ContentType_t content, Response* response)

NotificationServer
NotificationServer_init(NotificationServer* server): void
NotificationServer_enable_notifications(NotificationServer* server, RestService* service): void
NotificationServer_disable_notifications(NotificationServer* server, RestService* service): void
NotificationServer_clear_notifications(NotificationServer* server): void
NotificationServer_destroy(NotificationServer* server): void
NotificationServer_dispatch_event(NotificationServer* server, RestService* service, char* value, ContentType_t content): void
```

Figure 4.6: Application programming interface (API) of the REST and Notification servers along with the related structures. The functions in bold stand for important ones that must be used to achieve a minimal working.

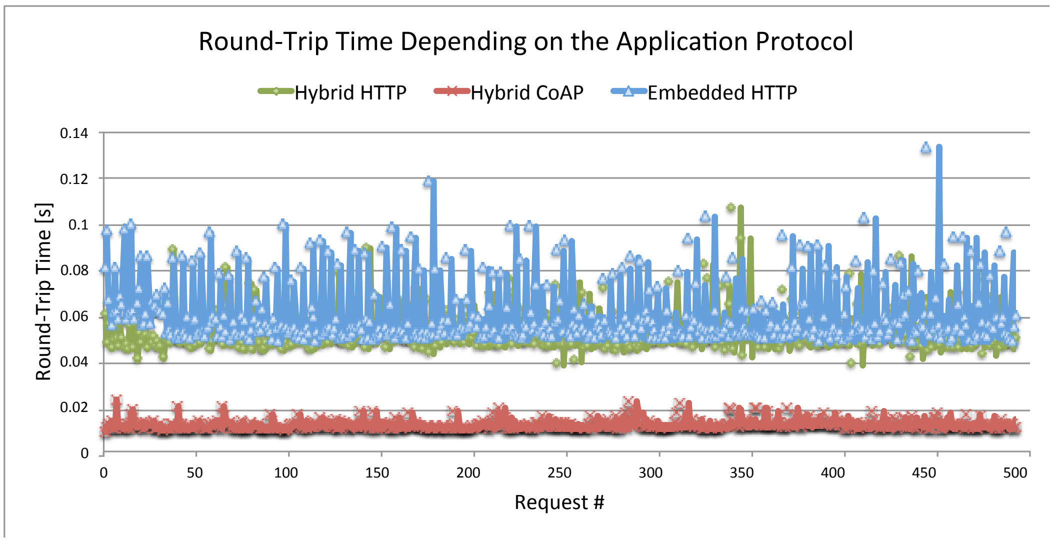


Figure 4.7: Round-trip time for 500 consecutive `GET` requests.

the processing. The embedded OpenPicus Web server is the worst performing implementation with an average round-trip time of 60 milliseconds. Our HTTP implementation performs better with an average of 52 milliseconds for 500 consecutive `GET` requests. When comparing the CDF of both implementations, it appears that they are very similar besides being shifted.

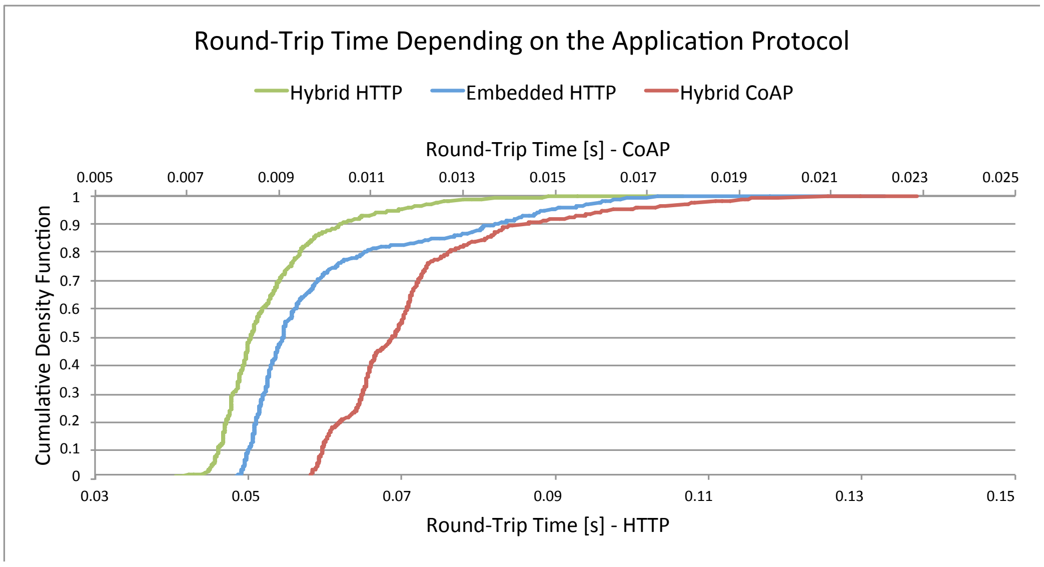


Figure 4.8: Round-trip time cumulative density function for the different protocol implementations.

The difference between our HTTP implementation and the embedded OpenPicus one (about 8 milliseconds in average) is difficult to explain as there is no documentation about what functionalities of HTTP are implemented. We can however through this comparison validate our implementation.

CoAP is intended for constrained nodes having limited memory and computational power, which is reflected in our evaluation. It indeed performs about five times better than HTTP. This optimisation opens new dimensions to use RESTful APIs in time-critical application whereas HTTP was historically the bottleneck. In this thesis, we will therefore use CoAP as application protocol whenever possible instead of HTTP.

Evaluating Concurrency In this evaluation, we observe the behaviour of the implementations when facing concurrent clients sending simultaneously requests. Similarly to the previous evaluation, we tested how CoAP and HTTP face this situation. In Figure 4.9 we show the round-trip time when several concurrent clients send `GET` requests to the same resource.

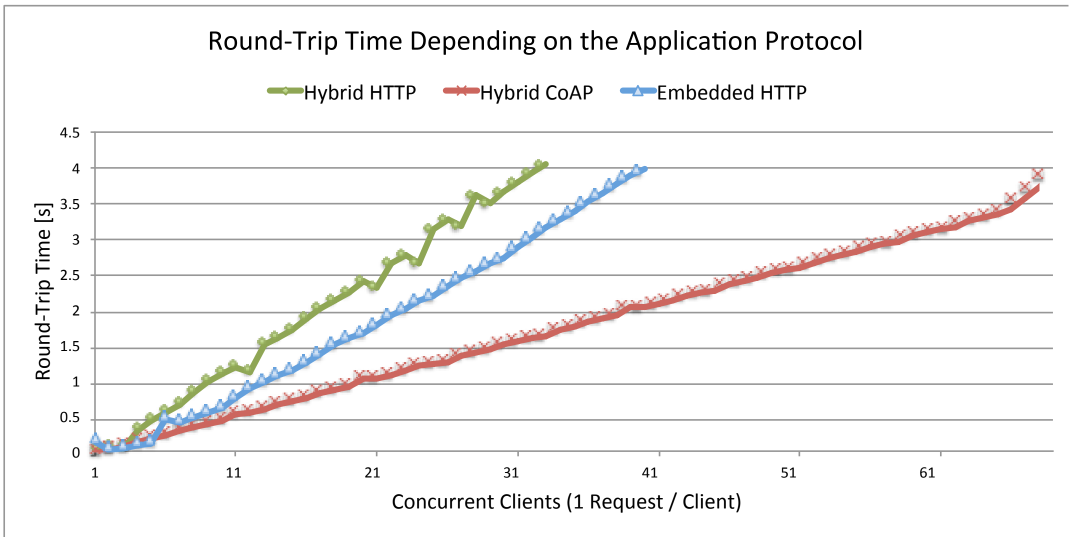


Figure 4.9: Round-trip time for concurrent clients sending `GET` requests.

Not astonishingly, CoAP is more scalable in terms of concurrent requests than both implementations of HTTP. The round-trip time for all implementations grows linearly with the number of clients. However, the slope of both HTTP implementations is larger than CoAP. In our evaluation, we considered a maximum timeout of four seconds. The HTTP implementations reach rapidly this value so that our implementation only supports up to 33 concurrent clients. The embedded OpenPicus HTTP server seems to perform slightly better than our implementation with up to 40 clients. CoAP is the most scalable with up to 69 concurrent clients.

Although these results seem at first glance limiting, they are more than sufficient in smart buildings applications where sensors and actuators will rarely be accessed by concurrent clients. These limitations are however not due to bad implementations but are a consequence of the OpenPicus API and its firmware offering no threading mechanism. When comparing our results with other CoAP implementations such as Tiny CoAP [185] or Erbium [186], the performances of our hybrid REST library are very similar and even better than those obtained with consecutive requests. Our hybrid REST library performs worst in the case of concurrent clients as other implementations rely on tiny operating systems offering threading. It is very likely that we should have obtained comparable results if the OpenPicus would have offered a threading mechanism.

4.1.2 A Building-Oriented Naming System

Relying on Web technologies and especially Web services does not only consists of exposing RESTful APIs to the world, but requires an underlying architecture that can bear the higher-level paradigms. Naming is a compelling example of such architectures providing services to the application layer. From an historical background, the Web leverages on the legacy Domain Name System (DNS) [187] to map IP addresses to and from human readable names. Besides easing navigation on the Web, it allows application developers to address machines by their rather stable names instead of IP addresses that can change over time. While it has been sufficient for classical Web architectures composed of Web pages and business services, it suffers from several limitations which tend to question its suitability in modern Web architectures that include mobile objects.

Currently used DNS architectures are rather heavy installations requiring expertise and efforts during deployment and management. Moreover, the current specification does not provide any support for the mobility and the dynamism of everyday objects [188, 189]. This is an important drawback in the context of smart building applications where pervasive devices can appear and disappear, as well as move inside the building. From a broader point of view, the DNS is a rather old and barely dynamic technology that has not evolved to provide needful services for the IoT nor for smart buildings.

Although some distributed naming systems like mDNS [190] are addressing these shortcomings, they do not fulfil all requirements related to building automation systems. We believe that the future naming system for the IoT and especially if used in buildings should offer solutions to the following issues: *scalability, autonomy and fault tolerance, resource orientation, pervasiveness, backward compatibility and efficiency.*

In this section, we present the design and architecture of a novel naming system at the convergence of the aforementioned requirements. We put a special emphasis on the distributed aspect that we especially tailor to the morphology of buildings, while reusing the already available REST server to provide a common application protocol. We presented this work in [6].

Requirements

The legacy DNS and its Internet of Things evolution, namely the multicast DNS (mDNS) are the two main architectures used to provide a naming service [191]. While the first one is hierarchically structured, its centralised and rather static properties are not compatible with IoT requirements. Multicast DNS targets sensor network environments and distributes the naming records among several nodes in the network over multicast messages. The mDNS naming follows a flat hierarchy and eliminates the notion of zones along with delegations.

It is precisely this hierarchical organisation along with the definition of independent zones which is a key advantage of the legacy DNS. Zones are related to a particular virtual organisational structure regrouping a set of domains. mDNS makes unfortunately abstraction of this concept as records are not organised and can be stored anywhere. Though this is suitable for small-sized networks, it is however not applicable to larger networks. A network interruption could cause a client to be unable to contact the name server storing the record, whereas direct communication with the target resource server is still possible. This observation leads to the requirement that name servers should be hierarchically organised and located as close as possible to the entries they store. Looking at the pitfalls of the existing solutions, we propose the following requirements for a building-oriented naming system:

Scalability: Office and factory buildings can be endowed with thousands of devices. If we push our vision further looking at future smart cities, million of devices will sense and interact with the city. The underlying naming system must thereby be highly scalable and support a very large number of devices. In addition, the number of devices should not impact the performance of the system.

Autonomy and fault tolerance: One major drawback of the DNS is that it has to be installed and maintained by professional people with expert knowledge. This shortcoming is one of the reasons why it cannot be considered for IoT applications. A building automation system will only be accepted by users if their contribution for installation is reduced to the minimum, targeting a plug-and-play approach. This means that users should not have to set up a DNS server and zone delegations as part of the installation. The naming system should configure itself and being autonomous regarding the variations in size of the network. Additionally, the naming system should be insensitive to single points of failure, meaning that the loss of servers should be automatically compensated.

Resource orientation: We can indeed identify a clear trend to push RESTful APIs down to field devices in smart homes and smart buildings [36]. According to the Web of Things, each device exposes its capabilities as REST services in form of Web resources, thus standardising the application layer. Existing classical building automation networks as KNX and EnOcean working with specific stacks have already been made compatible with Web technologies by mapping their devices to RESTful APIs [10, 13]. By following this trend, there is only a small step from considering a Web-based naming system, exposing name entries as Web resources.

Pervasiveness: With the advances made in pervasive computing, sensing devices are becoming smaller and more affordable. The limited available computational power restrains the tasks that can be run, as well as the protocols that can be understood by those devices. The naming system has then to be lightweight and compatible with constrained devices having only little memory and CPU.

Backward compatibility: We target our naming system to be compatible with the legacy DNS. As it is on the heart of the Internet and the current Web, dissociating our architecture from it would limit the interaction with IoT devices as data and automation algorithms are traditionally hosted on dedicated computers or in the cloud. Our proposed smart building naming system should therefore allow a transparent compatibility of requests issued from the legacy DNS or from the smart building's naming system.

Efficiency: Finally, energy efficiency is also a concern in the context of smart buildings. The naming system should therefore limit the number of messages for synchronisation and for data replication.

Peer-to-peer (P2P) networks have gained momentum in distributed system architectures as their main advantage is to provide a hierarchically structured overlay on top of an existing model [192, 193]. This concept has since be reused for naming systems where existing infrastructures need to be hierarchically organised according to virtual properties [194, 195]

We propose to leverage the concept of P2P networks as underlying distributed architecture. In our naming system we introduce an architecture and its implementation that enables smart things to form by themselves a logical structure, needing no human in the loop. It uses the notion of P2P networks overlays to provide a hierarchical organisation of naming zones.

Furthermore, we illustrate how the building structure can be used as reference to form independent naming zones and how multicast allows to reduce the traffic in order to minimise the footprint.

A Building Hierarchy

Peer-to-peer networks require a base topology that will serve to create an overlay on top of it. Sharing platforms such as Bittorrent, Gnutella or eDonkey use the network infrastructure (essentially IP addresses) as underlying topology. Due to the dynamic of smart things that can potentially move inside the network even without changing their IP address [196, 197], it is not feasible to base our hierarchy on the network infrastructure. Additionally, this structure will serve as naming scheme to uniquely identify smart things on the network, which requires logical name entities. The physical location of peers is an interesting alternative and has already been successfully used in P2P networks as well as for naming scheme [198, 199, 200]. We therefore propose to rely on the physical location of sensors, actuators and other types of devices to address them. The location of static devices can be manually configured by users. Moving devices can benefit from the advances made in indoor location to determine their position, for example through the placement of reference nodes [201].

Buildings are naturally organised and comparable to Russian dolls. A building is composed of several logical locations that include other ones. For example, a building contains floors which themselves contain some rooms (building->floors->rooms). Devices will be mostly attached to rooms, but can also be installed in floors or referring to the whole building, for example for external sensors like a weather station. This building morphology can be reused and translated to form a tree, as shown in Figure 4.10. Furthermore, this fits naturally with naming concepts as the legacy DNS itself is also following a tree model.

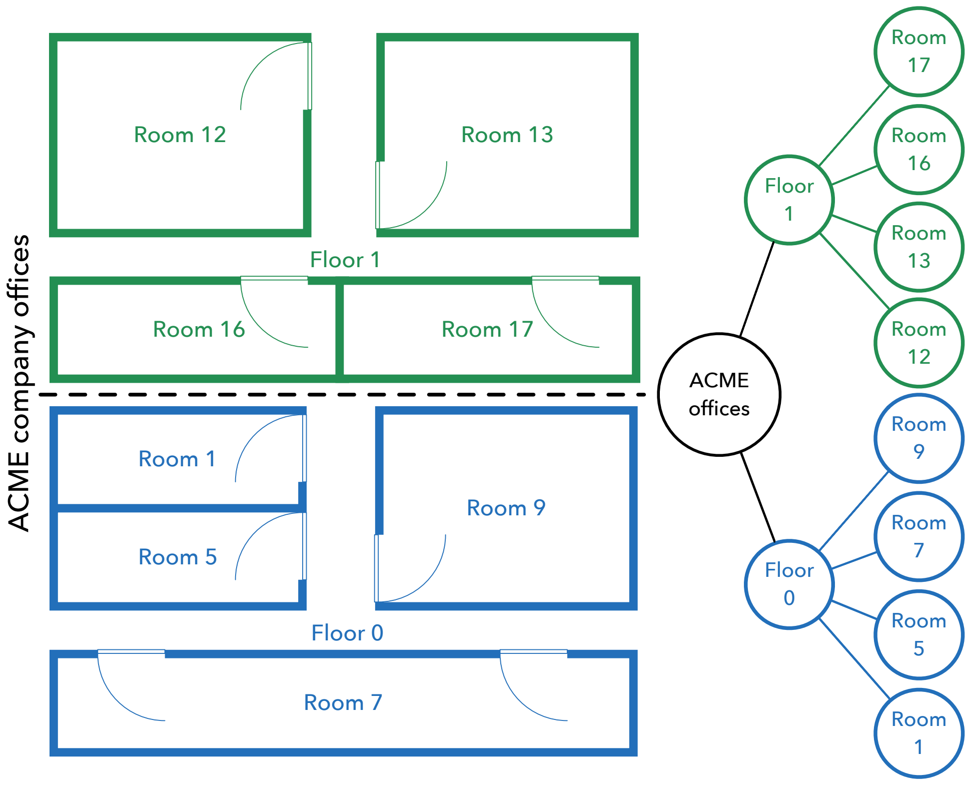


Figure 4.10: The building morphology composed of floors and rooms can be translated to a hierarchical tree structure. The creation of naming zones can subsequently rely on this organisational model.

The largest entity that encompasses all the others is the root node. The root will be most of the time a building but is rather flexible and can represent a broader location such as a district or even a town in the context of smart cities. At the other end of the tree, leaves bear the smallest entities where smart things can be attached to. They are not limited to rooms but can also go at a finer grained level such as windows, doors, racks, enclosures, walls or even furnitures like tables and shelves.

URIs impose a hard-defined scheme as follows `<protocol>://<authority>/<path>` where the `<authority>` part points to the machine that hosts the service. Logical names are usually preferred as humans are more able to remember names instead of addresses, and as IP addresses are volatile and can change over time. This authority field can be further decomposed as it identifies a specific host within a particular domain as follows `<host>.<domain>`.

Our approach of a building hierarchy naming can be transposed to URIs for consuming services. Each room is considered as a domain that can hold hosts. They are identified by their own name representing their functionality, as for example temperature, heating or lighting. So, the general way to determine the absolute name of a machine is simply achieved by navigating the tree and appending the name of the device. Figure 4.11 illustrates this principle where the `Device1` located in the second room domain (pretending it to be a temperature sensor named `temp`) could result in following authority `temp.room348.02.factory.acme.com`. The general composition of the authority part in combination with the building hierarchy structure is as follows `<device_name>.<location>.<organisation>`. The tree model may follow the already defined organisational naming of rooms within the company, and may facilitate the composition of absolute names for end-users.

As for any naming scheme, it is forbidden to have several devices with a similar name in a particular location. The probability of facing this situation should be small and can be troubleshot by making the tree more granular.

A Zone-Based Overlay

Having previously defined our reference model, we can now group those domains/locations to form an overlay. This requires some logical and repeatable conditions that will affect its structure and performance. For a hierarchical naming system, the most straightforward way is also to rely on domains, and in our context on room locations. The legacy DNS regroups contiguous portions of the domain space in zones for which the management responsibility has been delegated to a single entity [187]. In our approach, zones will have a higher semantic meaning as they will not refer to a management entity but will be regrouping several room locations. They therefore depict a broader location space named *area*.

Figure 4.11 shows how areas are built on top of the locations' tree to form a graph of broader spaces. An area starts at a particular location and extends itself downward in the tree to sub-locations in a recursive manner. This expansion stops either when reaching leaf locations or at the top-level of sub-locations where a new area starts. Each area is responsible to manage the records that are located in its contiguous domain space composed of one or several locations. Links between areas are called *delegations* and distinguish a trust relation to another entity.

Besides being the result of a logical decomposition of the building, following a location-based hierarchical model allows to combine the benefits of *Content Delivery Networking* (CDN) which is gaining momentum in sensor networks [202, 203]. The aim of CDN is to store the information where it is needed in order to reduce network traffic and to speed up the delivery of content [204]. Content servers are placed at the edges of the network backbone. These servers are working as a cache for the original content and will be regularly updated with new content. Applying as such this paradigm to sensor network and naming remains however questionable as it will require a significant amount of cache servers for each area, having to be maintained and updated. However, we will show that we can reuse some concepts of CDN in the context of smart buildings.

Smart buildings distinguish from other sensor network types as the interactions between participants happen in a very localised way. Actuators are using values provided by sensors to adapt the environment according to rules. When thinking of the morphology of buildings composed of floors and rooms, this interaction will mostly remain in the entity the things reside in. For example, a push button will drive the lights in the same room and not those in the next room nor one floor above. This observation applies generally to an entire building except some specific cases where sensor values are required from the outside.

This particularity of smart buildings allows to decouple areas from the rest of the building by

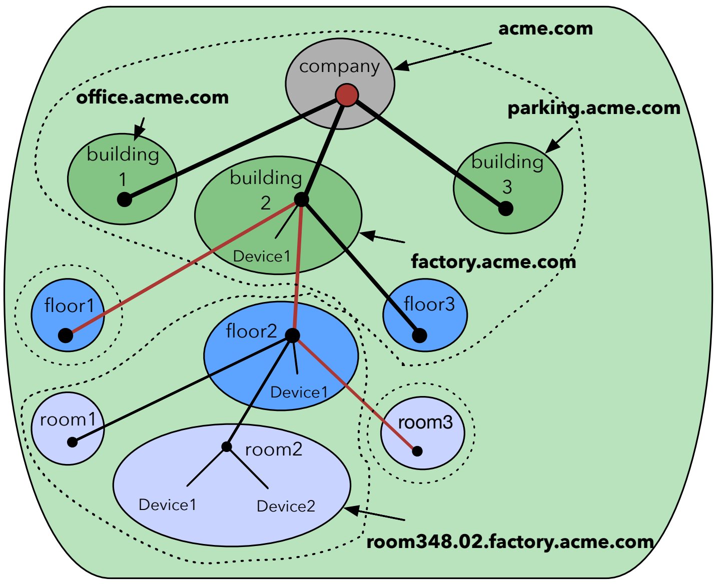


Figure 4.11: Areas regrouping contiguous locations are created on top of the tree model. Each area manages the name entries for its locations and reports to its parent area. Red lines depict delegations between areas.

relying on the proximity concept of CDN [205]. Name server should be placed where the most lookups will be performed, thereby in their own area. Proceeding this way has two advantages:

Congestion reduction and energy savings: The lookups and responses will not have to traverse a large quantity of routers or to be forwarded many times in mesh networks. As the content (name records) are stored close to the clients, the network traffic will not have to potentially pass through the entire building. This obviously reduces the total number of packets and reduces the risk of congestion. An important side-effect is the overall energy consumption reduction of network equipments [206]. Nodes participating in a mesh network are the most beneficent of these energy savings as there will be fewer packets to forward.

Network failures: While the interaction is localised in rooms, so do usually the network infrastructure. Wiring follows the morphology of the building as devices in a room will be attached to a switch, which is itself linked to the floor's router. Border routers connecting mesh networks to the network backbone are also generally placed in every room or floor. As name servers will be placed in the area they manage, this ensures that the failure of network equipment in other areas will have no or few consequences. The lookups within an area will continue to work regardless of the network status in the rest of the building.

Area Composition Strategies

The number of areas influences the availability and scalability of the network. Having more areas increases the need for higher bandwidth as delegations induce synchronisation traffic. Moreover, as each area will be composed of multiple name servers (master and slaves), the synchronisation exchanges will also have a negative influence. However, this approach allows to decouple rooms. Fewer areas will allow to lessen the synchronisation induced by delegations between parent and children areas. Another consequence is to require less servers to manage the name space. This choice is left to the developer according to the requirements and is actually a trade-off between network independence and traffic, respectively server minimisation.

As we target an energy-efficient name system, our proposed algorithm tries to reduce the number of areas in order to minimise the network traffic. The partitioning of the tree and the related number of areas depend on several variables which are: the maximum number of name entries for an area and the number of desired slave servers. While a brute force approach would result in having the best decomposition of the tree, its CPU and memory requirements would be too high for constrained devices. We therefore opted for a less CPU consuming algorithm that will potentially not find the best global solution. We here explain the most important steps of our algorithm that are also depicted in Figure 4.12:

1. In the first step, we ensure that all leaves of the tree have at least two servers. The algorithm will recursively merge leaves with their parent until each area holds at least two servers.

2. Once each leaf has the required number of servers, the algorithm will merge non-leaf areas with their children so that each one holds at least two servers and does not exceed the maximum number of entries. We minimise the number of areas by trying every possibility to merge non-leaves with their children. At the end we obtain a new tree where each area will be managed separately.

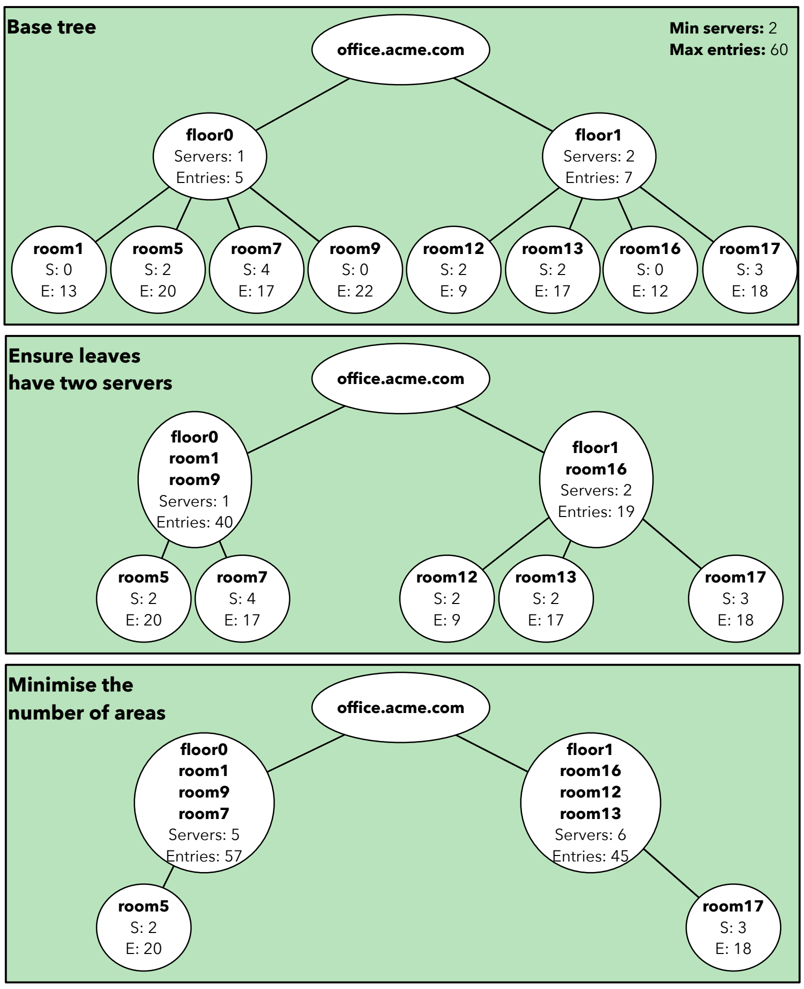


Figure 4.12: Example of the partitioning of a name tree with a minimum of 2 servers and a maximum of 60 entries per area. Each location corresponds to a single area in the first step, represented by the circles. The algorithm then ensures that all leaves meet the constraints. Finally, areas will be merged together whether possible to form combined circles.

The partitioning of areas is triggered in two distinct cases:

- An area exceeds the maximum number of entries it can handle. The master of the area will partition its own area and delegate the management of new ones to other servers. This operation results in at least one new area.
- An area can be merged with one of its children when the sum of entries of both areas is lower than the maximum it can handle. In this scenario the parent will repatriate the child area and merge it with its own. This operation results in one area less.

Server Roles

Our architecture follows an agent role repartition to improve robustness and responsiveness in case of failures. The loss of an agent is not detrimental while they are sufficient to take over. This means that agents will be synchronised to ensure smooth failovers. In our vision of the WoB, sensors, actuators and everyday objects having enough computational capacities can run server agents, avoiding the need for dedicated name servers. Agents are pre-installed with a software piece that controls their behaviour. When starting up, they will have no particular role and will simply wait for conditions to have one of the followings:

Master: The agent is responsible for an area. It stores the name entries and will answer the lookup requests. Additionally, it ensures to have the appropriate number of slaves servers by promoting new ones if required. The master is the only agent empowered with the decision ability to merge a child area or to split its own one.

Slave: The failover is assured by slaves that are continuously synchronised with the master. This ensures that there is no delta between the master and a slave, so that no entries will be lost. A slave has a passive behaviour and reacts solely when it notices the master is down.

Idle: This is the basic role of an agent that has registered in an area. It is in an idle state and waits to be eventually promoted to a slave server by a master. An agent will remain in idle as long as there are enough other slaves in the area.

Name servers are not manually configured like it is the case for the legacy DNS, but are elected among server agents. This election process contributes to the autonomy of the naming system as there is no need for a human in the loop. Each area is managed autonomously and has to ensure that it has at least one master and one slave name server, which guarantee replication. Server agents can either promote themselves to name servers or be designated by others depending on certain conditions, as detailed below. A server agent always starts up with no specific role. Its first step is to inquire for a name server managing the location it resides in. If it receives no response, it will then promote itself as master for an area that only comports its domain.

As it is the first master of the building network, it will include in its own authority all new locations. Using full qualified names, it can gradually internally build the location tree. For example, a promoted master with the location `00.c.organisation.com` will know that the hosts `05.00.c.organisation.com` and `d.organisation.com` are in the same location tree, as parts of their full qualified location are identical. New masters will be designated as soon as the conditions regarding the maximal amount of records will exceed and results in a partitioning of the area.

There are also other ways for a server agent to become a name server:

- The server agent is promoted to a slave of the area by the master. This ensures that the area has always at least one slave if possible.

- A slave server promotes itself as master as soon as it detects that the master is no more responding, and will in a second step designate a new slave. This ensures that there is always a server that has the possibility to repartition the area with at least one slave for replication.
- After the partitioning of an area, new masters will be designated for the resulting new areas. Each new master has then the responsibility to designate slaves.

The criteria of selection to promote idle server agents to slave name servers can influence the overall performance of the naming system, especially for mesh networks. Two main criteria are relevant when selecting new servers: the closeness to the master for slaves and the number of available servers in a location. For the first one, the locations of the master and the slave will influence the costs for replication and the ability to overcome connection breaks. While close-located master and slaves (numbers of vertices in the name tree) allows to minimise replication costs, it induces some potential difficulties if the rest of the area gets disconnected from the servers. Meanwhile we recommend to choose a slave server that is the closest to the master as we want to reduce the communication costs. Another argument advocating to choose servers based on their closeness is that if one part of the area is disconnected from the servers, other server agents will temporarily promote themselves as name server until the connection is recovered. Regarding the selection of new masters after partitioning, we put more importance on selecting a name server located in a domain having the highest number of server agents. We motivate this choice in order to comply with the aforementioned criteria, as there will be more probability to select a slave residing in the same domain (same location) and thus optimising the replication costs.

Multicast Data Replication

Naming data replication on at least one slave is mandatory to avoid the single point of failure in case the master fails or gets disconnected. Different strategies of data replication are applied in DNS and mDNS. In the legacy DNS, the zones are periodically exchanged, creating a delta if the master fails. This delta can be reduced by replicating zones on a very regular basis, although increasing communication costs. In mDNS, data replication is ensured through the multicast messages that are processed by each host and added to their knowledge. This has the advantage to avoid full replication of zones between hosts. However, some information could be lost if a host misses a multicast message, and would result in a loss of knowledge.

In our architecture we aim at combining the strengths of both approaches. Master name servers will periodically fully replicate areas on their slaves, depending on a configurable value which can be time-based or according to a number of updates. In order to lessen as much as possible full replications and the related communication costs, we propose to associate a different multicast group to each area. This multicast group is used by hosts to update records (i.e. announcement, update and deletion) and to make lookups, instead of using unicast communication with the master. Following such an approach allows to bring two key foundations of our architecture regarding data replication and autonomy. First, one multicast message will have as effect that both the master and the slave will update their records simultaneously, thereby reducing the need for full replications. Secondly, using a multicast group for lookups and record updates is decoupling the area's management from its physical name servers. Indeed hosts have no clue with which physical server they exchange as they communicate over multicast. In the case of a master name server failing and being replaced by a slave, this will have no influence on the hosts as they do not communicate directly with servers.

A RESTful Naming Protocol

In this section, we describe the naming protocol relying on the aforementioned concepts of a distributed area-based naming. Unlike the legacy DNS and mDNS, we opted to define our own protocol instead of reusing the frames and data models specified by DNS [207]. We motivate this decision by the objective to homogenise the application layer and to leverage on RESTful APIs. This choice brings one important advantage in a world of constrained devices: having to implement only a single application protocol instead of several. The side-effects are an easier development process as well as the ability to save memory.

Method	Resource	Purpose
Management interface: predefined multicast address		
GET	/area/{domain}	Discover which zone manages this domain (full qualified name)
Area interface: area multicast address		
GET	/host/{host}	Lookup the IP address of this host (full qualified name)
PUT	/host/{host}	Add or update the IP address of this host (full qualified name)
DELETE	/host/{host}	Delete the entry of this host (full qualified name)
PUT	/server/{id}	Register a server agent
DELETE	/server/{id}	Unregister a server agent
PUT	/area/{id}/size	Child area notifies about its number of entries
PUT	/area/{id}/group	Parent area informs about its new multicast zone group
DELETE	/area/{id}	Parent area wants to merge this area with its own one
Server interface: unicast server address		
PUT	/area/{id}/{role}	Designate the server master or slave
DELETE	/area/{id}/{role}	Inform a slave that it is no more a name server
PUT	/area/{id}	Perform a full replication of the name entries

Figure 4.13: RESTful API protocol for the distributed naming system. Three distinct interfaces are available according to the scope and purpose of the services.

As show in Figure 4.13, the API is decomposed in three distinct interfaces according to their scope and purpose:

Management interface works with a predefined multicast group shared by all participants in a building. This service is exposed by master servers in each area. It allows for agents or clients to discover which area manages a particular domain. The master server having the domain under its authority will answer with the multicast group of its area.

Area interface is reachable over a random generated multicast group different for each area. This technique allows to impersonate the interaction and makes therefore the system very scalable and fault tolerant. A first series of services are dedicated to basic client operations to perform lookups and for registration, update and deletion of hosts (name entries). This interface serves also for communications between areas where masters will be able to repatriate a child area in their own authority.

Server interface is the only interface using unicast addresses and is dedicated to the interactions between the agents. A master will promote agents to the role of slaves and will replicate entries by addressing specific agents.

One particularity of our approach is to consider fully qualified names as Web resources. For several purposes as lookups and entry management, clients have to provide an absolute name. Working with ROA, these can easily be represented as resources that can be read, modified or deleted. For example, the absolute name `temp.room348.02.factory.acme.com` can be translated to the following resource path `/com/acme/factory/02/room348/temp`. Moreover, REST resources are also following a tree model which fits with our naming structure following the building morphology. The resource representation is only reversed from the classical notation.

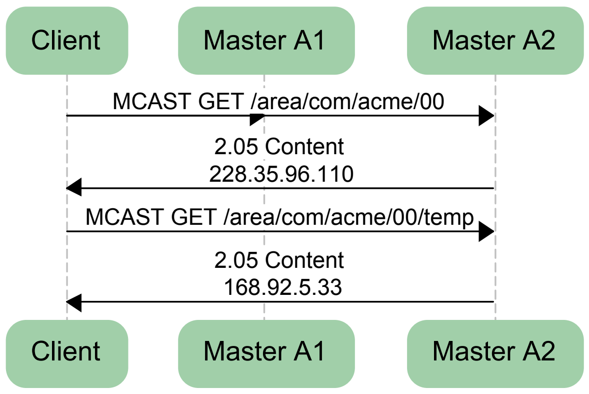


Figure 4.14: A client first uses the management interface to discover which area is managing the location. In a second step, it performs the lookup by using the provided area group address.

The lookup process as well as the management of name entries by hosts is achieved in a 2-phase procedure, as shown in Figure 4.14. During the first step, a client will issue a multicast request over the general management interface to discover in which area resides the domain. All master servers will receive the request and check if the provided domain belongs to their authority. Only the master having a match will respond with the group address of the area. This step no longer needs to be repeated for the same domain as the client can cache the result. Indeed, the area group address is unlikely to change over time.

During the second step, the client will address the area over its multicast group and will ask for the IP address of the fully qualified name. The master of the area will respond with the current IP address of this entry or with a `Not Found` response if the host does not exist.

Ensuring Legacy DNS Compatibility

Ensuring a transparent compatibility with the legacy DNS is a constraint that cannot be overstepped. However, our proposed autonomous naming system relying on Web technologies is naturally not compatible with the DNS. This can be bypassed by introducing a gateway that will interconnect both worlds and that performs seamless protocol translations.

From an Internet user point of view, the gateway acts as a standard DNS server being authoritative for the domains that are managed by the distributed naming system. This implies that a DNS delegation has been configured on a higher positioned server. The gateway transparently forwards requests to our naming system using the Management and Area interfaces to first discover areas and to then perform lookups. The responses are sent back over the standard DNS protocol.

On the other side the gateway will act as master server for domains that are not managed by our naming system. The aforementioned 2-phase procedure remains valid for this purpose. WoB clients intending to communicate with a legacy DNS machine can recover the IP address by

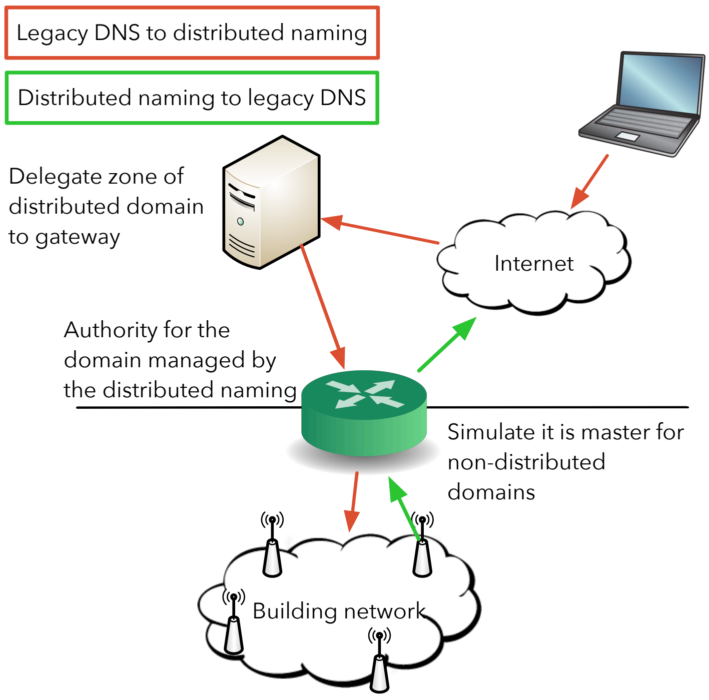

Legacy DNS to distributed naming
Distributed naming to legacy DNS
Delegate zone of distributed domain to gateway
Internet
Authority for the domain managed by the distributed naming
Simulate it is master for non-distributed domains
Building network

Figure 4.15: The building network using the distributed naming can be made compatible with the legacy DNS by introducing a gateway. Its role is to translate the requests and responses between the DNS protocol and the Web resources.

first issuing a request over the Management Interface. The gateway will respond if the searched domain lies outside of the building network. The gateway will in a second step translate the Web lookup into DNS record read messages forwarded to legacy DNS servers. The response will be unmarshalled and sent back to the client.

Both communication ways are illustrated in Figure 4.15. The gateway should be placed at the border between the building network and the Internet. In order to be able to forward the requests and unmarshal the responses, it has necessarily to implement our distributed naming protocol as well as the legacy DNS protocol. The gateway allows to link the different implementations together resulting in a single name tree that is composed of WoB devices and legacy DNS machines.

Evaluation

In order to evaluate the benefits of our Web-based naming system, we implemented a prototype version in Java that aims at simulating agents. We motivate the choice to perform simulations rather than real-life experimentations as scaling up to hundreds of rooms and areas was not feasible in the time frame of this thesis. Moreover, a real-life implementation would have made performance metrics difficult to observe on each single device.

Our implementation follows the guidelines and concepts explained in the previous sections. Our evaluation focuses on validating the architectural concepts around distributed areas rather than lookup response time. More precisely, we want to observe the consequences in terms of data transfers generated by the architecture. Efficiency being an important requirement, we opted for CoAP as application protocol to leverage on its short header. Regarding the data representation, JSON was used for the full replications between masters and slaves.

We chose as simulation playground three buildings of the College of Engineering in Fribourg (HEIA-FR), Switzerland. The buildings are composed of office, meeting and class rooms spread over five floors per building, giving a total of 180 rooms. Our simulation randomly makes hosts (sensors) and server agents appear and disappear according to predefined ranges as follows:

Office: 2 server agents maximum, 20 hosts maximum

Meeting: 1 server agents maximum, 10 hosts maximum

Class: 2 server agents maximum, 12 hosts maximum

Corridor: 2 server agents maximum, 8 hosts maximum

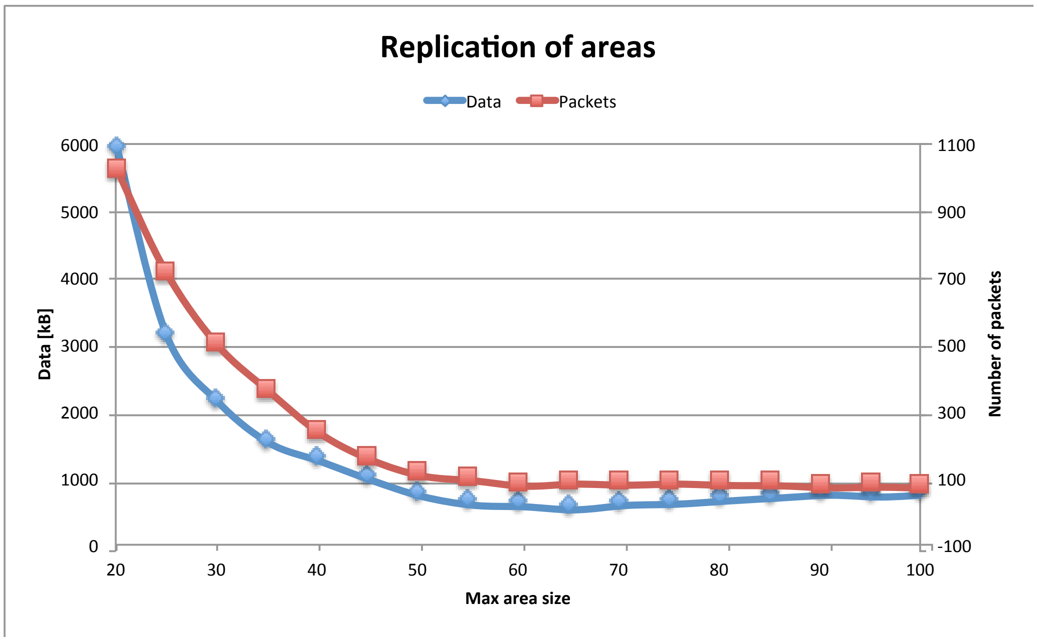


Figure 4.16: Total number of packets and data for 5000 random operations (add host, remove host, add server agent and remove server agent) on a simulated building structure composed of 180 rooms.

Each repeatable simulation generates a total of 5000 operations which are of the following type: add host, remove host, add server agent and remove server agent. The location tree as well as the areas are built dynamically by the agents during the simulation. For each simulation, we varied the maximum size of areas and observed the number of packets along with the data size (UDP + CoAP + payload) resulting from the data replications. From the Figure 4.16 we can see that the number of packets and the amount of data are closely related. However, defining small areas increases their number and thus implies a substantial amount of synchronisation data between them. We can observe that both metrics tend to become steady as soon as the area size can cover at least two rooms, which is approximately 45 hosts in our simulation. The explanation for this observation is relative to the balance of the number of exchanges between and within areas, which are both at their lowest. More generally, we observe that with our solution 5.6kB in average are needed for one replication, while the total number of exchanged data is 608kB in the best case.

We also performed the same simulations using legacy DNS servers as proposed in [198] and using mDNS [190]. Both approaches gave us a higher value of total exchanged data for replications, namely 1220kB (+100%) for DNS and 1040kB (+71%) for mDNS. This is explained by the DNS protocol itself having a relative consequent overhead compared to the combination of CoAP and JSON. Moreover, masters and slaves agents simultaneously update their records using shared multicast hosts announcements, updates and deletions. Such procedure avoids performing full replications at a high frequency. Compared to the legacy DNS and mDNS, our naming system outperforms in terms of number of packets and exchanged data, and can therefore be considered as more efficient.

4.1.3 Service Descriptions

The Web of Things paradigm allows us to expose device capabilities as Web services that can be consumed in a very straightforward manner. While this simplicity is a fundamental advantage, it also raises other issues. One of these is related to the semantic of services: *Given a heterogeneous ecosystem of smart things, how do we understand what is the purposes of the services and how they can be used?*

The Web encountered the same problematic as it grew from a research-oriented network to a collection of billions of mixed documents. Web pages rely on HTML to formalise the content and can be parsed to gather a certain understanding about the topic. As this process is not straightforward, several keywords related to the topic can be placed inside the document. These are used by search engines to index a Web page in order to make it searchable.

While searchability is addressed in Section 4.1.4, the WoB requires a similar formalism to describe resources. In this direction, the `WS-*` stack provides a set of tools [43] [315]. These are neither suited for ROA nor in the context of sensor networks [208, 209]. Moreover, the context of smart buildings differs from the Web and `WS-*`, as both, humans and machines should be able to understand what a service can achieve.

In this section, we report on a semantic-based description model that once shared among smart things allows each other to understand their capabilities. This model distinguishes from others by providing a solution self-explaining how to consume a service. Then, we illustrate how the REST architectural style can be used to browse descriptions.

A Semantic Web Description Model

In order to make smart things reusable, they need a mechanism to describe what they are and the services they expose. As application developers and eventually machines are going to build mashups, a way to describe a smart thing following Web principles is needed. The descriptions formalism has also to be seamlessly understood by both actors. Being able to describe resources is not a new research area specific to smart buildings, but is rather a complex field that is currently tackled by many Semantic Web initiatives [210]. Moreover, it is also related to the context of ubiquitous computing [211, 208, 212].

We propose here a model intended to be shared among all devices in a smart building, allowing them to describe themselves by leveraging on Web formalisms. This model, shown in Figure 4.17, is based on the observation that a common model describing a domain knowledge can contribute to achieve homogeneity at the application level [213, 214, 215]. To build our model, we consider the device and service properties from a building automation point of view rather than trying to propose a general solution applicable to any sensor network. We describe a smart thing according to two clusters of information:

Device-related properties refer to information that is relevant to the device itself and do not depend on the exposed resources.

Type: Contains a description of what the device is in terms of platform and software (e.g. manufacturer, model and version).

Location: Specifies where the physical device is geographically located.

Resource-related properties describe the purpose of services, their signification and how they should be consumed.

Capability: Allows to understand what the service provides in terms of functionality and data.

Gate: Gives insights on how to consume the service (protocol, parameters and method).

Relations: Provide links to the parent and children resources allowing to browse and crawl APIs.

We do not pretend that this model is exhaustive. However we can state that it is sufficient to cover the properties of smart buildings, allowing humans and machines to understand what things and resources are good for. The possible lack of description should not be considered as the impossibility to formalise specificities, but is instead an opportunity to extend this model. This opens a way for evolvability as other requirements targeting either other sensor network types or smart building characteristics can be included to complete the model.

Although the proposed model could be represented by using several formalisms such as WSDL [315], SensorML [216] and oBIX [316], we intentionally do not consider these solutions. The main argumentation for this choice lies in their poor Web compatibility and to the fact that they are not intended for search purposes.

We also observe that several languages have emerged as candidates to describe Web resources. The Resource Description Framework (RDF) [217] [304] and microformats [317] are the most widespread ones, as their specification includes a set of guidelines and are well provided with available semantic description formats.

Microformats are intended to be included in the HTML representation of a resource by marshalling the content with tags. These tags refer to classes giving a semantic meaning to a particular content. Usually more than one microformat is used to describe a Web page. Several standardised microformats are already defined and can be simply reused in descriptions. While microformats provide an interesting way to enhance Web pages with a human and machine readable description, it does not fit to the resource-orientation of smart buildings. Resources representing sensors, actuators and other related services do not use HTML as representation language.

RDF is especially interesting in the Web of Buildings. First, it is agnostic to the format of the resource so that it can describe any type of resource. Secondly, many domain ontologies centralising particular knowledge are available, which eases the conception phase. The descriptions themselves can also be formalised according to different languages and formats, and allows therefore to opt for the one having the lowest footprint. Finally, RDF comes with a powerful query language that allows retrieving descriptions according to some parameters.

From these observations, we propose to rely on RDF to describe sensors, actuators and any kind of service in the Web of Buildings. Rather than building a domain ontology from scratch that encompasses the aforementioned model, we can reuse and integrate already available and standardised ontologies. This approach allows to augment the probability of being compatible with other building-related ontologies. Moreover, it is as step towards domain-ontology interoperability as bridges can be built to translate knowledge between ontologies [218, 219]. Composing on top of standardised ontologies therefore avoids heterogeneity at the knowledge level.

To illustrate this, we show how this approach is applied in the conception of our domain ontology. We reuse some existing well-defined ontologies in the sub-groups of our description model.

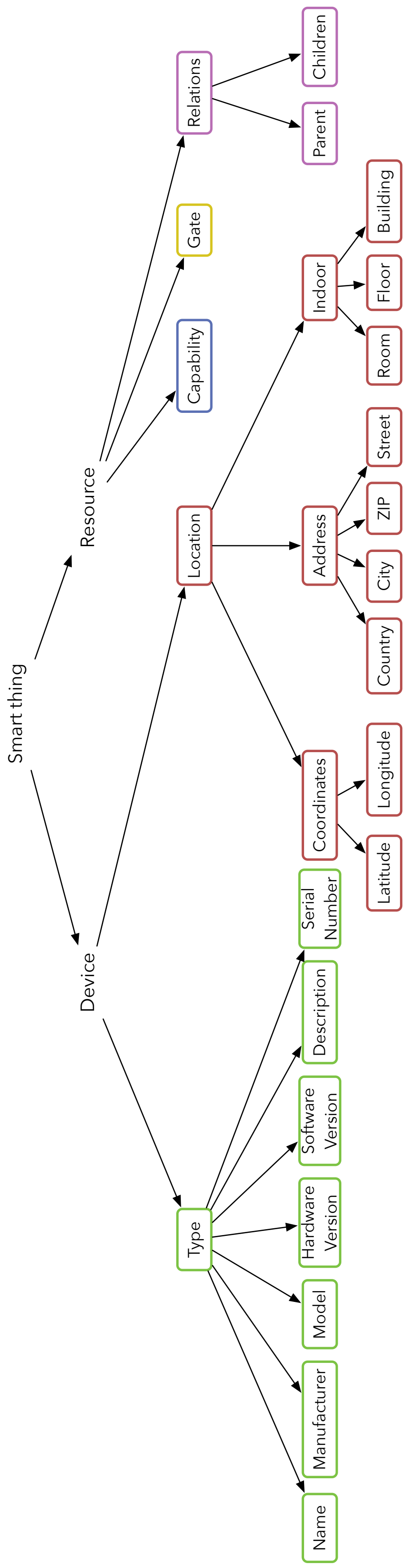


Figure 4.17: Our description model encompassing properties for sensors and actuators. This model is not fixed and can evolve according to additional needs. See also Figure 4.18 for an illustration of the capability properties and Figure 4.19 for the notion of gates.

Type Each sensing or actuating device is associated with a device type. This sub-description informs a user about the platform that is providing the functionalities. While this information is not crucial when consuming resources, it can give interesting insights about what kind of devices are used in the building network. It can, for example, also serve to detect outdated software versions.

The type sub-group is very similar to a basic product description, allowing to reuse existing ontologies. While it exists plenty of them describing products, we opted for the GoodRelations [318], and this for several reasons. Unlike other ontologies, it remains quite easy to have an overview of the classes and relations available to build the description. Another important aspect is that this ontology has been used by various major actors of E-Commerce (BestBuy, Sears, O'Reilly) to describe their products [319].

Model property	Class	Property	Description
Name	gr:ProductOrService	gr:name	Human-friendly name for the device
Manufacturer	gr:Brand	gr:name	Manufacturer of the device
Model	gr:Offering	gr:name	Specific model name
Hardware version	gr:Offering	gr:hardwareVersion	Hardware version **(custom property)**
Software version	gr:Offering	gr:softwareVersion	Software version **(custom property)**
Description	gr:ProductOrService	gr:description	Human-friendly description of the device
Serial number	gr:Offering	gr:serialNumber	Unique ID for identifying the device

Table 4.2: Type sub-group properties can be represented by leveraging the GoodRelations E-Commerce ontology.

The GoodRelations ontology provides a complete set of classes ranging from the product itself, the price model, up to payment methods. All this information is currently unnecessary in our description model. They however can be included to trace the origin of devices forming the building network. A part of the GoodRelations ontology is shown in Table 4.2. We added to the ontology class *gr:Offering* two properties related to hardware and software versions that are necessary when dealing with electronic equipments. In Listing 4.4, we show how a building automation device can be represented using the Turtle Notation 3 [320], leveraging the GoodRelations ontology.

```
<http://temp.kitchen.home.com/> rdf:type gr:ProductOrService;
  gr:name "Temperature" ;
  gr:description "Temperature from -20 to +60 Celsius" ;
  gr:hasBrand[
    gr:name "Eltako"] ;
  gr:includes[
    gr:name "SR65TF251" ;
    gr:harwareVersion "1.2" ;
    gr:softwareVersion "2.3a" ;
    gr:serialNumber "sflkj498fk3"] .
```

Listing 4.4: Describing a device type with the GoodRelations ontology in Turtle Notation 3.

Location Providing locations in buildings is an important aspect that has many consequences on systems. First, it allows to geographically locate devices in a given space and to reference measurements by their locations. This is necessary for example in data mining analysis or for machine learning applications. On the other hand, the location can be used to identify devices requiring human interventions. This can be very useful in buildings equipped with hundreds or thousands of devices.

Three different kinds of geographical locations can be referenced, depending on the required precision. Coordinates allow to precisely locate an outdoor equipment using GPS. It can serve for smart city devices or to give the coordinates of a building, and can as consequence be shown on a map. In the same way, a device can provide a full address, a street, or an area location where it is installed. This is useful for installations covering more than one building and for smart cities. These location representations however only refer to external positions and do not allow to identify a device inside a building. For this purpose, we propose to include the relative location approach that relies on the building morphology and the room arrangement.

```
<http://temp.kitchen.home.com/> a vcard:Individual;
  vcard:hasAddress [ a vcard:Home;
    vcard:country-name "Switzerland";
    vcard:locality "Fribourg";
    vcard:postal-code "1700";
    vcard:street-address "80 Bd de Perolles" ];
  hasGeographicLocation [
    latitude "46.792194";
    longitude "7.160291" ];
  hasIndoorLocation [
    building "D";
    floor "20";
    room "15" ].
```

Listing 4.5: Describing a device location with the hCard ontology in Turtle Notation 3.

The hCard microformat [321] is widespread over the Web to represent people including their address. As no similar ontology exists in RDF, some propositions have been made on how it should be translated into the Web Ontology Language (OWL) [220]. A tentative OWL version is already available on the Web and used in our model implementation. Regarding the relative indoor location, apart from academic projects, no ontology describes the morphology of buildings. We therefore defined our own OWL classes representing the main types of rooms allowing to position a device in a building. Table 4.3 shows how locations defined in our description model can be represented in the Semantic Web using ontologies. In Listing 4.5 we provide an example how the location of a device could be represented.

Capability Before consuming a Web service, a client has to discover what the service performs and what is its purpose. This is important in the context of building automation systems as plenty of different resources are exposed by sensors and actuators. A client must be able to distinguish what the resource represents and if it is intended for sensing or actuation. The capability is dependent on a resource as a single device can measure different physical properties, as for example CO_2 sensors often also measure the temperature and humidity in addition to the particles concentration. Figure 4.18 shows the description model part related to the capabilities of a resource.

Devices are often classified depending on what task they perform in the building automation system. This is a precious information as switches can be used for many purposes. Without this property, a client would not be able to distinguish what is the task of the device: *does this switch*

Model property	Class	Property	Description
Coordinates	GeographicLocation	latitude, longitude	External coordinates where to find the device
Address	Address	country-name, postal-code, locality, street-address	Address of the building where the device is located
Indoor	IndoorLocation	building, floor, room	Indoor location of the device **(custom property)**

Table 4.3: Location sub-group properties can be represented by leveraging on the hCard microformat translated ontology. We propose an extension of the properties to describe indoor locations of devices.

control the light or the blinds? As no ontology describes these functional categories, we defined our own making rational choices by considering the current offering of building automation system device manufacturers. Note that our categories are voluntarily kept simple in Figure 4.18 in order to represent them graphically. However, we allow for more detailed descriptions by adding levels in the description model, which enables to have better insights on the usefulness of a resource.

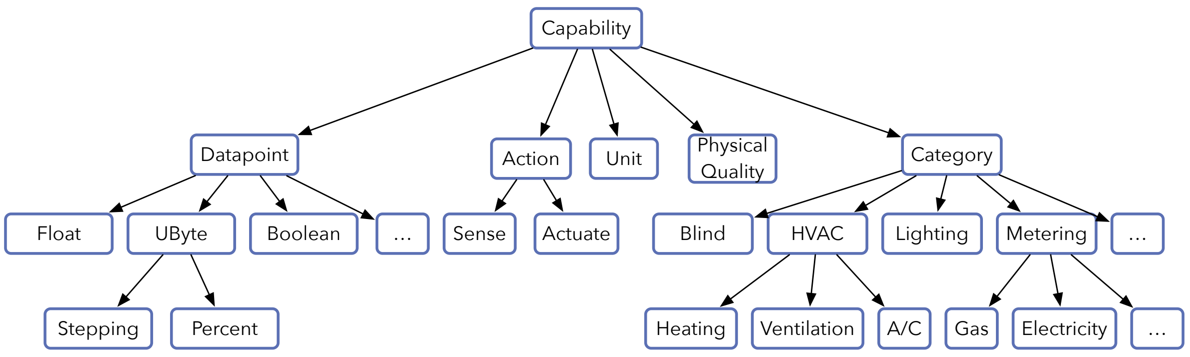


Figure 4.18: A capability describes the actual purpose of a resource. By reading this description, a client is able to understand if it can actuate or sense and how it has to interpret the value.

Physical devices may offer several functionalities that can be either sensing or actuation actions. For example, a humidity sensor only encompasses a sensing action while a power outlet can offer sensing (voltage, current) and actuation (switch on or off). To distinguish the type of feature, the *Action* class gives insight if the capability represented by the resource is performing sensing or actuation.

The physical quality property indicates what physical aspect of the environment is represented by the resource. Including this information in the description model allows a client to easily compose mashups between devices that are of the same type. For this purpose, we propose to use the UCUM ontology [322], defining more than fifty physical properties.

The unit of a sensing device should be clearly described to avoid ambiguities or confusions. Confusing temperature units between Celsius and Fahrenheit for example can have unpleasant consequences on the temperature management in a building. The UCUM instances ontology [323] provides an OWL class for a majority of commonly widespread units. These are used in our model instance to inform which unit the resource refers to.

Finally, interacting with a resource requires to know what the payload means and how the value is represented. As described in Section 2.2, most building network systems (KNX, EnOcean and BACnet among others) define their own data representation, also called datapoints. These datapoints describe several properties of the exchanged data, including the type (integer, float, etc.), the maximal and minimal accepted values as well as the accuracy and the sensitivity. In our description model we define classes allowing to recompose each possible datapoint. In Figure 4.18 we show how the datapoints are modelled. The Semantic Sensor Network (SSN) [324] ontology along with the DOLCE UltraLite (DUL) [325] are precisely targeting to describe sensor endpoints and are therefore used in our model. Table 4.4 shows how we use the various ontologies to build the capability description. More information is given in Section 4.2.2 about how interoperability is achieved between different manufacturers.

Model property	Class	Property	Description
Datapoint	ssn:MeasuringCapability	ssn:MeasurementRange, ssn:Accuracy, ssn:Precision, ssn:Resolution, ssn:Sensitivity	Define the representation of the data
Action	Action	sense, actuate	Informs if the resource represents a sensor or actuator **(custom property)**
Unit	UCUM instances classes	-	Unit of the returned value
Physical quality	UCUM classes	-	Type of physical measurement
Category	Categories classes	-	Purpose of the sensor or actuator **(custom property)**

Table 4.4: Several existing ontologies are used to build the property sub-group of a description.

Gate We here introduce the concept of *gate*, shown in Figure 4.19. Gates target to answer the following question: *how should I (human or machine) interact with this resource?* Most semantic descriptions only describe the purpose and the properties of a physical device. Describing the way how services should be consumed is also necessary and, to the best of our knowledge, was never addressed in the context of smart buildings. Adding such a semantic description will however contribute in the overall autonomy of a system offering the possibility to machines to compose by themselves mashups.

Augmenting RESTful APIs with service descriptions is subject of two main propositions. The RESTdec [326] initiative shows how the Semantic Web can be used to annotate hypermedia APIs. It aims at automating the consumption of services through agents capturing the functionality of the API. It does not define a new ontology but reuses existing vocabularies and shows how to interpret these so that a machine can discover the functionality by defining a new formalism for RDF. Trying to offer a description formalism similar to WSDL, the RESTful Service Description Language (RSDL) [327] defines in a XML schema properties of RESTful interfaces such as the methods, parameters and status codes. Clients can automatically generate functions to interact with resources by consuming the XML description (similar to WSDL and `WS-*`). While this schema seems very interesting, it comes with two important drawbacks: the use of XML as format instead of the Semantic Web, and its strong relationship with HTTP. For these reasons, we decided to create our own description that takes the strengths of both solutions.

We enhance our description model with properties aiming to facilitate and even to automate the consumption of resources exposed by sensors and actuators. We first add a property indicating what application protocols are supported by the resource. Clients can thereby switch to their preferred protocol automatically. In the same way, the semantic description can expose what notification mechanisms are supported, which gives clients more flexibility. Moreover, it provides a solution for clients using HTTP callbacks to discover which resource they should address for registration. The path of the registration resource is furnished within the description, allowing for each device to define its own registration resources.

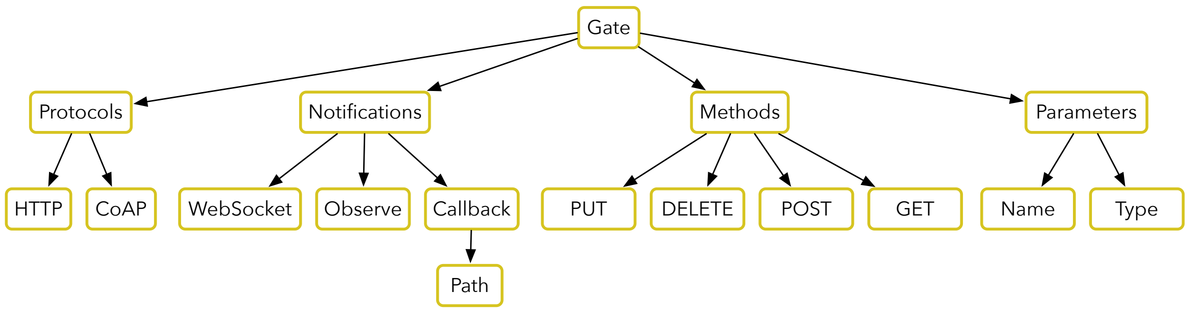


Figure 4.19: The gate properties describe how a resource should be consumed. This can serve as API documentation for humans building Web applications, but is especially intended to allow automatic mashup composition between machines.

Web resources can be manipulated according to certain operations (see Section 3.2.4) specified in the request. Although HTTP provides the `OPTION` method to retrieve the allowed operations, CoAP for its part does not offer such a mechanism. To overcome this limitation, we propose to include this information in the description model so that it will also be possible for CoAP clients to retrieve the list of supported methods for a resource. By knowing these methods, a machine can for example automatically infer how it can interact and can determine if the resource is read-only or allows updates.

Finally, URIs can hold parameters to filter data contained in resource collections. Application developers need an API documentation to know the filtering possibility. Machines for their part are not able to look for an API documentation, and even less to process and understand it. This lack can be bridged by providing in the description the allowed parameters along with their type.

In Listing 4.6, we show how a gate can be described by relying on our description model. In this example, the resource supports both CoAP and HTTP, along with the observer pattern and HTTP callbacks. For the latter, the resource used for registration is mentioned. Clients can issue either `GET` or `PUT` requests with one parameter of type Integer.

```
<http://temp.kitchen.home.com/> rdf:type gt:Gate;
  gt:protocol gt:CoAP ,
    gt:HTTP ;
  gt:notification gt:Observe ,
    gt:Callback .
gt:Callback gt:path "/observers" .
<http://temp.kitchen.home.com/> gt:method gt:GET ,
    gt:PUT .
<http://temp.kitchen.home.com/> gt:parameter _:param1 .
_:param1 gt:name "limit" ;
  gt:type rdf:Integer .
```

Listing 4.6: Describing a gate in Turtle Notation 3.

Relations Resources are organised to form a tree and require a mechanism to recover this structure. Clients having no prior knowledge need to browse the tree from the root resource in order to look for potential interesting resources. We include in our model two properties, parent and children, allowing to navigate within the tree.

Browsing Descriptions

By sharing the same description model, devices are able to understand each other's capabilities and general properties. In order to achieve this global agreement, the way how descriptions are retrieved also needs to be explored.

In order to comply with the REST architectural style, retrieving descriptions should not depend on the application protocol. Clients using CoAP or HTTP should use the same application protocol-agnostic mechanism. Though both protocols are RESTful, they do not offer the same header options. While HTTP has an `OPTION` method returning the allowed operations (that could potentially include in the payload the semantic description), CoAP does not provide such a mechanism. Another requirement of REST is being able to negotiate the content. HTTP is very flexible as the content-type is specified in text format whereas CoAP uses numbers identifying the content type from a restrained list in the specification. This situation avoids from leveraging the richness of RDF descriptions that can be serialised in many formats such as Turtle [328], Turtle Notation 3 [320], N-Quads [329], JSON-LD [330] and RDF/XML [331]. These are indeed not defined in the CoAP specification.

In the Web of Things architecture [48], Guinard proposes another way to select the content-type by considering it as a file extension. For example, consuming `/generic-nodes/1/sensors/temperature.json` will have as effect to receive a payload formatted in JSON.

We propose to rely on a similar approach that is protocol agnostic by specifying the description format in the URL. We however only apply this to the content negotiation of semantic descriptions and not in a general manner for the payload content. Clients are thus able to select the serialisation format of a description by adding the standard file extension related to the format at the end of the path. For example, sending a `GET` request to `coap://temp.kitchen.home.com/.n3` will return the semantic description in Turtle Notation 3. We do not recommend to use XML-based serialisations due to their large verbosity and memory footprint.

With the ability to retrieve descriptions, clients can henceforth begin to browse them by using the relations properties. Application developers are able to look for interesting resources on a device simply by following links in the same way as for Web pages. Furthermore, machines are able to crawl smart things in order to gather knowledge and possibly to index the resources in a lookup database by extracting the descriptions from the parent and children URIs.

However, browsing and crawling resources has an unavoidable drawback. Retrieving the semantic descriptions of resources requires many HTTP or CoAP calls to be issued. Machines crawling resources could impact the network throughput and could cause some congestions. This is especially true when using HTTP that is more verbose and not optimised for constrained devices. Moreover, a discovery mechanism by following links is not an alternative to a real discovery service. Clients need indeed to know the existence of a resource before they can browse it, implying either a manual configuration or a central registry that can be avoided with discovery. In the next section, we show how our description model can be used to build a semantic-based mechanism that goes beyond simple discovery.

4.1.4 A Semantic-Oriented Query Engine

As smart buildings can potentially hold a tremendous number of devices, it becomes important to supply application developers with a discovery mechanism. In order to optimise application development and mashups composition, the discovery system must rely on a straightforward architecture allowing to rapidly target interesting smart things.

The Web is a compelling example where users rely on search engines to lookup Web pages matching their search criteria. Search engines such as Google and Bing crawl the Web and index the pages they find according to topics and keywords. However, as previously mentioned, crawling is not adapted to the context of sensor networks. In addition, smart things can appear and disappear as well as change their context, which would result in outdated indexing. Search engines would therefore return obsolete results.

On the other hand, searching for relevant Web pages by providing keywords is an interesting concept giving much flexibility to users. Some discovery mechanisms used in pervasive environments like DNS-SD [190], UPnP [221] and SLP [222] tend to imitate this behaviour. Although they allow to filter the results by specifying a scope, it remains hard to target very specific devices [211, 208].

In this section, we propose a distributed query architecture allowing fine-grained lookups according to parameters, as presented in [4]. We overcome the potential congestions due to crawling and due to the single point of failure of search engines by performing end-to-end discovery. Our proposition takes benefit from the Semantic Web and relies on our description model as common ground.

Requirements

From the perspective of discovery, building automation systems are significantly different from classical pervasive devices for which simply locating the existence of services is most of the time sufficient. Actually, an extension from simple discovery to query is here required. Indeed the context of a device or, in our case, a resource is decisive during discovery. Application developers and also machines are targeting very specific devices when composing mashups or implementing automation rules. Many building automation systems have to look for devices being in a certain state or context for regulation or alarming purposes. For example when creating alarm strategies, only particular devices are of interest. A building administrator would look for motion sensors located in a specific room and being able to report slight movements.

In order to bear the dynamic of sensor networks, a discovery architecture should avoid any registration mechanism before making devices discoverable. This augments availability as devices that have not yet reported and are not known by the system will nevertheless be part of the architecture. Relying on a plug-and-play approach allows to have highly dynamic systems requiring no previous knowledge of the available resources.

Working with constrained devices and low-power networks, the energy efficiency of the new layer is also a key factor that has to be optimised in order not to affect life span of battery-operated devices. This concerns particularly the traffic footprint of the discovery architecture that has to be kept as low as possible.

We can list a minimal set of requirements for a discovery architecture in the context of smart buildings as follows:

Optimised for constrained devices: Smart buildings are composed of IP-enabled sensors and actuators relying on electronics offering only few computational power. Other equipments based on KNX, BACnet or EnOcean technologies are typically not IP and will require light gateways to expose the services in the IP world.

Plug-and-play installation of devices: Devices should be integrated in the existing architecture with no human interaction. The simple act of being present should suffice for a device to be searchable.

Discovery of the entire network: For management purposes, an overview of the entire network should be retrievable in a simple manner.

Selection of devices according to some contextual parameters: Contrarily to classic pervasive environments where only static parameters are needed to look up for specific services, a more dynamic approach is necessary in the context of smart buildings. Users should be able to formulate queries by indicating properties through keywords, in a similar way as Web search engines.

Scalability and fault tolerance: A scalable architecture is required as networks in smart buildings will evolve over time, including new devices, themselves running under new technologies. The tolerance to faults is close to zero as building automation systems manage critical systems such as access, lighting and cooling that cannot undergo breaks.

Energy efficiency: The number of messages to run the discovery architecture should be kept as low as possible and should not endanger the proper working of networks.

The Semantic Web offers ways to standardise resource description and is therefore increasingly used as ground for discovery applications. Existing discovery mechanisms for pervasive computing only offer small flexibility regarding the filtering of requests, as resources are searchable according to a limited set of pre-defined parameters. Description models for their part are much more rich regarding the description properties, which can be reused for discovery purposes.

Several projects have already addressed discovery from a Semantic Web point of view [223, 209, 224]. The SPITFIRE [225] project is particularly interesting, fully reusing a domain ontology as discovery ground. Furthermore, it allows users to query by indicating properties. Although users can define their own queries, they are limited to static properties and the ontology does not describe gates. Because of this lack, clients are unable to look for devices supporting a particular application protocol. As these solutions do not meet entirely our requirements, we propose in this section a novel discovery architecture especially tailored for smart buildings.

Smart Thing Context

Each smart thing has its own context represented by the description model. The properties allow to understand what the device can offer as resources and what are its features. A context is meanwhile largely depending on the device itself and more complicated than simply providing functionality descriptions. It can also reflect the internal state along with other properties that are not defined in the description model. These properties differ from those described in the ontology by being mostly dynamic and changing over time.

We can distinguish two types of properties making a smart thing's context, as shown in Figure 4.20:

Static properties are described in the domain ontology and refer to values that do not or rarely change over time. For example the unit, or physical properties related to a resource are stable during operation. Other ones like the location can potentially change if the device is moved to another location inside the building.

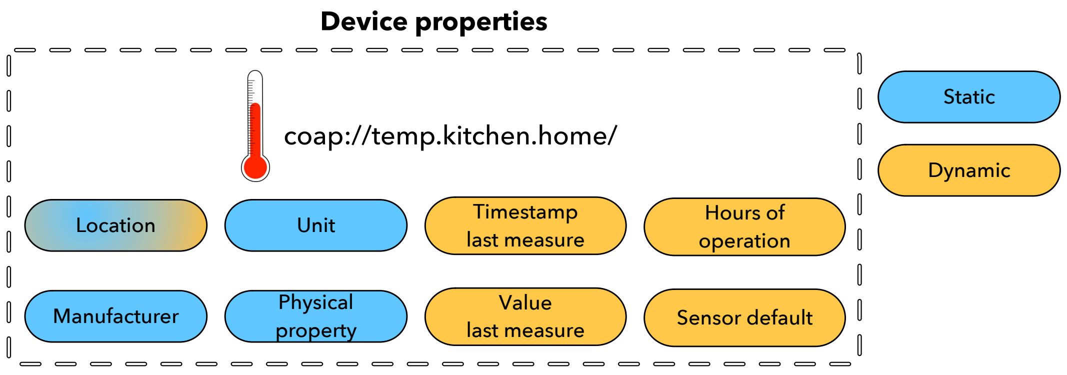


Figure 4.20: Device properties can be categorised into static or dynamic ones whether they are fixed or are frequently changing.

Dynamic properties reflect internal values that are changing during the operation of the devices. These are not necessarily part of the description model and are rather dependent on the device itself. They can for example provide information about current measured values or internal conditions.

The dynamic properties can also be very useful when an application intends to discover devices according to some internal representations. For example, one could search for all temperature sensors having a value above a given threshold. Other interesting types of properties can help in the maintenance of the building network. An administrator could discover all the devices having exceeded a certain number of hours of operation, or for relays having surpassed the maximum allowed number of switches.

From a broader view, dynamic properties allow to get knowledge on the network health and to anticipate the potential issues that could occur. Administrators could for example, scan the network and observe the evolution of the internal properties.

From Discovery to Querying

Classical discovery systems consist of detecting the presence of devices or services in the near environment. Protocols like UPnP [221] or SLP [222] provide basic filtering to discover the existence of devices like printers or media centres. This is achieved by specifying a scope in the discovery request. Only devices matching with the given scope will respond.

This discovery model is largely sufficient in an home context. However it offers low flexibility with regards to smart buildings composed of hundreds or thousands of devices. Building automation requires to link devices with each other to perform actions. Before composing a mashup, application developers or machines must be able to look for very specific devices that could be of interest for that mashup.

To provide such a mechanism, discovery needs to undergo a fundamental change: querying the environment. The building network can be seen as a collection of resources that can be queried by formulating an inquiry distributed all over the network. The Semantic Web provides precisely a query mechanism to filter semantic descriptions with SPARQL [226, 227]. Originally thought to query RDF triplestores (RDF documents collection), SPARQL can be used to gather RDF descriptions of smart things matching given parameters [225]. This query language is similar to SQL in terms of functionalities but is especially made to filter RDF descriptions. The user can specify in the query what subject from the RDF triples to retrieve., as well as the parameters (subjects and predicates of the RDF triple) used for filtering purpose. This allows to receive back only specific parts of the description, or in our case simply the URL of the description matching the query. Listing 4.7 shows how an application developer or a machine could express a SPARQL query by relying on our description model.

```
SELECT ?node WHERE {
?node a ssn:Sensor ;
ssn:category HVAC ;
ssn:observes <http://purl.oclc.org/NET/muo/ucum/physical-quality/
   temperature> ;
ssn:hasUnit <http://purl.oclc.org/NET/muo/ucum/unit/temperature/
   degree-Celsius> ;
inBuilding "home" ;
inFloor "ground" ;
inRoom "kitchen" ;
rdf:value ?lv ;
FILTER (?lv > 25) .
}
```

Listing 4.7: SPARQL query looking for temperature sensors located in the kitchen having measured a value higher than 25° Celsius.

Existing implementations relying on SPARQL centralise the device descriptions into central directory servers. These servers will then further execute the requests on the stored description. We believe that this architectural style is not suitable for building networks. It first requires a procedure during which devices will announce themselves and push their description in a central repository. Furthermore, the centralisation approach will cause tremendous traffic to be exchanged between devices and the central repository, as the RDF description will have to be updated every time a dynamic property changes. We therefore propose to store the RDF descriptions on the smart things themselves, while endowing them with a SPARQL processing engine.

A Distributed Query Resource

The major drawback of the SPITFIRE architecture previously presented lies in the centralisation of the descriptions in repositories, causing some issues with regards to our smart thing context. Updating the descriptions that contain dynamic properties can potentially result in a high traffic load on the network. We therefore reconsider the basic architectural concepts and propose to migrate towards a multicast query solution.

In our vision, devices can be endowed with more responsibility than performing a simple sense and react pattern. Instead of relying on central repositories performing the queries on the RDF descriptions provided by smart things, we deport this process directly on the devices themselves. The central RDF triplestore will be replaced by a distributed architecture where each smart thing participates in the common effort by being able to process SPARQL queries on its own descriptions.

In order to distribute the triplestore, we leverage the CoAP group communication specification [159], allowing to share resources over several devices. Every device embedding the Management Level discovery block will include in its RESTful API a service dedicated for querying. This special service should be exposed as direct child of the root resource as follows `/query` and be bound with a predefined IP multicast address.

As shown in Figure 4.21, a client querying for resources will issue a `GET` CoAP request to the query resource by including two important informations. First, it will choose how the RDF

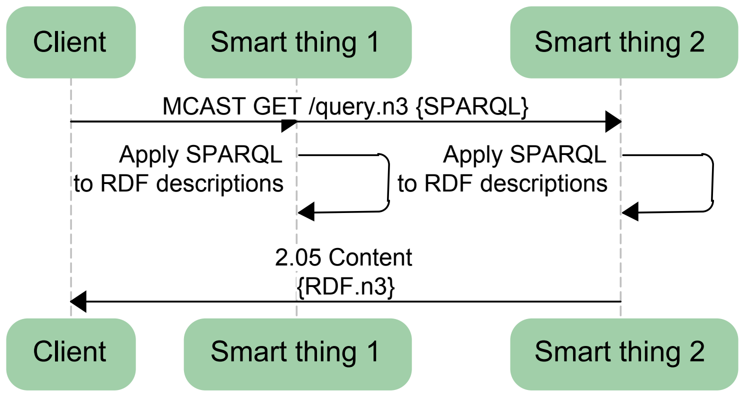

Client
Smart thing 1
Smart thing 2
MCAST GET /query.n3 {SPARQL}
Apply SPARQL to RDF descriptions
Apply SPARQL to RDF descriptions
2.05 Content {RDF.n3}
Client
Smart thing 1
Smart thing 2

Figure 4.21: Clients will send their SPAQL queries over a predefined multicast group. Every smart thing in the network checks its RDF descriptions and will only respond if there is a positive match (Smart thing 2).

description should be formatted according to the concept explained in Section 4.1.3. Then, the SPARQL query will be provided in the payload of the request. All smart things in the smart building will receive the request and check the RDF descriptions of their resources to look for matching ones. In order to not overload the network, only devices having a positive match will respond to the client by providing the RDF description in the appropriate format.

Such a fully distributed approach has three main advantages over the classical centralisation of SPITFIRE. First, we can guarantee that the dynamic properties included in the descriptions are up to date as they come directly from devices and not from a repository. Second, devices are not required to push a new version of their description to the central repository each time a dynamic property changes. Finally, using a predefined multicast group impersonates the communications and allows a plug-and-play discovery model where no prior knowledge of the infrastructure (essentially central repositories) is required. In order to benefit from these advantages, devices obviously have to embed a SPARQL engine. This requirement does however not represent a limitation, as will be shown later in this section.

Architecture

In this section, we describe an experimental software architecture in accordance with the aforementioned concepts. As shown in Figure 4.22, the client and the server are composed of several modules. This architecture enables flexibility and makes modules easily exchangeable with other implementations.

Interface As previously mentioned, we use CoAP as application layer to interact with resources and therefore use our hybrid REST server library to register and expose the `/query` resource. The CoAP server listens on a multicast socket with preconfigured port and IP address. Its role is to dispatch incoming requests to the discovery and query engine. If matches are found, it will respond to the source of the request with the related RDF description in the appropriate format.

The client part consists of the microcoap [310] library offering C functions to build CoAP queries. The responses are then forwarded to a user-defined callback function that in our case will be the response parser.

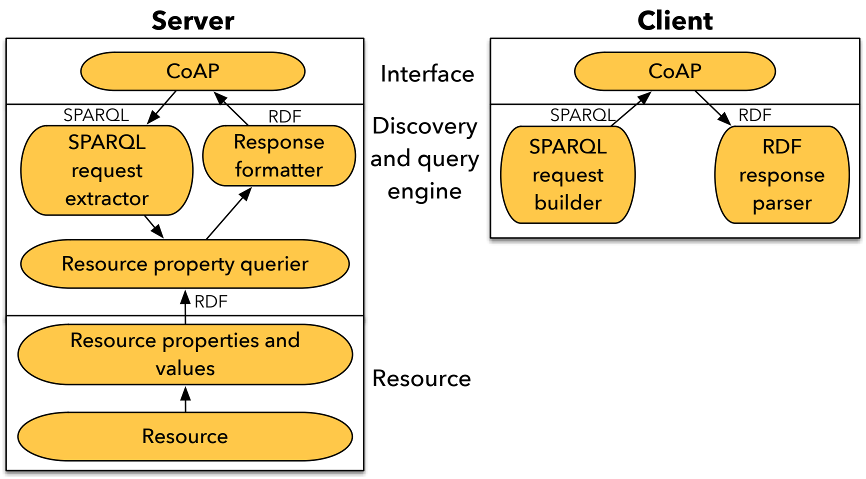


Figure 4.22: Modular architecture of the Semantic Web query mechanism allowing to discover particular devices according to dynamic and static parameters.

Discovery and Query Engine Going one level down in the server, the discovery and query engine is responsible to evaluate requests and to find corresponding matches. It starts with a SPARQL request extractor that will take out the query from the CoAP message and performs some validations regarding the format. Once the request is considered as well-formed, it is passed to the querier.

The querier will gather the RDF representations of all the resources present and available on the device. Once the collection is complete, the SPARQL query is applied to each RDF representation of the collection. Each match is then stored in a collection of results. If this collection is empty after querying, the process can stop at this step. Otherwise, the collection is forwarded up to the response formatter. This module iterates through the results collection and formats SPARQL responses according to the type specified by the client. The last step consists of responding to the client over the CoAP interface by including the descriptions in the payload.

For the client, the process is inverted and begins with a SPARQL request builder that will format the query in an appropriate format that can be validated. Once the request has been sent, it waits a certain amount of time for responses. With CoAP group communication in conjunction with SPARQL queries, it is not possible to know in advance which devices will have matches and will respond to the query. The RDF response parser collects the RDF descriptions coming from the clients and waits until the timer elapsed to process them. Finally, it returns a collection of results to the user.

All this part is handled by relying on the Rasqal RDF query library [332]. This library written in C offers interesting functions to process RDF documents and SPARQL queries. While a large variety of RDF formats are natively supported, we refined the library to only support the most lightweight ones: Turtle, Turtle Notation 3 and JSON. We decided to exclude all the other formats like RDF/XML and HTML because they require too much memory for processing.

Resource Finally, the Web resource has to provide its entire description in form of a RDF document to the lower module. Each time the description is retrieved by either a remote client or internally by the query engine, this module will collect the actual state of dynamic properties and places them inside the RDF document. In order to save computing power, this document is only updated when required and not every time a dynamic property changes.

Automatic Mashup Composition

The ultimate aim of semantics is to automate interactions and service composition between various applications. In the same perspective, sensors and actuators can benefit from semantics by being able to discover potential partners and to build mashups automatically with no human in the loop.

Since our proposed ontology not only describes resource properties but also so-called gates, we open the path for a new self-composing sensor network. In classical approaches, administrators or developers are required to manually link devices with each other, following a pre-configured and non-scalable interaction mechanism. For example, they have to choose between HTTP or CoAP, the available parameters, their position and meaning. From now on, scenarios where a new sensor can integrate itself in an existing environment by discovering neighbours are possible.

We can imagine an application scenario for a room where lights and blinds will be controlled according to the illumination level and presence indication. The illumination sensor being responsible for the whole lighting will send a query to discover potential partners that are located in the same room. Since each device describes its gates, the illumination sensor will know how to register for notifications on the presence sensor and to turn the light on or off, in addition to drive the blinds.

This could be a future solution for self-configuring home automation networks, where users would simply configure strict minimal properties such as name and location. The mashups and interactions between devices would be done automatically inside the network by leveraging the semantic descriptions.

Evaluation

In [228] some performance tests have shown that executing SPARQL queries on constrained devices implies a longer processing time than on powerful machines. According to their experimental results, it takes in average 190ms to process a relative simple query like the one shown in Listing 4.7 with a collection of five RDF descriptions hosted on the device. Although this time may seem long, it does not restrict the usage of the Semantic Web directly in sensor networks, as it will only be used for discovery purposes, which is not time critical.

In our evaluation, we focus rather on the quantity of messages exchanged that are required to perform the queries. We especially want to assert that using a distributed multicast approach can be more efficient than the centralised one used in SPITFIRE. The context of each resource can hold dynamic properties that potentially change frequently, hence resulting in a lot of update messages to synchronise the central repository.

In order to have an overview of the communication costs of our multicast-based approach in comparison to SPITFIRE, we simulated both by reusing the same configuration as for the Distributed Naming in Section 4.1.2. We thus simulated three buildings with a total of 180 rooms, with an average of 13 devices per room. The descriptions are updated in average every 10 minutes with a RDF description of 750 bytes. This value was selected to reflect the expected update rate of dynamic properties. We foresee that the number of queries issued by users is insignificant with regards to the ones sent by devices to discover potential mashup partners. We therefore set this value to 100 similar queries per hour of a 320 bytes long SPARQL. Each query was targeting only one device returning the description of a single resource. This gives us a single response containing a description of 750 bytes.

In Table 4.5 we compare the results of a 24 hours simulation for both approaches. The first

	SPITFIRE		**Distributed triplespace**	
	Messages	Data [MB]	Messages	Data [MB]
Description updates	67392	26.2	0	0
Queries	4800	2.6	4800	2.6
Total	**72192**	**28.8**	**4800**	**2.6**

Table 4.5: Comparison of the number of messages and application-level data between the centralised SPITFIRE solution and our distributed triplespace approach.

obvious observation is that both have the same impact regarding the querying of descriptions. This can be explained by the fact that both solutions are relying on CoAP at the application level and send the same number of queries. The 2.6 MB of exchanged data include the SPARQL query payloads, the RDF descriptions contained in the response and the CoAP headers.

The primary difference is related to the update of the dynamic description properties. In the case of SPITFIRE, each update will cause the device to push the new description to the central repository. This generates a tremendous number of messages, each one containing the RDF description, resulting in 26.2 MB of traffic at the application level. In contrast, our distributed triplespace avoids this issue by storing the descriptions directly on devices, resulting in no traffic when dynamic properties change. The distributed architecture globally reduces the number of messages and the data by approximately 90% compared to the SPITFIRE architecture.

There is however some uncertainty about the impact of multicast communications that is rather difficult to quantify. While they do not represent an issue over wired communications as for Ethernet, wireless technologies like Wi-Fi and IEEE 802.15.4 have to forward packets to every node in the network. This can lead to congestions and increases the energy consumption. As the queries are only used sporadically by users and devices, there is only little probability that the overall proper working of a wireless network will be affected. Moreover, the gains obtained by relying on multicast avoiding to update the descriptions should be much more beneficial than the overhead caused by multicast communications.

4.1.5 Related Work

To facilitate the development of RESTful APIs and their integration in applications, libraries try to reduce the required efforts by handling all server-related tasks. Developers declare their API by indicating the location-path, the methods, and the parameters with possibly other information. The library then handles the requests and calls the appropriate functions containing the implementation of the services. HTTP being at the origin of REST, many solutions are available and this for various languages. Some of them require to be coupled with an underlying HTTP server such as for Java Jersey [307]. Other ones like Django [333] in Python, or node.js [334] which is renowned for its performance, already embed all the HTTP handling. Due to their size and memory requirements, those implementations cannot be embedded on constrained devices. While it exists tiny HTTP libraries [312, 313], they only parse HTTP requests and do not allow to define RESTful APIs.

With the emergence of CoAP in sensor networks, some libraries especially targeting constrained devices have appeared. They usually are written in the C language and are only compatible with specific platforms, such as Erbium [335] for the Contiki operating system. Other ones like libcoap [336] and microcoap [310] are not dedicated to a particular platform, but they must be adapted to the functions accessing UDP sockets. As current approaches promote the use of centralised servers to handle sensor data, libraries written in higher-level languages (principally Java and .NET) are available to build server-oriented applications [309, 337, 338].

Because HTTP and CoAP still have to coexist in sensor environments, smart things should be able to understand both protocols whenever possible. We joined this research and proposed in Section 4.1.1 a hybrid REST library mapping both protocols to an abstract representation.

When working with Web technologies, especially if humans are interacting with devices, it becomes mandatory to rely on a naming system. As discussed by Masdari et al. [191], the legacy DNS faces a set of shortcomings when used in sensor networks. Several propositions have since tackled this problematic through the development of fully or partially distributed naming systems. They consider the naming space either flat or hierarchical. In systems which are organised in a flat manner, names have no relation with each other, and can potentially be managed by any server [229, 230]. Other flat systems require any device to act not only as client but also as server [231, 232]. These works have led to the Multicast DNS [190] that however reuses the legacy DNS messages protocol.

The hierarchical systems keep the notion of domains and zones from the legacy DNS. They however differ by relying on a set of techniques to ensure mobility and availability in sensor networks. Nodes are designated to serve zones either statically or dynamically during runtime [198, 233]. Some naming systems base their behaviour on concepts of peer-to-peer networks to ensure a fully distribution in a self-organised manner [195, 234].

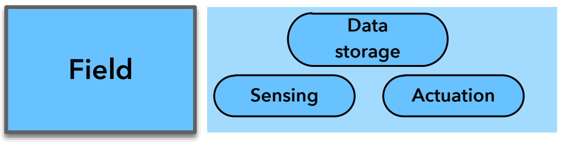
When considering interoperability between devices, the necessity to describe functionalities is a key component. As a consequence, several researchers have been looking at specific ways to describe sensors and actuators. SensorML [339] is a XML-based schema language that can be used to define sensor network applications and the related elements. In the same category, DomoML [235] is a schema that allows to describe all automation related tasks including resources (sensors, actuators and location), their aggregation and the interactions among them.

The Web also provides solutions to structure descriptive information of resources. Initially, the Semantic Web [210] was intended to annotate Web pages, making them consumable by machines. This topic of research has led to the Resource Description Framework [217] (RDF) providing some standard languages and formalisms. Since then, domain-specific ontologies were developed to describe knowledge of specific contexts including sensor networks [223, 236]. The Semantic Sensor Network Ontology [324] depicts generically sensor properties and functionalities in a reusable manner. These ontologies are an interesting solution when it comes to build an overall knowledge of different systems [218, 215]. With the REST architectural gaining momentum, some specific semantic descriptions methods were proposed to describe RESTful APIs [237] [327].

Our approach is to compose from several existing standard ontologies in order to form a reusable description model. To achieve this, we not only encompass static properties describing functionalities, but enhance our model by describing how RESTful APIs can be consumed. Furthermore, relying on the Semantic Web and RDF opens new dimensions for discovery services.

Being able to discover particular functionalities in pervasive environments is not a new topic of research. Zhu et al. [208] survey the different architectures that are used to discover services. Several protocols fulfilling requirements related to constrained devices and sensor networks have been proposed [212, 238, 205]. The DNS-SD (Service Discovery) [239] protocol is an evolution of mDNS, and is gaining momentum in sensor networks. It relies on the naming system and offers a discovery service by adding some properties to the records describing the type of services that are exposed by a device. With the increasing use of Web technologies on smart things, some projects tackled a seamless integration of Web clients [240, 241]. Central repositories store resource descriptions that are advertised by devices and retrieved by clients only with HTTP requests. Then, the Semantic Web was used to standardise descriptions in central repositories [224]. The SPITFIRE platform [225] is a reference implementation that combines the strengths of the Web and the novel RDF techniques, allowing to query resources with SPARQL [226].

4.2 Field Level



Sensors and actuators are spread all over a building to interact with the physical environment. The Field Level provides communication means for those devices. It encompasses the network topology and the application layer. The latter provides mechanisms for devices residing in the same network to exchange information and to execute simple tasks.

Smart buildings can nowadays be equipped with several building automation systems leading to heterogeneity and preventing a global management. In the vision of the Web of Buildings, Web technologies are intended to replace the variety of existing legacy protocols. We therefore address the connectivity of heterogeneous smart things by composing on Web technologies and on the notion of smart gateways to provide a transparent and reusable application protocol dedicated to smart buildings. Moreover, we propose that data produced by sensors, which is intended to be further processed by higher-level applications should not be centralised outside the sensor network. To do so, we consider smart things as a distributed storage space where historical data can be stored.

In this section we propose Web-based blocks tackling these challenges. We start by providing guidelines describing how RESTful APIs have to be exposed and organised on field devices. Furthermore, we detail the notion of datapoints, crucial to achieve a mutual understanding of data between different building automation systems. We also introduce the concept of smart gateways allowing seamless compatibility with legacy devices. Finally, we provide a cloud-like storage striving to reuse available resources to store historical data within the building network.

4.2.1 Web APIs for Sensing and Actuation

As already mentioned throughout this thesis, sensors, actuators and other smart things are able to interact with each other by using Web technologies. This is specifically achieved by leveraging the Web of Things and exposing services as resources [49, 51]. In the context of smart buildings, Web of Things paradigms are not sufficient to achieve interoperability among devices coming from various manufacturers. More guidelines are required to ensure that any kind of device can be seamlessly integrated in a sensor network. This especially concerns how the resources used for sensing and actuation have to be exposed to make them serviceable.

In this section we propose a set of guidelines that specialise the Web of Things for the Field Level of building automation. We start by explaining how the REST architectural style should be used and then move towards the description of resources. Some parts related to Web APIs have been published in [1, 2].

RESTful APIs as Federating Endpoints

To participate in the WoB, every device has to embed a REST server in order to expose resources to the rest of the network. These resources are identified by URIs having to follow a specific scheme: `<protocol>://<device_name>.<location>.<organisation>/<path>`.

Protocol denotes which application protocol is used for the interaction. While HTTP and CoAP are both RESTful, we however encourage to use the latter as it is optimised for constrained networks.

Device name gives insights on the type of device that hosts the resources. This name should be human-friendly and can represent the functionality of a smart thing, as for example `temperature, light or co2`.

Location identifies where a device is placed in an absolute location within the building. It has to be decomposed according to the building structure typically including floors and rooms.

Organisation is the last part of the host identification and allows defining to which organisational structure the device is attached, for example the name of a company having its own domain name.

Path points to the resource that is targeted by a request. It can either be the root resource or any of those exposed by the device.

If a smart thing exposes several resources, they should be logically organised as a tree whenever possible. In Figure 4.23, we show an example of resource organisation for a smart power strip. Resources having a logical relation among them can be structured so that the browsing is made easier for humans, almost like browsing directories on their file system.

Resource collections are very useful to group identical children identified by a number. This particularly fits the properties of sensors and actuators offering several identical actions, as for example multiple push buttons embedded in one device. When accessing a resource collection without providing the identifier of a specific child, the server should at least return a list of URLs pointing to the children resources.

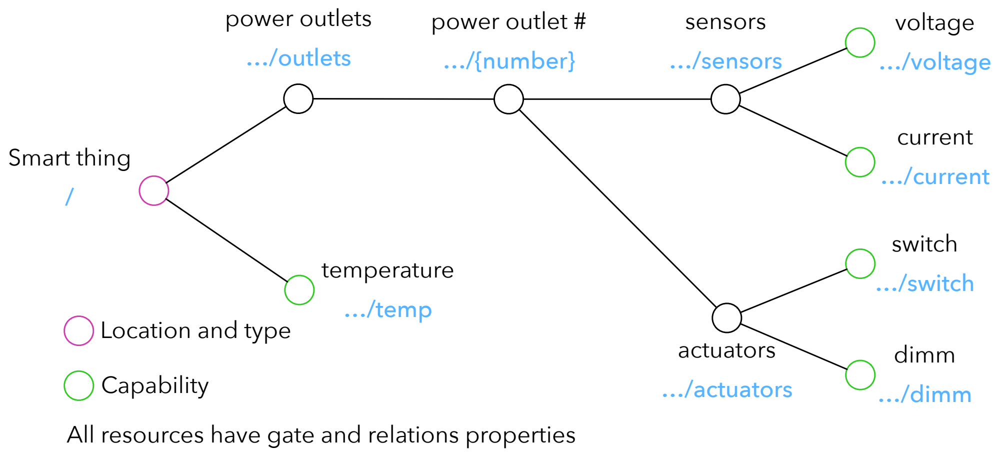


Figure 4.23: Resources on smart things can be represented with a tree like a file system. Collections can for this purpose regroup identical sub-resources identified by a number. In this concrete example we show how resources could be organised for a smart power strip.

HTTP and CoAP both define several methods for manipulating resources. For sensors and actuators, only two of them are useful. First, sensors should only allow to read measured values through `GET` requests. This restricts the interaction possibilities with a resource that is read-only. Resources that can be also updated with a new value, which is the case for actuators, must in addition support the `PUT` method. The new value for its part will be contained in the payload of the request.

Finally, in order to avoid the implementation of polling strategies, each resource should offer a notification mechanism. This requirement is mandatory for resources sensing the physical environment. Mashups applications and actuators require to be aware of the latest value for taking decisions and performing actions.

Describing Resources

As previously pointed in Section 4.1.3, descriptions have a capital importance when it comes to understand what a resource represents and to answer the question: *what can it offer to me?*

To envision a mutual comprehension of smart things, each resource must also expose its related description. Depending on the position of a resource in the tree, the description will encompass different property types, as shown in Figure 4.23. First, the *location* and *type* descriptions being related to the device must not necessarily be repeated in the sub-resources. We propose therefore to only describe the device-related properties in a description that is attached to the root resource.

The actual functionalities of a device will be located on the leaves of the tree, hence the requirement to describe the *capability* properties at this location. Every resource exposed on the RESTful API must include in its description at least one *gate* along with the *relations* to its parent and children.

This repartition of description properties allows to lighten the overall need for descriptions and to accordingly comply with the space requirements of constrained devices. Moreover, as only the leaf-resources perform useful tasks, providing fewer descriptions on the intermediary resources allows to speed up the browsing. Clients will reach the leaf-resources without spending too much time at interpreting and understanding unimportant ones.

4.2.2 Towards Homogeneous Building Automation Systems

As smart buildings are increasingly equipped with more than one building automation system, this situation leads to inconsistency at several levels. This makes it impossible for building networks to connect to each other from a physical point of view. Even if bridges allow to interconnect them together, they still use different protocol stacks, which can be assimilated to different languages.

To solve this limiting situation, we first propose to reuse the concept of datapoints crucial in building automation systems with the WoB. Then, we discuss how IP and Web protocols can be used to homogenise the application level around smart gateways.

Data Compatibility Through Datapoints

Building automation systems like KNX, EnOcean and BACnet faced at some point the problem of compatibility between devices coming from different manufacturers. Indeed, a device labelled as KNX-compatible should work with all other KNX devices regardless from the manufacturers. To guarantee such compatibility, all of them have standardised the data that is exchanged between devices, the so-called datapoints. More than a simple information about the type of data (float, integer, boolean, etc.), it reflects how the value has to be interpreted (a percentage, stepping, on/off, etc.)

The Web faced the same issue as the HTML used to visualise Web pages needed to be standardised so that every browser interprets and renders the code identically. The only way to achieve a mutual comprehension of data exchanged by different devices in the WoB also goes through standardisation.

Interoperability between building automation systems therefore requires some kind of standardisation. In the WoB, we reuse the notion of datapoints that will be shared as common data interpretation. Each resource associated with the physical environment (i.e. a physical value measured by a sensor or the actual state of an actuator) has to be associated with a datapoint. This information is encompassed in the semantic description of the resource.

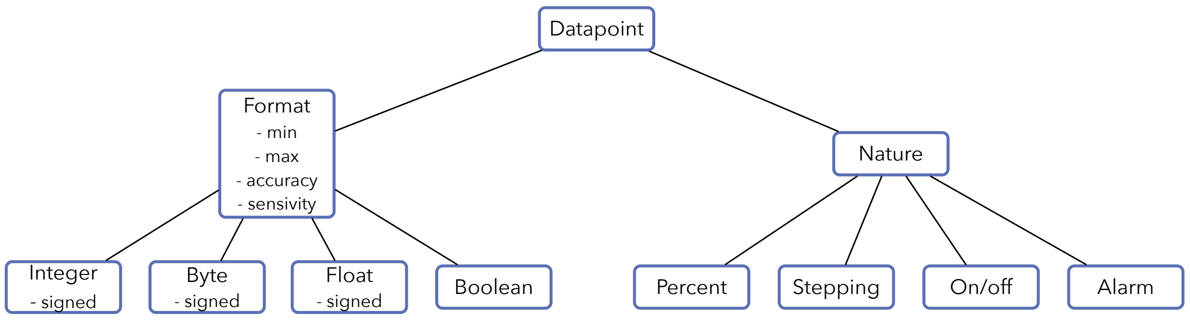


Figure 4.24: Datapoints are not predefined but can be composed by putting together different properties and classes. This approach offers much more flexibility compared to a finite set of definitions.

While one option would be to define a class for each possible datapoint as done in [94, 131], we envision to offer more flexibility. The idea to define in an ontology a finite set of all the possible datapoints used by several technologies only little opens doors for evolvability. We therefore opted for a more descriptive approach where device manufacturers can pick up several classes to compose the description of the datapoint. In Figure 4.24, we show the main classes along with the principal properties that are at disposal to compose a datapoint.

By using a descriptive approach rather than predefined datapoints, we can keep the ontology simple and flexible. New building automation systems can be integrated without the requirement of defining new classes, which increases evolvability. Using a finite set of datapoints in an ontology would not allow any evolution. It would indeed result in several incompatible versions of the ontology being distributed on devices. The heterogeneity would then be relayed at the semantic level, which is avoided in our proposition.

From Legacy Protocols to WoB Networks

With the emergence of the Internet of Things, smart buildings tend to be equipped with different technologies. Our proposition is to federate all these devices around the WoB. While this is easily feasible for IoT devices natively addressable over the IP protocol, legacy BASs do not offer any IP connectivity as they rely on their own network stacks.

Connecting Legacy BAS to the IP Backbone In order for legacy BASs to be able to participate in the WoB, they first need to be connected to the IP network for addressability and Web interactions. Figure 4.25 shows three different approaches with the following properties:

1. **Centralised server:** This approach is currently the most used one. A server at the IP backbone is directly interfaced with the building automation system and handles all the communications. Some of them expose SOA heavy `WS-*` for integration into enterprise systems.

2. **Protocol bridge:** The telegrams from the field level are encapsulated into IP packets. These bridges are often limited in the number of tunnels they can handle simultaneously.

3. **Smart gateway:** By implementing the field stack and the Web one, mappings between Web services and field endpoints are possible. Remote devices acting as clients are not aware that the final device resides in another type of network and even having no IP connectivity.

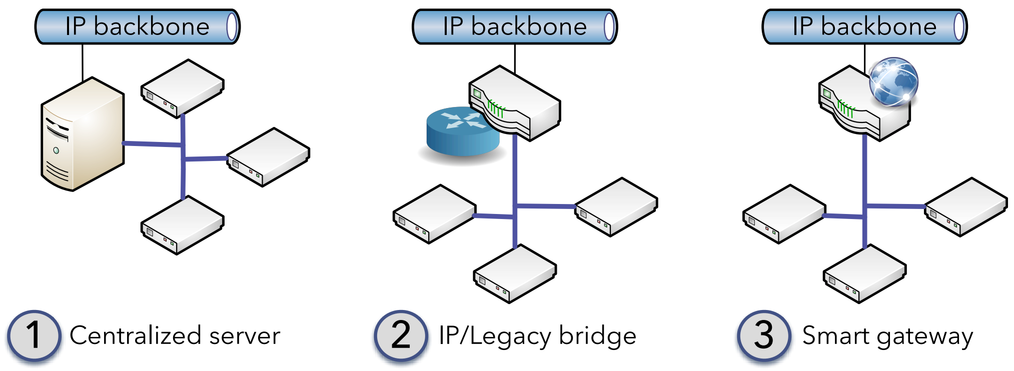


Figure 4.25: Connectivity with legacy building automation system can be achieved through several ways: a centralised server physically interfaced, a protocol bridge encapsulating telegrams in IP packets, or with a smart gateway mapping device functionalities to Web services.

The first two solutions are not adapted to the Web as they are either centralised or requiring the clients to understand the legacy BAS protocol. In contrast, smart gateways mask the complexity and specificities of the legacy BAS by mapping those functionalities to RESTful APIs. Some smart gateways have already been successfully deployed in home automation scenarios [150, 36]. For these reasons, the most straightforward and Web-compliant method for integrating legacy building automation systems into the WoB is by relying on smart gateways.

However, the existing Web gateways do not comply with our vision of APIs for sensing and actuation mentioned in Section 4.2.1. They typically expose a general API regrouping all the functionalities that are offered by the legacy devices. We aim at making smart gateways totally transparent, so that legacy devices appear as they would natively embed a Web server.

Classical smart gateways indeed structure their API in order to allow clients browsing the entire resource tree. A smart gateway therefore appears as a single device with many capabilities. It is not possible for a client to differentiate what device offers which functionalities. Although this can be useful for crawling, the dependency to the gateway is too strong and does not allow mobility. We advocate that even legacy devices can be first-class citizen of the WoB with no differentiation from native IP devices.

Mapping Resources to Legacy Functionalities In our proposition, to achieve transparency, a client does not have to know the legacy address of the device with which it wants to interact. Every legacy device will, in order to comply with this requirement, get its own name and host identifier like for every smart thing. This means that it will have its own IP address which will point to the gateway. Obviously, the IP of the gateway will be registered in the naming system several times but with different names.

A solution for this approach is that the gateway only exposes one endpoint that accepts any kind of path. Each request will be dynamically evaluated by using the internal mapping table where the host identifier is linked to a particular device in the legacy BAS. It can this way retrieve the legacy address and can use it to interact with the device. The path provided in the request is also mapped to a device endpoint if necessary. Furthermore, each device will appear as native WoB having its root resource exposed. For example, `temp.kitchen.home` and `co2.living.home` will point to the gateway, but by analysing the `Uri-Host` option, the gateway knows which device is meant and can return the appropriate response.

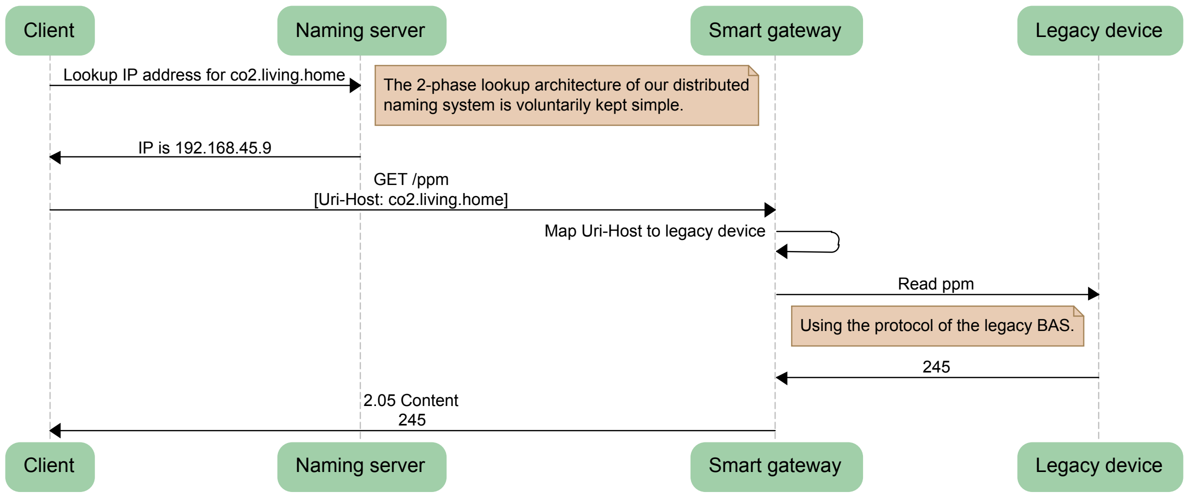

Client
Naming server
Smart gateway
Legacy device
Lookup IP address for co2.living.home
The 2-phase lookup architecture of our distributed naming system is voluntarily kept simple.
IP is 192.168.45.9
GET /ppm
[Uri-Host: co2.living.home]
Map Uri-Host to legacy device
Read ppm
Using the protocol of the legacy BAS.
245
2.05 Content
245
Client
Naming server
Smart gateway
Legacy device

Figure 4.26: Clients reuse the same interaction concepts with legacy devices. The request will however be handled by a smart gateway mapping the Uri-Host to an address. Once having performed the action with the legacy device, it will put the result into the response.

Exposing virtual WoB devices is a very important concept in our architecture. A description can be attached to the root resource of a virtual WoB device containing the device properties mentioning the legacy device and not the gateway platform. This totally decouples the legacy devices from the gateway and let them appear as native ones. This last observation is beneficial because it allows to use the same interaction concepts for native WoB devices and those connected to legacy building automation systems.

In Section 5.1 we report on concrete gateway implementations integrating KNX and EnOcean devices into the Web of Buildings. These legacy devices will not be identifiable as being behind a smart gateway by applying the aforementioned virtual WoB concept.

4.2.3 A Cloud-Like Storage for Historical Measurements

By applying the architectural design presented in the preceding sections, clients are able to interact with sensors and actuators in a straightforward manner. Many applications running in a smart building require more than accessing the current value of a sensor. Control loops and data-driven techniques targeting an intelligent adaptation of the building according to the users' behaviour make an intensive use of historical data [17].

Current approaches are deporting the storage outside the sensor network in a single database or using commercial cloud services. Although being viable solutions, they involve constraints which are often not acceptable for building owners. The first worry is about the privacy of sensitive data such as presence indication or electric consumption data. Secondly, forwarding the data outside the sensor network implies a gateway that is susceptible to be overloaded by the traffic and that represents a single point of failure, as well as a point of entry for attacks.

Considering those problems, it appears that the best solution for owners would be to disconnect the sensor network from the rest of the world, thus increasing security while ensuring privacy by storing historical data within the sensor network.

In this section, we present a mechanism for distributing historical sensor data on smart things, forming an in-network cloud relying on Web technologies as presented in [7]. Our system distinguishes itself from others by being fully autonomous and requiring no human in the loop. We put special attention on limiting the network traffic while ensuring maximum fail safety and scalability.

Requirements

With the advances made recently in electronics, smart things embed significant amounts of memory that are often underused. This is also the case for gateways having storage capacities of several megabytes or even gigabytes. All this memory can therefore be used to store data directly on smart things [242], specifically historical data in our scenario.

On the other hand, cloud storages are gaining momentum in data analysis applications. This is explained by their ease of use and the elastic scalability adapting to the customers' needs. Offered as SaaS (Software as a Service), clients typically interact with the cloud storage through a Web API or more traditionally with SQL. By doing so, they are not required to configure any particular behaviour of the system such as the hardware or software. The data replication is automatically ensured by the cloud storage without asking the client for any intervention. Furthermore, the storage capacity adapts itself to the current content and can increase or decrease accordingly.

In our architecture, we intend to reproduce this behaviour by providing a cloud-like storage using the already available devices in the building network. Historical data will be stored inside the building network on devices being able to contribute to the overall storage effort. Our solution should follow the SaaS concept by allowing devices to store data without any knowledge of the underlying infrastructure. Because of the dynamic of sensor networks, it is particularly relevant to ensure a data replication in order to guarantee availability. Moreover, the underlying infrastructure has to adapt automatically to the storage demands by either encompassing more or fewer storage devices.

Besides the already mentioned characteristics, we aim at complying with the following requirements:

Location-based storage: The data should, when possible, be placed closest to where it is produced to avoid traffic propagation over the whole network. This also ensures that a failure in a specific part of the building will not affect the rest of it. We hereby base our grouping according to the logical room structure of the building.

Replication for prevention: To ensure availability, the data should be at least duplicated on different servers. Because of the dynamic of sensor networks and to ensure that the cloud storage service is always accessible, the agents offering services must themselves ensure a smooth failover in case of failure.

Energy-efficient retrieval As data will be distributed and replicated inside the building network, it is necessary to rely on an algorithm that will retrieve the data efficiently.

On-demand storage: Not all sensor measurements need to be stored for further processing in a smart building. Moreover, some specific data need to be stored for a longer period of time than others. Clients should therefore be able to announce which resources have to be stored and for how long.

Web-oriented efficient application layer: To comply with our WoB framework, the services must be accessible by following the REST architectural style. In order to reduce the footprint of the storage system, the architecture must rely on CoAP as application-layer protocol for accessing RESTful APIs. CoAP is lighter than HTTP and contributes to the efficiency of the system.

Multicast-based communications: As previously seen throughout this thesis, multicast is interesting for synchronising simultaneously several agents. We therefore envision to reuse this pattern to reduce the network traffic.

As outlined in Section 4.1.2, the building morphology naturally fits as underlying topology. We therefore propose to use a cloud-like storage working over a peer-to-peer network. We can, by following this approach, inherit from the autonomous, self-organising and structural properties of P2P networks. Because of the different nature of a distributed storage compared to the distributed naming, the overlay cannot be built according to the same parameters.

Minimising the traffic that has to traverse the network is achieved by storing the historical data the closest from where it is used [243, 244]. This observation leads to the notion of distributed storage spaces managing the storage of a particular location. We therefore build our overlay around the notion of data producers and storage peers locations to minimise the traffic.

Towards Location-Based Storage Spaces

While many distributed storage systems do not necessarily require a hierarchical architecture, we decided to organise the overlay by grouping locations, as shown in Figure 4.11. We call these contiguous locations inside the building hierarchy tree *storage spaces*, which are quite similar to the *naming areas* described in Section 4.1.2. A storage space groups contiguous locations of the building containing data producers (mainly sensors) and storage peers, which will store the data.

The idea behind this grouping is to follow the concept of *Content Delivery Networks* (CDN) aiming to provide the information where it is mostly used. This observation is also valid in smart buildings as historical data will be needed closely from where it is produced. In classical building automation, historical data are used by control loops to increase their accuracy by taking into account past observations. The output of those loops serves to achieve a certain state by driving actuators. It is obvious that the chain *sensors* → *storage* → *control loops* → *actuators* is something happening in a localised area. In the same way, data analysis algorithms such as machine learning training systems will also eventually be residing in the network instead of being server-side. In Section 4.4 we expose ways to deport this computing inside the sensor network.

The localised areas of interaction are principally represented by rooms holding the devices and the control strategies. We can, for example, mention the situation where a room controller device allows users to set their preferred temperature and controls the air conditioning according to the current temperature and the trend curve that is determined from historical data. Storage needs therefore to be organised according to rooms. As not each room potentially holds devices offering storage capacity, they need to be grouped in storage spaces that can supply rooms with sufficient capacity.

By following this approach, storage spaces can be formed around the building morphology in a hierarchical way. Each storage space manages independently where the data should be stored inside its authoritative locations. Storage spaces remain organised among themselves to react to changes in the storage capacity necessities. If a storage space gets out of capacity, it can ask a sibling to participate in the effort. Furthermore, relying on location-based storage spaces has the following advantages:

Traffic reduction and energy savings : Storage involves two phases: producers pushing the historical data into the storage, and control loops or other higher-level mashups retrieving it. By placing the storage where the data is produced and needed, the traffic is significantly reduced. Each data pushed to the storage by a producer will not have to traverse the network to reach its destination. More importantly, the retrieved historical data which can represent a large amount of information could cause network congestion if traversing the entire network. Restraining traffic in specific locations also contributes to lower the energy consumption of the network as less network equipments or nodes will have to forward data.

Increasing availability: Having access to the historical data is essential to ensure proper operation. Distributing the data where it is produced and required increases the availability. Indeed network interruptions or fails happening in other parts of the building will not have any consequence on the availability of the data within a storage space. This would not have been the case if the storage would have been located outside the sensor network either server-side, or even worst in the cloud.

Storage Actors

Each smart thing present in the building network can contribute in the storage system. To do so, devices embed an agent behaviour depending on their capacities. Using an agent approach comports many advantages compared to centralised systems. First, the loss of an agent will have no impact as the failover is automatically handled by another agent continuing the work. This results in an highly autonomous network where human intervention is not required. Agents take their own decisions and synchronise them with the other ones residing in the same environment. The following behaviours describe the responsibilities of agents. A smart thing is not limited to a single role but can assume both if required.

Storage coordinator (SC): When having this role, a smart thing is responsible to manage a storage space. It takes the decision which storage peer has to store the data of a resource while ensuring duplication. It internally holds a table containing the information of which storage peers are storing the tuples of a resource. The structure of the table looks as follows: `Resource-From-To-Peer`. With this table, it is able to process client queries to retrieve historical data stored on storage peers (SPs). In addition to this information, it also stores the original semantic descriptions of the resources that generate historical data. Moreover, it has the legitimacy to either merge a child storage space or to split its own one when necessary. To ensure availability and failover, a SC is always backed up by another one remaining passive as long as it has not detected a failure.

Storage peer (SP): The historical data is stored by smart things able to provide storage capacity. Any storage capacity is welcome so that even constrained devices offering only a few kilobytes can also participate. They offer an abstract interface hiding the internal storage mechanisms. Once a new data is available, they will store it in their internal database. Deleting outdated data is among their responsibilities.

These agents require no human configuration besides their location. Storage coordinators are elected among themselves during operation. Using a promotion mechanism between agents avoids the need for human intervention and makes the architecture only dependent to itself, as well as autonomous. In the same manner as for the naming agents, SCs start by being not authoritative for any storage space. If there is no other SC already managing the location of the storage space, then it will promote itself. Promoting itself as SC means that it is the first one in the building network and becomes automatically authoritative for any new location that can be logically determined as a child or parent of the location tree (using the full qualified names indicating the location). New SCs will only be present after the first partitioning. Otherwise, it will wait until one of those three actions arises to participate in the common effort: promoted as backup by a master, a backup becomes master when the master fails, or is designated after the creation of a new storage space because of partitioning.

A storage peer will after start up announce its presence to the SC managing the location it resides in. If there is no response, then this step will be repeated continually. Once registered with the SC, it awaits orders to store data generated by producers.

There are two other actors in the storage system that do not work as agents but rely on the interaction principle of the Web of Buildings. The cloud-like storage is built upon these actors that do not require any adaptation to benefit from the storage.

Client: Control loops, training algorithms or even users represent clients who want to have access to historical data. Before data of a resource is stored, a client must first announce its interest in that particular resource. Once this step done, it can further send simple queries to retrieve the data.

Producer: Historical data is issued by devices generating events, mostly sensors, but can be any kind of resource. They have no particular role other than exposing RESTful APIs with observable resources.

Storage Space Composition Strategies

In the same way as for naming, the number of storage spaces has an influence on the traffic and the availability of the system. The problem is however slightly different as the historical data will be carried by the network from the producer to the SPs. The distance between the producer and SPs plays a role in the overall traffic.

A large number of zones increases availability and lessens the impact of data transportation, i.e. producers and SPs will be close to each other. It will however generate tremendous traffic for synchronisation between the spaces. On the other hand, a small number of storage spaces induces few synchronisation traffic, but increases the impact of data forwarding through the network besides augmenting the risk of being affected by a network failure. The optimal solution minimising the transmission depends therefore on several parameters including the distance between producers and SPs, and the costs of synchronisation between the storage spaces.

As previously mentioned, our cloud storage is split into several spaces forming a tree. Each storage space is authoritative for one or many contiguous locations of the building. The location grouping process can be seen under several perspectives, and many algorithms can be imagined for defining spaces. However, as our aim is also to build an energy-efficient solution, we opted for an approach limiting the traffic between all the agents. SCs face two types of decisions during the partitioning process: should they split their own space into two new ones, or should they regroup a child space with their current one. Decisions are taken according to notions of costs. This partitioning process is triggered only at special configurable occasions to limit the energy impact of the computing:

- The storage capacity of child spaces is below or above a certain threshold.
- The number of stored resources varies and exceeds a predefined threshold.
- The storage space becomes authoritative for new locations.

Splitting The coordinator of a storage space S will first compute the cost $C(S_1, S_2)$ for each possible decomposition into two new spaces S_1 and S_2 according to Equation (4.1). We limit this computation to storage spaces having at least two SCs and two SPs, and having obviously enough space to store the historical data of resources residing within the actual space. Finally, the minimum cost value among the possible decomposition is compared to the individual cost $CI(S)$ of the current storage space S. The storage space will be split if the evaluation of Equation (4.2) is true, i.e. if the best decomposition (minimising the cost) brings a gain of efficiency compared to the current situation.

$$C(S_1, S_2) = CI(S_1) + CI(S_2) + CB(S_1, S_2) \tag{4.1}$$
$$CI(S) - \min(C(S_1, S_2)) > \epsilon_s \tag{4.2}$$

CI is the *internal* cost equals to the total number of hops (locations) between producers and storage peers within the storage space. We consider that forwarding data to distant storage peers is penalising.

CB is the cost *between* storage spaces depending on the filling rate of the child space. Children storage spaces are regularly sending some status data to their parent space, generating traffic. This traffic essentially depends on the filling rate of the child space, as more messages will be exchanged as capacity decreases.

ϵ_s A cost gain factor above which the splitting is performed. This value can be for example determined empirically to meet some requirements or be fixed to a default value for a standard behaviour.

Merging A SC can decide to merge its own storage space with a child one in two situations:

Space: When one of the storage spaces becomes full and has no more storage capacity. Grouping the spaces will allow to increase the storage capacity and to postpone storage saturation.

Efficiency: When the cost of a single storage space would be less than the actual situation. However, a SC has no knowledge about the internal structure (locations, storage peers and producers) of a child space. This results in the incapacity to compute the costs for the merged storage spaces. We opted for a solution where we compute an estimation of the cost $CM(S)$ for the *merged* space depending on each spaces costs and internal location tree depth, as described in Equation (4.3). The merging is performed if the evaluation of Equation (4.4) is true, i.e. if the merging brings a gain.

$$CM(S) = (CI(S_1) + CI(S_2)) * \ln(D(S_1) + D(S_2)) \tag{4.3}$$
$$C(S_1, S_2) - CM(S) > \epsilon_m \tag{4.4}$$

$C(S1, S2)$ Two-space cost computed as above.

D Depth of the space's internal location tree. Each storage space is composed of one or more contiguous locations represented as a tree. The depth of the tree influences the probability of having distant producers and storage peers. The logarithm function is here used to fit with the notion of tree depth.

ϵ_m A cost gain factor above which the merging is performed.

If we compare our approach to existing solutions such as P2P storage or the increasingly used Hadoop Distributed File System (HDFS), we can see that their distribution strategies do not include the notions of closeness and areas. P2P storage systems typically distribute pieces of data randomly over nodes that can be located anywhere [245, 246]. On the other hand, HDFS works with a predefined cluster of machines [340]. Their data distribution mechanism is however quite similar to ours as what they call a *NameNode* hosts the file system index. *DataNodes* store the data by ensuring that it is by default replicated on three different locations. Their architecture is however heavy and static, which is not suited for dynamic environments, besides requiring dedicated hardware.

Multicast-Based Data Storage

In order to ensure availability, historical data need at least to be duplicated on storage peers. This need for duplication is caused by the dynamic of smart things acting as storage peers that can potentially suddenly disappear. While being not an optimal replication solution (we do not claim to be compliant with isolation levels of databases), we propose to reuse the multicast capability of CoAP to distribute historical data simultaneously on several storage peers.

The SP could earn the role of retrieving the latest data from the sensors, following a polling strategy. Such an approach is obviously not feasible for two reasons. First, using polling would not allow to guarantee that all values have been read and stored. Indeed resources can change irregularly and very quickly. Then, the read requests would generate a large amount of unnecessary traffic. We therefore advocate that producers should notify SPs when a new value has to be stored, reusing notification techniques.

On the other hand, multicast communications allow to address several peers at the same time. Instead of sending values one at a time to the SPs, a single multicast message is sufficient to transfer the value to the SPs. Being in most situations an advantage, multicast impersonates the communication as the sender does not know which are the receivers, increasing autonomy of the system. However, it avoids the sender to ensure that all receivers (in our case the storage peers) have received the new value. This can cause non-identical replications in the case a SP does not receive the multicast message.

This last point is actually the reason why CoAP observation and group communication are mutually exclusive. It is indeed not possible to benefit from reliability, ensuring that a message has been delivered at its destination when using multicast. In our distributed storage, we reuse our proposition of reliable CoAP group communication as outlined in Section 5.2.2.

In our architecture, resources must be observable in order to participate in the cloud storage as they will notify peers. The SC is responsible to generate a token that will be used to map the notification with a resource. As shown in Figure 4.27, it then activates the storage one by one on the peers by sharing the token. From that moment, SPs are listening for notifications featuring the token, and will store each new value. Then, it enables observation of the resource with this same token and indicates which peers are storing the values, so that the producer can determine if they have all received the notification according to the acknowledgements. Finally, new values are sent over multicast by the producer, reducing the number of messages.

The RESTful Storage Protocol

We present the relative application protocol following the guidelines and concepts of the in-network cloud storage. Each interaction happens over RESTful services hiding the nature of the device. Additionally, relying on RESTful APIs homogenises the application level as every application exploits the already ubiquitous REST server.

In the same manner as for our distributed naming system, the API shown in Figure 4.28 is decomposed according to the purpose and scope of the services in three distinct interfaces:

Management interface is reachable over a predefined multicast group address that must be known by clients and agents. Each master storage coordinator listens to this interface. It

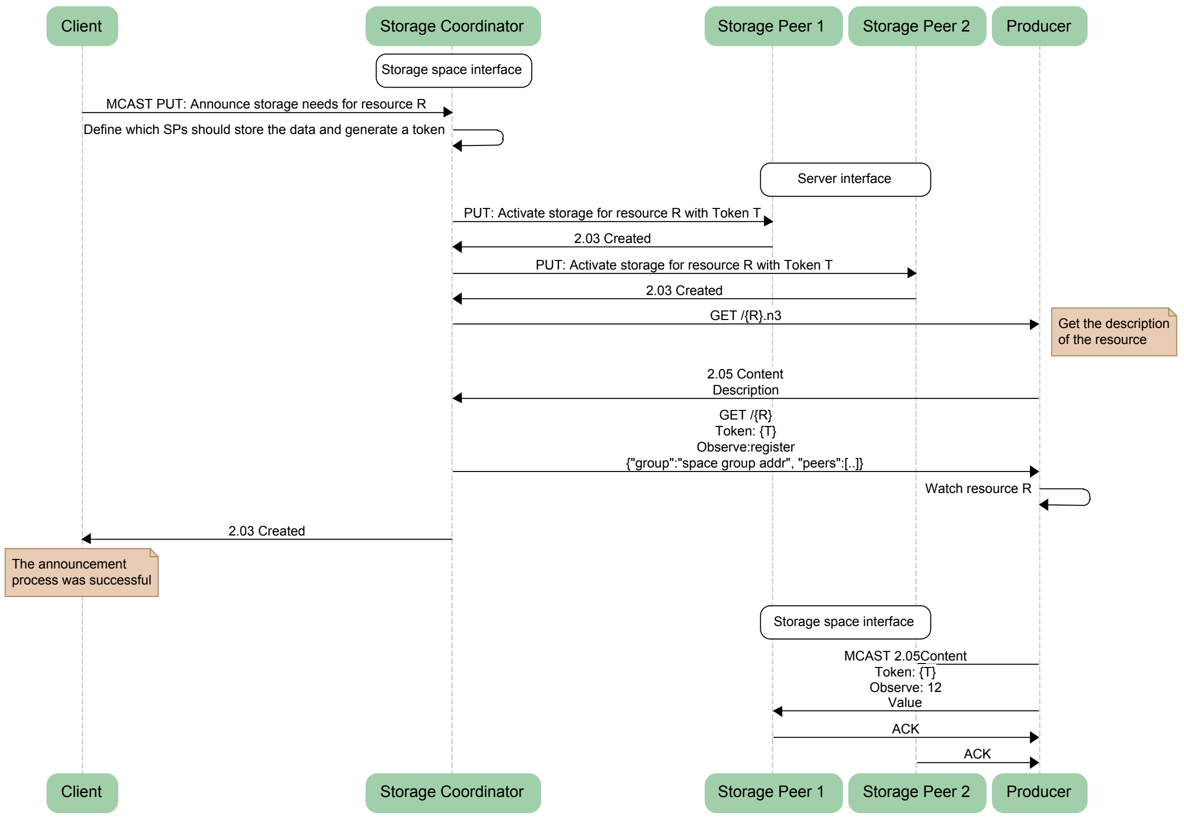

Client
Storage Coordinator
Storage Peer 1
Storage Peer 2
Producer
Storage space interface
MCAST PUT: Announce storage needs for resource R
Define which SPs should store the data and generate a token
Server interface
PUT: Activate storage for resource R with Token T
2.03 Created
PUT: Activate storage for resource R with Token T
2.03 Created
GET /{R}.n3
Get the description of the resource
2.05 Content Description
GET /{R} Token: {T} Observe:register {"group":"space group addr", "peers":[..]}
Watch resource R
2.03 Created
The announcement process was successful
Storage space interface
MCAST 2.05Content Token: {T} Observe: 12 Value
ACK
ACK

Figure 4.27: The sequence of exchanged messages when a client announces data storage. Producers send notifications over multicast to address simultaneously several storage peers.

is intended for clients and agents discovering which storage space manages the location of interest. They can therefore retrieve the group address of the storage space having the location under its authority.

Storage space interface is associated with a random generated multicast address different for each storage space. This allows to restrict communications that are only useful for a specific storage space. The most important services for clients are those which allow to announce storage needs for a resource, and to retrieve the historical data. Another set of services is dedicated to the announcement of agents. The last services concern the management of inter-space communications for splitting or merging.

Server interface uses the unicast addresses of agents to communicate among them. It is mostly dedicated for agent promotion and for the storage coordinator managing the peers (activation and retrieval).

The first step for a client is to announce its intention to store data for a particular resource and for a given amount of time. This is realised by sending a multicast packet containing the necessary information to the SC. In order to offer a lightweight and robust mechanism for the announcement, we decided to specify the interchange format with a JSON schema, shown in Listing 4.8. JSON schemas [341] are the equivalent to XML schemas and are used to validate the data interchanged with Web services. As we are working in a world of constrained devices, using JSON appeared a reasonable choice. The SC performs a validation to ensure that the received data is correct.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Storage",
```

Method	Resource	Purpose
Management interface: predefined multicast address		
GET	/storagespace/{location}	Discover which storage space manages this location (full qualified name)
Storage space interface: storage space multicast address		
GET	/storage/{resource}	Retrieve the historical data for a resource (fully qualified name and resource)
PUT	/storage/{resource}	Announce storage need for a resource (fully qualified name and resource)
PUT	/coordinator/{id}	Register a storage coordinator agent
DELETE	/coordinator/{id}	Unregister a storage coordinator agent
PUT	/peer/{id}	Register a storage peer agent
DELETE	/peer/{id}	Unregister a storage peer agent
PUT	/space/{id}/capacity	Child storage space notifies about its storage capacity
PUT	/space/{id}/group	Parent storage space informs about its new multicast zone group
DELETE	/space/{id}	Parent storage space wants to merge this space with its own one
Server interface: unicast server address		
PUT	/space/{id}/{role}	Designate the server as coordinator master or slave
DELETE	/space/{id}/{role}	Inform a slave that it is no more a coordinator
PUT	/space/{id}	Perform a full replication of the repartition table and the descriptions
PUT	/storage/{resource}	Activate storage on a storage peer for a resource (fully qualified name and resource)
DELETE	/storage/{resource}	Deactivate storage on a storage peer for a resource (fully qualified name and resource)
GET	/storage/{resource}	Retrieve historical data for a resource stored on a peer (fully qualified name and resource)

Figure 4.28: RESTful API protocol of the in-network cloud storage. The functionalities are split in three distinct interfaces according to their scope and purpose.

```
    "description": "A resource storage announcement",
    "type": "object",
    "properties":{
      "max-age": {"description": "How long the data should be stored
          ", "type": "integer"},
      "unit": {"description": "Unit for max_age", "enum": ["day", "
          month", "year"]}
    },
    "required": ["url", "max-age", "unit"]
}
```

Listing 4.8: JSON Schema for data storage announcement.

In the case of a producer disappearing, its historical data will remain accessible as long as it has not reached the maximum age. In order to have a trace of what the resource was representing, the SC stores its semantic description and makes it accessible to the rest of the network. This allows control loops or other applications to use and understand the meaning of the historical data even if the producer has ceased to generate events.

Once the announcement performed, clients have the possibility to retrieve the stored data, as depicted in Figure 4.29. This is achieved by the client sending a `GET` request to the SC responsible for the resource. The request must hold two parameters indicating the interval's start and end dates as follows `...?start={start_date}&end={end_date}`. This enables the client to filter the history by dates. The SC verifies that the requested resource is within its storage space. If it is the case, it will then compute on which peers the data has to be retrieved by using its repartition table, so that the communications are minimised. Indeed, as each tuple of data is stored at least twice with overlapping intervals, it is important to avoid unnecessary exchanges. The results are then merged together in order to form a sorted JSON array of historical data.

Due to the limited computational power of certain constrained devices, applying compression algorithms such as GZIP is not possible when exchanging historical data between SPs and SCs. However, JSON could be transposed in binary format to optimise the communications.

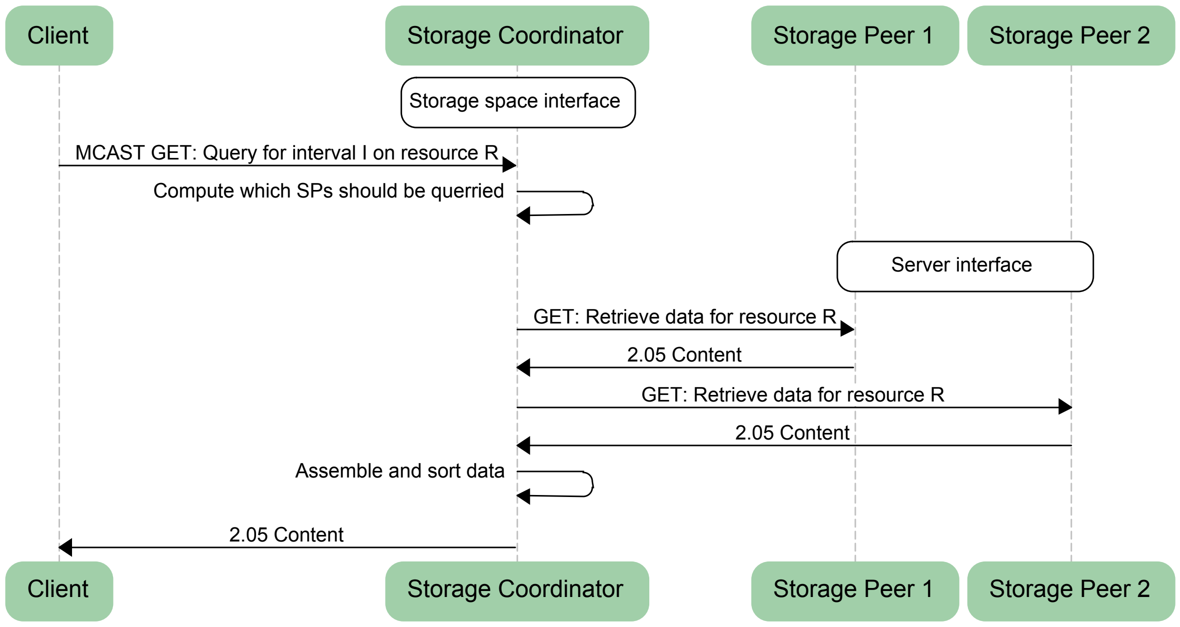


Figure 4.29: The sequence of exchanged messages for a client querying historical data with the interval of time I on resource R. The storage coordinator computes on which storage peers data should be retrieved to optimise the network traffic. It then assembles and returns it to the client.

The use of RESTful APIs on the peers storing time-series is an interesting alternative to the common SQL. Indeed smart things being mostly composed of constrained devices do not have the ability to implement SQL. REST allows to standardise the way historical data are exposed in sensor networks, this in a very simple manner, and probably sufficient for most scenarios.

Evaluation

We already evaluated the concept of splitting the building into independent zones from a network traffic point of view for our distributed naming system in Section 4.1.2. We here want to assess the performance for data storage. An important aspect of data storage is the time required for performing queries.

In order to evaluate the performance of our system, and especially the retrieval of data, we set an experimental storage space. Our test space was composed of one SC (Raspberry Pi) and three SPs (one Raspberry Pi and two OpenPicus Flyport). Each SP was preloaded with storage data as follows: 15072 entries for the Raspberry Pi, 14499 entries for the Flyport Wi-Fi and 14072 entries for the Flyport Ethernet. The SC was for its part preloaded with the data repartition on the peers.

The same dataset was stored into two commercial cloud solutions, allowing us to compare the performance of our architecture to common used databases, considered as references. We selected for this purpose the Amazon RDS with MySQL [342], and the MongoDB no SQL database [343]. Both were hosted in the cloud by their provider and were offered as a service.

Distributed vs. Centralised To determine the consequence of using a SC instead of directly talking to a REST data source, we set a second test bed only composed of a SP (Raspberry Pi). This peer was preloaded with the entire data set that was distributed among the peers in the previous configuration (without duplicate entries). The same retrieval request was sent for each test. It queries for historical data of a specific resource covering an interval of 12 hours. The answer for this request returns 39 entries, each one containing the timestamp of the measure and the associated value.

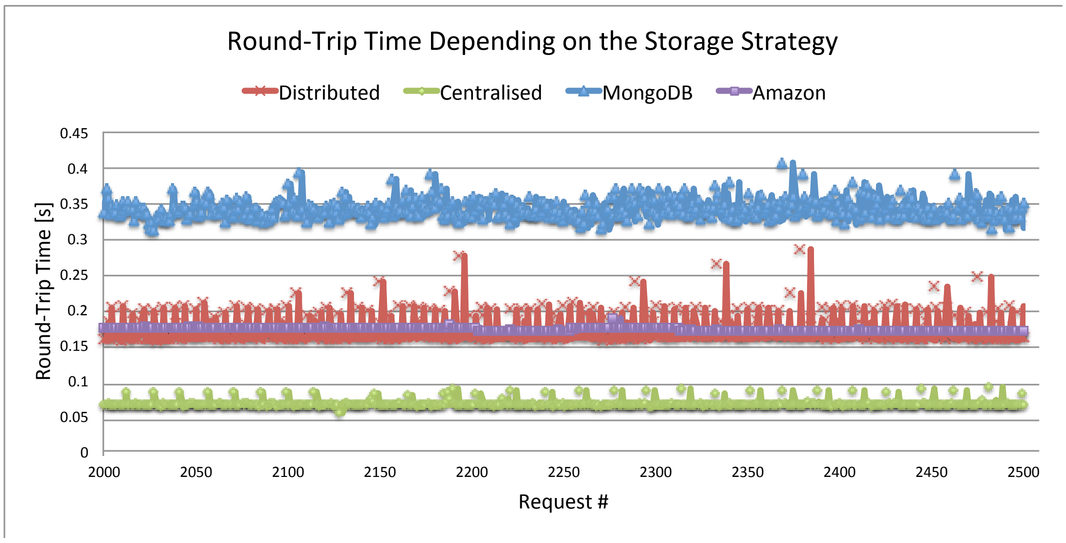


Figure 4.30: Round-trip time for 500 consecutive `GET` requests for retrieving historical data directly on a centralised sink, with our cloud-like storage or with the MongoDB and Amazon cloud solutions.

Figure 4.30 shows a typical measured round-trip time for 500 consecutive requests. One can see that obviously the centralised approach performs better than the distributed one. For the first case, the client bypasses the SC and performs the requests directly on a storage peer storing the whole dataset. This results in having a short average round-trip time of about 70 milliseconds. The cumulative density function (CDF) of the round-trip time is shown in Figure 4.31. One can see that 90% of the queries are performed within 71 milliseconds.

We would like to emphasise on the fact that the network conditions can have an influence on the achievable performance for the cloud solutions (MongoDB and Amazon). The college of engineering of Fribourg (HEIA-FR) from where were performed the tests has access to Internet through the *SWITCH* backbone [344]. All Swiss universities are connected to this backbone offering high data rates and prioritised access to services. In the same way, the traffic load within the building (Ethernet, Wi-Fi, 6LoWPAN, etc.) can have an impact on the performance of our in-network storage.

For the distributed approach, the query process is more complicated as each request is sent to the SC that will collect the distributed data on the SPs. In this case, the average round-trip time for 2500 consecutive requests is 179 milliseconds. The CDF shown in Figure 4.31 outlines that 90% of the queries were answered within 200 milliseconds.

When comparing the results of our approach to the commercial cloud solutions, we observe that MongoDB performs the worst with an average round-trip time of 342 milliseconds. The MySQL storage of Amazon has comparable results to our distributed storage with an average round-trip time of 175 milliseconds.

Not surprisingly the performance for direct communication is better. We can deduce that the time used by the coordinator for retrieving the distributed data and building the response is

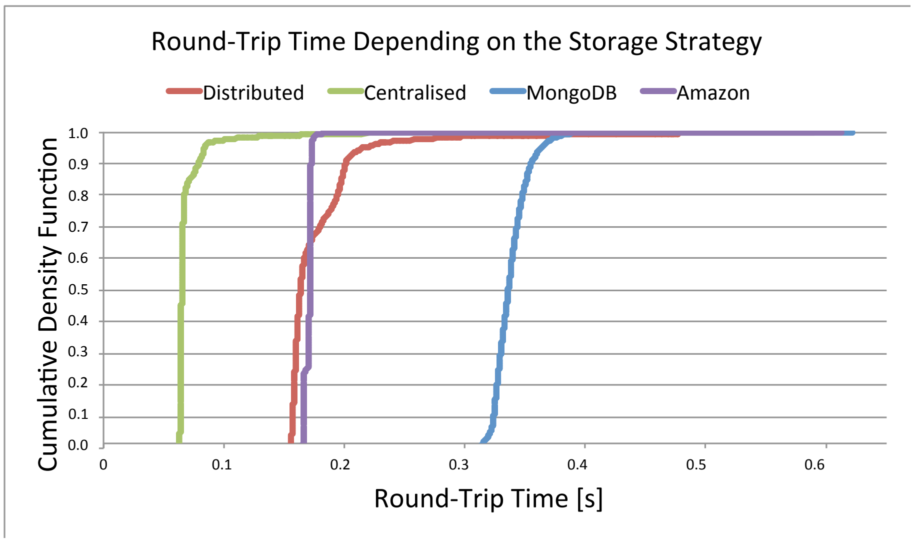


Figure 4.31: Round-trip time cumulative density function for the direct access, with our cloud-like storage or with the MongoDB and Amazon cloud solutions.

about 110 milliseconds. However, we believe that this additional delay is compatible with most application scenarios.

Evaluating Concurrency In this evaluation, we want to assess the difference between the approaches in terms of concurrent requests. Similarly to the previous case, we performed requests using the distributed approach and directly to the SP. Figure 4.32 shows the round-trip time when having up to 100 concurrent clients.

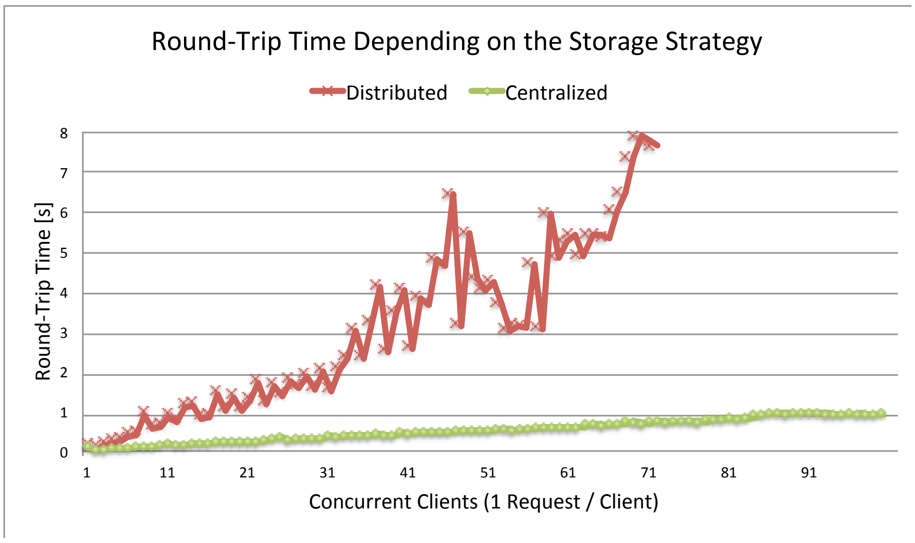


Figure 4.32: Round-trip time for up to 100 concurrent clients sending GET requests for retrieving historical data directly on a centralised sink or with the cloud-like storage.

Not astonishingly, the centralised approach scales much better. The round-trip time of the distributed approach increases rapidly with the number of concurrent clients. With 10 concurrent clients queries already take about one second with distributed data, whereas 100 concurrent clients are required in the centralised approach to reach this round-trip time.

The cloud storage solutions are much more scalable in terms of concurrency. Indeed the round-trip time does not significantly vary even with 100 concurrent clients. All the queries are responded, so that the success rate is for both commercial solutions of 100% even with 100 concurrent clients.

From Figure 4.33 one can see that the success rate of the cloud-like storage drops as soon as the number of concurrent clients is reaching 70. On the other hand, the centralised server is still able to guarantee 100% success rate with 100 concurrent clients.

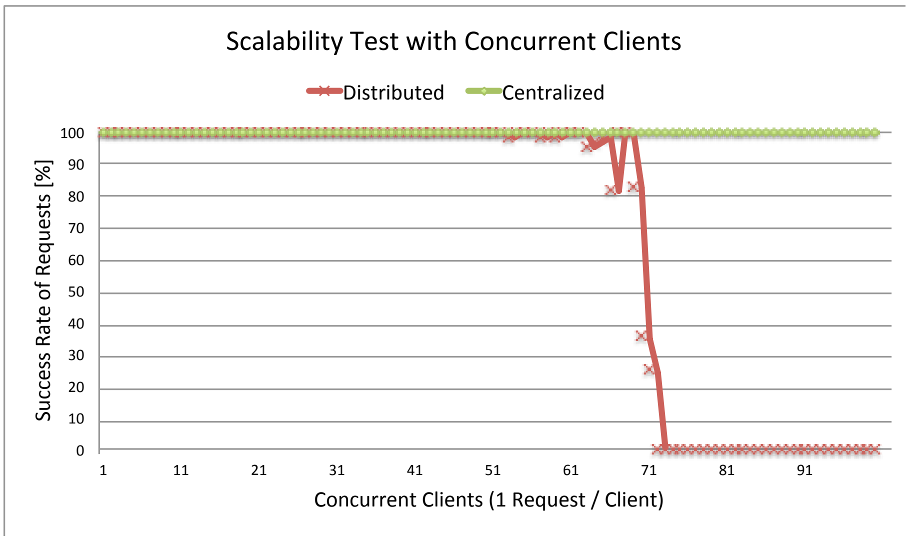


Figure 4.33: Success rate of requests as a function of the number of concurrent requests for the centralised and the distributed storage.

These limitations can be explained by the number of sub-requests that are sent by the SC to the peers. Indeed each client request will result in up to three peer requests. It is very likely that the SC, having limited memory and CPU, is not able to handle such an amount of requests. Furthermore, the OpenPicus Flyports do not provide any threading and can only accept one request at a time, which is the main bottleneck. Other node technologies could perform better for such situations.

Although the distributed approach performs worse than having a centralised sink, we must consider that only a limited number of simultaneous requests will be executed in a storage space. Indeed a storage space will mostly manage a limited number of locations, each one encompassing a couple of control loops. In addition, the probability of simultaneous requests for historical data at the same time is low. Finally, even if the centralised approach performs better, it is unlikely to find in the building network smart things having enough capacity to centralise all the data.

4.2.4 Related Work

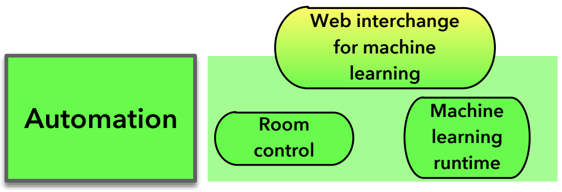
RESTful APIs along with HTTP have been successfully used in projects to interact with smart things such as sensors [136, 36, 137]. Because of the heavy memory requirements of HTTP [184], CoAP has been more and more preferred in sensor networks [73, 148, 247, 132]. Moreover, the Web of Things architecture [48] provides a set of guidelines how smart things should expose their functionalities as RESTful APIs.

Besides sharing the same application protocol, devices need to be compatible in terms of data they exchange. The Open Building Information Xchange [316] (oBIX) format describes in XML properties of sensors. It can be combined with SOAP and replaces WSDL to advertise services specifically dedicated to building automation devices. oBix serves as base format in [131, 132] to predefine a set of contracts specifying interactions with devices along with the data format specification. These contracts and their related properties are modelling KNX datapoints. In our proposition, we take a more abstract approach offering much more flexibility as datapoints can be dynamically described and are not predefined.

To interconnect devices working over different protocols, two solutions have been explored: multi-protocol devices and smart gateways. For the former, devices embed several application stacks enabling interactions with various technologies. In [29], prototype devices running KNX and BACnet were developed. As not all functionalities can be translated, smart gateways appear as a better solution. Several implementations allowed to map functionalities of devices that cannot run a Web server to RESTful APIs [123, 129, 131].

With all the devices producing data, it becomes important to provide a storage infrastructure. While external cloud services offer simple interfaces to store and to access data, they generate tremendous traffic in comparison with in-network storage [244]. The idea of storing historical data directly on sensor devices is not a new topic. The Capsule project proposed in [242] is a rich, flexible and portable object storage abstraction offering stream, file, array, queue and index storage objects for data storage and retrieval. Moreover, fully distributed architectures, where constrained devices deployed in sensor networks are considered as local storages, were studied in [248, 249]. Some research have focused on where data sinks should be placed in a sensor network in order to minimise communication costs [250, 243].

4.3 Automation Level



In the Automation Level, we aim at offering tools for implementing control strategies that either target an increase of comfort or an optimisation of energy consumption. With smart things accessible over RESTful APIs, applications can be easily built on top of sensing or actuation capabilities.

Standard control loops have already been studied extensively by the scientific community [84, 251, 22]. We put here a particular attention on novel data-driven techniques gaining momentum in smart building applications. Machine learning distinguishes from traditional control by being a more complex process involving several steps. In the Automation Level, we propose to integrate a machine learning runtime as core of the WoB by distributing it within the building network. For this purpose, we reuse concepts and technologies of the Web to model this process so that machine learning becomes a first-class citizen of the Web. Considering runtime algorithms as Web resources enables to integrate them in mashups and therefore augments the potential achievable optimisations. In this direction, we show how end-users can easily create mashups by themselves with no specific knowledge.

In this section we first provide a short overview on the most used machine learning approaches. We then introduce the Web-based MaLeX format dedicated to the exchange of machine learning mathematical models. This format is used in the runtime infrastructure where smart things are able to execute models in order to create higher-level sensors. We then come with a proposition of a mashup application allowing end-users to easily create control scenarios. Along with the distributed architecture for executing mashups, we dematerialise classical room controllers while delivering more opportunities.

4.3.1 A Machine Learning Approach

Traditional control loops are typically relying on simple comparisons against thresholds. Machine learning in contrast relies on historical data to build more complex prediction or classification. In the context of smart buildings, machine learning can also benefit from a large amount of sensors able to gather higher-lever information. Activity detection [89], appliance recognition [88] and thermal inertia [252] among others are fields that can benefit from machine learning to improve the management of a building.

In machine learning, once models have been trained by relying on historical data (more details on this step are provided in Section 4.4), they can be used for classification or prediction in a running system. This last part is called the decision or testing process. As it is this process that takes the live data (from sensors in our context) to perform the classification, we refer to it as the machine learning runtime.

If we concentrate on classification tasks, two approaches can be used, assuming a probabilistic bayesian framework:

Generative models: The inference problem consists in building a mathematical model for the likelihood $p(x \mid M_k)$ for each class model M_k. The decision part is performed using the Bayes theorem to compute the posterior model probabilities with:

$$p(M_k \mid x) = \frac{p(x \mid M_k)p(M_k)}{\sum_k p(x \mid M_k)p(M_k)} \tag{4.5}$$

where $p(M_k)$ are the class prior probabilities. The class membership for the new observation x is obtained by comparing the posterior probabilities and selecting the one with the highest value. The term *generative* comes from the fact that new observations can potentially be generated by the models. Naive Bayes classifiers, Gaussian Mixture Models (GMMs) and Hidden Markov Models (HMMs) are examples of models belonging to this category.

Discriminative models: The inference consists here to compute a mathematical model of the posterior probability $P(M_k \mid x)$. As for the *generative* models, the class membership for the new observation x is obtained by comparing the posterior probabilities. Support Vector Machines (SVMs), Artificial Neural Networks (ANNs) and Multinomial logistic regression are examples of algorithms belonging to this category. The term discriminative is related to the fact that the models are generally computing a decision border between classes instead of the class conditional probability density functions.

Both approaches have different strengths from a classification point of view. Discriminant models are usually preferred, leading to good tradeoffs between performance and size of training data. In fact, discriminative modelling can be seen as an easier task as we try to directly solve a problem (find the posterior probabilities) without solving a more general problem as in the generative modelling (find the likelihoods). Evidences show that discriminative modelling leads to a lower asymptotic error, however generative modelling approaches this error faster than the discriminative [253]. Finally the generative approach are reported to deal better with missing input values, outliers and unlabelled data [254].

As shown in Figure 4.34, the classification output can be further processed in mashups or other

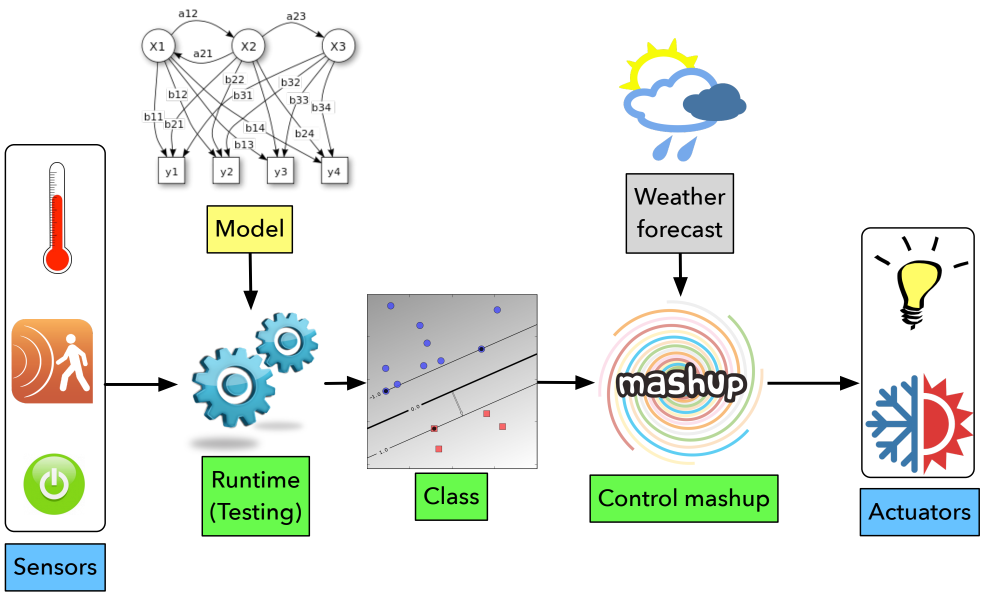


Figure 4.34: Mashups and control applications use the runtime output to drive actuators in order to achieve a certain state within the building.

automation applications. These mashups can use several informations to select the actions that have to be taken to achieve the optimal state in the building. For example, external services such as weather forecasts allow to anticipate temperature changes and to adapt the HVAC thresholds.

4.3.2 MaLeX: A Web Interchange Format

As the learning and runtime steps can be separated processes, models need to be exchanged between them. The learning process generates models that will be used by the runtime along with live data. For this reason, models need to be expressed with a shared format understandable within the Web of Buildings.

Exchanging information between different systems is always a tricky task. One has to agree about formalisms of data representation and structure. This is especially true for machine learning applications having to exchange mathematical models. The Data Mining Group attempted to solve this problematic by proposing the Predictive Model Markup Language (PMML) [345]. This XML schema semantically describes how to represent in XML several widespread predictive models. However, PMML comes with several drawbacks limiting its use for the WoB. First, it does not describe widespread generative models such as GMMs or HMMs. Furthermore, the PMML structure is rather complex and induces a high verbosity, which is less suitable for smart things. We therefore developed our own data model, the *Machine Learning Exchange Format* (MaLeX) to describe generative models, including HMMs and GMMs.

Regarding the exchange format, we opted for JSON that is widespread in Web applications and commonly used with mashups. In order to ensure that all participants of the WoB are able to understand the exchanged model, we defined a JSON schema [341] describing general properties and mathematical models entities[1].

The schema aims mainly at describing generative models such as HMMs and GMMs but is not limited to those. In the case of other types of models being required, the schema can be

[1]Publicly available at http://www.wattict.com/web/download/ml-model.json

supplemented by including additional descriptions. For this purpose, the general structure of the schema defines three sub-components:

Global properties provide information about the purpose of the runtime such as appliance or activity recognition. Additionally, optional locations can give insights about which parts of the building are concerned by the model.

Dimensions represent in the machine learning jargon the inputs that provide the live data. In our context, live data will be provided by sensors identified through their unique URI. The dimensions are categorised in three types: sensor, weather service, or agenda. Depending on the application, other types of dimensions could be introduced. Indicating the type of dimensions allows processes to understand how to handle the returned data.

Classes is the most complex part of the schema and depends on the type of model that has to be described. Each model type must be defined in this section with its specific properties and data formats characterising the classes. For example, the HMM description encompasses information about the different matrices along with the required normalisation and the internal states.

4.3.3 Distributing the Machine Learning Runtime

Typical machine learning (ML) implementations are pushing the data acquired by Internet of Things devices to server-side computers or cloud services in charge of performing data storage and ML computation. While this is a satisfactory configuration for some scenarios, it is not conceivable for many applications where privacy is a concern.

The main concern is the traffic generated by the transfer of information through a border router, which can be overloaded and also represents a single point of failure. Having the sensor network connected to the Internet opens a door for possible attacks.

With respect to the aforementioned problematic, deporting the ML runtime into the sensor network appears as a meaningful strategy. We also observe that smart things are showing increasing capacities in terms of storage and CPU, underlining the interest of such strategy.

We therefore propose in this section an architecture to distribute machine learning algorithms in a sensor network, as it was presented in [8, 9]. The computational power of smart things is exploited, forming a cloud space for ML algorithms.

Requirements

In contrast with the machine learning training process, the runtime does not require a substantial amount of CPU and memory to perform classification. Algorithms at testing time are less computationally expensive than at their training time and are candidates to be run on constrained devices. Smart things usually perform few computations, which merely consists of reading physical conditions or actuating.

The available computational power of smart things can be reused to participate in an overall effort, which consists to provide higher-level information. The smart things installed in the building network share their available computational power to form a cloud-like space executing the ML runtime. Offering an architecture that is easy to use, self-organising and requiring no expert knowledge within the building network can foster deployment of novel ML applications. Following this vision, we intend to build our solution in a SaaS manner like the Google Prediction API [346] and BigML [347] cloud services.

Additionally to the aforementioned properties and behaviour, our distributed runtime solution strives to fulfil following requirements:

Integration with data analysis software: The models are generated by training algorithms usually running on data analysis software such as Matlab, Octave or Python. In order to make our architecture accessible for developers, they should be able to distribute their models directly from the data analysis software.

Machine learning abstraction: The machine learning process is often managed by data specialists who define and optimise algorithms fusioning data to gather high-level knowledge. This has as consequence to restrict the use of tML only to initiated developers. We therefore want to lower this barrier so that any developer can use ML, in a similar way as for interacting with sensors and actuators.

Low network traffic footprint: Deploying mathematical models can result in high traffic due to the high quantity of data to exchange. We aim at reducing this footprint so that it does not represent a threat for the proper working of the building network.

Web-oriented application layer: To be integrated in our WoB architecture, components of the distributed machine learning runtime have to follow Web paradigms. RESTful APIs homogenise the application layer so that ML classes and properties are accessible over standardised interfaces.

As previously mentioned in this thesis, the closeness of data producers is relevant to minimise the network traffic footprint. We will therefore follow this concept by deploying the runtime near the live data. This will limit the propagation of data and lower the network traffic.

Runtime Models

As mentioned before, classification can be performed according to two different approaches, namely generative and discriminative. Although targeting the same purpose, they differ in the way the classification is performed. This has an impact on the achievable distribution as both approaches offer different possibilities for distributed computation.

In our case we are interested in the possibility to distribute the computation load of the decision process on several devices. With generative models, this is naturally done considering that each model contribution M_k can be computed individually. In other words, the likelihood computation of a given model is independent to the others and the computation of class conditional probabilities $p(x \mid M_k)$ (likelihoods) can be distributed in different nodes. This distribution is usually not possible or more difficult for discriminative models such as ANNs or SVMs where class parameters are shared in the model and where class decisions are computed in one pass.

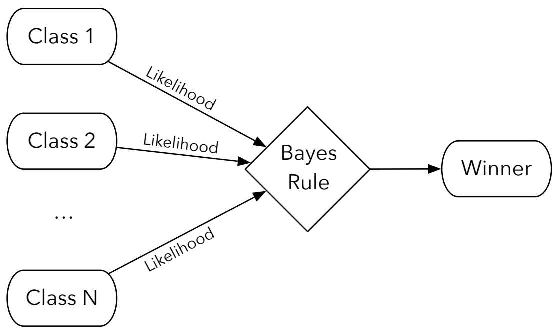


Figure 4.35: The likelihood values are compared according to the Bayes rule to designate the winner.

From this last observation, we can state that generative models appear an interesting solution as classes can be individually distributed with no dependency to each other. Each class holds its own set of mathematical models required to compute the likelihood, which are mainly HMMs and GMMs. As shown in Figure 4.35, the likelihood values can then be compared and, according to the Bayes rule, the class having the highest likelihood is designated as the winner.

Runtime Agents

We propose an architecture for ML algorithms in which classification tasks are delegated to constrained things, accessing their capabilities using RESTful APIs either with HTTP or CoAP. The REST architectural style is extended to non-physical resources like model parameters to offer a standard and flexible manipulation. Our architecture is based on a set of agents having the capability to execute pre-deployed specific ML algorithms by receiving as input the model parameters (pre-trained). We opted for such an approach rather than dynamically deploying code due to the memory restrictions of smart things, where an application container can usually not be executed.

In order to comply with a ML process for classification tasks, we introduce two novel entities, namely the *Virtual Sensor* and the *Virtual Class*. They are intended to hide the complexity of the decision process by exposing simple RESTful APIs. These interfaces contribute to the abstraction of the runtime into a set of comprehensible and reusable components. The general architecture and the interactions between these entities are shown in Figure 4.36.

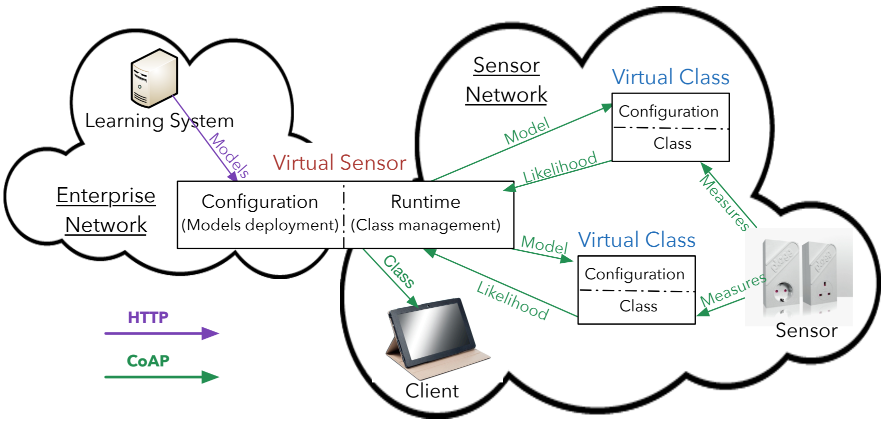


Figure 4.36: Architectural concept of *Virtual Sensor* and *Virtual Classes* with the information exchanged during runtime.

In our architecture, autonomy and scalability are achieved by relying on an agent behaviour. Smart things endowed with sufficient computational power are elected to run agents that are instances of Virtual Sensor and Virtual Class. Each agent exposes a semantic description giving insights about its capabilities and location within the building. These descriptions will be further used during the deployment process to select appropriate agents.

Virtual Sensor A Virtual Sensor instance is able to compute a classification decision, aggregating likelihoods computed by the Virtual Classes. In other words, a Virtual Sensor instance merges other sensors measurements in order to extract knowledge, as for example a class. In our architecture, a Virtual Sensor instance stands for a non-physical sensor exploiting generative models. A Virtual Sensor can then be considered as a "higher-level" sensor emitting a value related to a class decision in the sense of Bayes' law. Within the sensor network, they appear as regular sensors performing measurements. Hiding the complexity of the data fusion by acting as a conventional sensor increases reusability and eases the integration within already existing sensor networks. We also propose to decouple the configuration API from the runtime API in order to hide the ML part.

The configuration API is HTTP-based in order to be accessible by high-level applications like training systems residing on the enterprise network. New Virtual Sensors can be created by PUTting for example to `http://virtual-sensor.kitchen.home/virtualsensors/{id}`, which would be an available agent. The client has to provide in the payload the models parameters using our MaLeX format. The role of the Virtual Sensor is then to distribute the models further on available smart things. It will therefore query for agents able to execute the provided models, becoming Virtual Classes.

The runtime API represents the output of the decision making task. It will apply the Bayes rule to the likelihoods provided by Virtual Classes. Each Virtual Sensor instance possesses its own runtime resource accessible over CoAP by issuing a `GET` request to an URI such as `coap://virtual-sensor.kitchen.home/{virtual_sensor_name}`. For example, a Virtual Sensor dedicated to activity detection would expose an `/activity` resource returning the winning class that represents an activity. We opted for CoAP as application protocol instead of HTTP as it is conceived for sensor networks and more suited for constrained devices.

Virtual Class As previously mentioned, a ML classification task relies on the computation of several class likelihoods. These classes hold a mathematical model such as a GMM containing parameters of mean and covariance matrices. To simplify the interaction with those complex mathematical models, we apply the same principle of high abstraction level by encapsulating them in Virtual Classes. Similarly to the Virtual Sensor, we make them acting as regular sensors by decomposing their interface in two distinct APIs accessible with CoAP.

Virtual Class deployment is realised by a Virtual Sensor through several resources located on the configuration API. The API structure depends on the type of algorithm the agent is able to execute. There will be a resource for each model parameter type, allowing the replacement of specific parameters instead of the whole model. The first step is always the creation of the Virtual Class by sending a `PUT` request such as `http://virtual-class.kitchen.home/virtualclasses/{id}` with the general model properties contained in the payload. The `/run` sub-resource can be POSTed to launch the execution engine of the model. During this phase, the Virtual Class will register as observer on the sensors used as input dimensions of the model to avoid polling. Additionally, it will schedule the execution of the algorithm according to the polling interval property of the model. HMM and GMM algorithms are requiring a polling strategy to execute the model at constant intervals even if no sensor value has changed.

The class API appears similarly to a sensor device and exposes a resource to retrieve the current likelihood of a class. After each algorithm execution, the new likelihood will be copied by the execution engine as new value to its class resource. Virtual Sensors will send `GET` requests as `http://virtual-class.kitchen.home/{class_name}` to retrieve the current value. We decided to make this resource observable for notifying Virtual Sensors in a real-time manner.

Deployment Strategy

Numerical analysis systems performing the training task will generate a JSON document provided during the creation of a Virtual Sensor. The Virtual Sensor first validates the document according to the MaLeX schema to ensure semantic correctness. The general properties of the Virtual Sensor will be used to form a RDF document allowing its semantic discovery by following our domain ontology. Then, the newly created Virtual Sensor will seek for Virtual Class agents being able to execute the models and having enough computational power. The selection is made according to the complexity of the models (dimensions of the matrices), the available memory on the agents and their closeness to the data producers. Opting for Virtual Class agents located close by the sensors lessens the traffic propagation.

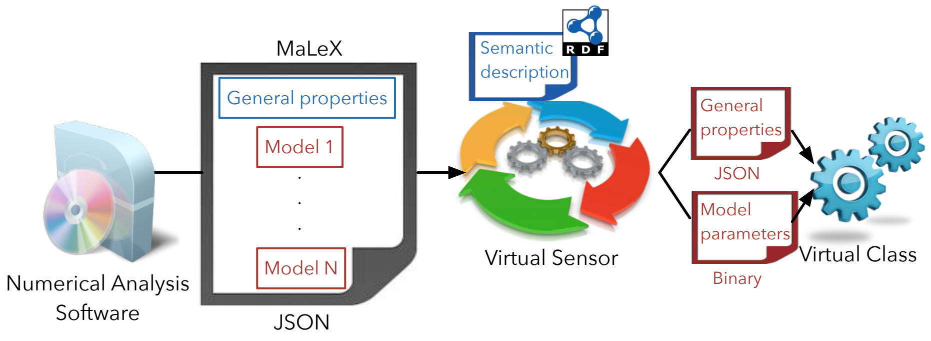


Figure 4.37: Models are exchanged between the different entities and adapted to the most appropriated format.

Although MaLeX provides a strong formalism for representing models, it cannot be used as such to create Virtual Classes. Constrained devices have not enough memory to handle big JSON files. First, the representation of the models composed of matrices made of float values is extremely verbose in JSON, as each character produces one byte of data. For this reason, we opted for a binary representation following the IEEE 754 notation of floats, already decreasing the required memory size by about 75%. This solution was preferred over the binary encoding of JSON, because BSON [348] forces float representation with 64 bits while 32 are sufficient. Using the IEEE 754 notation is in our case more efficient than BSON. Nevertheless, this does not allow a model deployment in one step on most smart things, the matrices being too big. We overcome this restriction by deploying matrices stepwise. The whole process of model deployment is shown in Figure 4.37.

Operating Modes

The decision making task is dedicated to the Virtual Sensor which will collect the current likelihoods of all its Virtual Classes. It then compares the likelihood values and, according to the Bayes rule, the model having the highest likelihood is designated as winner. Depending on requirements of client applications, this decision making task can be executed according to two different modes, selectable during the creation of the Virtual Sensor.

In the *End-to-end* mode, the Virtual Sensor will sequentially request all its Virtual Classes to retrieve their current likelihood. Once having collected all the data, it will be able to perform the decision and to return the winning class to the client, as visible in the left part of Figure 4.38. This mode, which can also be considered as on-demand approach, is particularly suitable for non time-critical applications where the round-trip time does not play a key role.

In contrary, the *Sync-based* mode is fully event-driven and is especially designed for real-time systems. Instead of retrieving the likelihoods only when requested, the Virtual Classes will notify their Virtual Sensor each time the likelihood has changed, as visible in the right part of Figure 4.38. The decision making is performed as soon as a likelihood changes and is cached on the Virtual Sensor. This allows to avoid the sequential requesting of Virtual Classes and improves

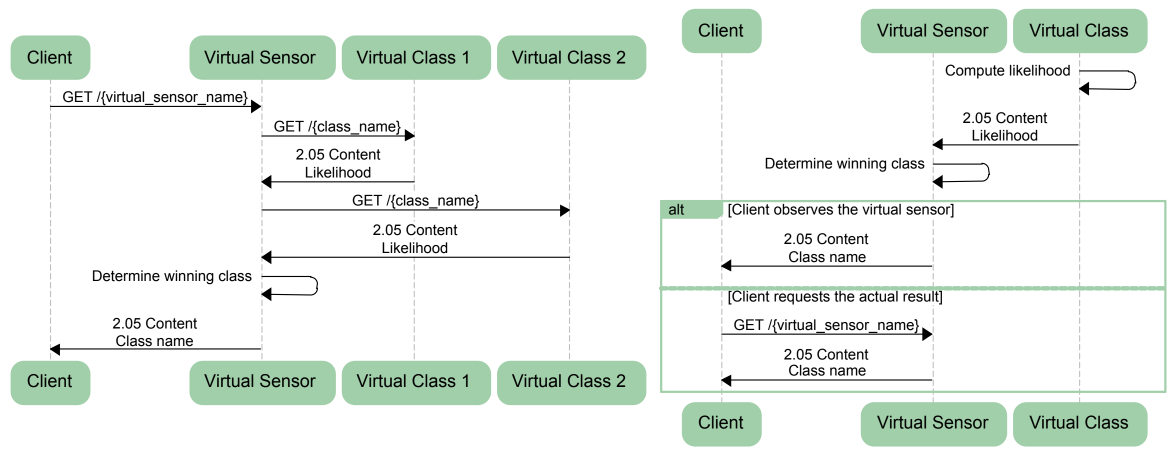


Figure 4.38: Clients can choose during the deployment of a Virtual Sensor the operating mode. On the left, with the end-to-end mode, the Virtual Sensor will sequentially retrieve likelihoods to determine the winning class. On the right, notifications are used with the sync-based mode to cache the winning class.

the scalability as well as the round-trip time. Clients can optionally observe a Virtual Sensor that will send notifications as soon as the winning class changes.

Evaluation

We evaluate our proposal on a concrete scenario of appliance recognition using electricity consumption measures. Smart plugs IoT devices placed between the appliance plug and the electrical wall socket provide the live data. The aim of the recognition process is to analyse the electrical consumption signature to recognise the appliance category, e.g. coffee machine and its state of use, e.g. stand-by. Five OpenPicus FlyportPRO WiFi modules are acting as Virtual Class agents able to execute HMMs. They compute class and state likelihoods through a standard Viterbi algorithm. A Raspberry Pi is used for running a Virtual Sensor agent on top of a Jetty Web server [349] .

The aim of this evaluation is to assess the performance of the architecture in terms of response time and scalability. The achievable identification rates are not part of this evaluation as these mainly depend on the types of algorithm and the complexity of the models.

End-to-End vs. Sync-Based Operation Although both modes of operation will obviously return the same value, they differ in the way the winning class is retrieved. As previously mentioned, the end-to-end mode should be privileged for occasional queries as it will generate several sub-requests to gather the current likelihood of the Virtual Classes.

In Figure 4.39 we show the round-trip time for 500 consecutive requests on the Virtual Sensor returning the current winning class. As expected, the sync-based mode performs better than the end-to-end one. This is due to the fact that the value is directly returned by the Virtual Sensor from its cache. This is reflected by a low average round-trip time of 32 milliseconds. From the cumulative density function (CDF) shown in Figure 4.40, one can see that 98% of the requests are answered within 50 milliseconds.

For the end-to-end mode, the process gathering the winning class is more complicated as the Virtual Sensor has to retrieve the likelihood of all its Virtual Classes before performing the de-

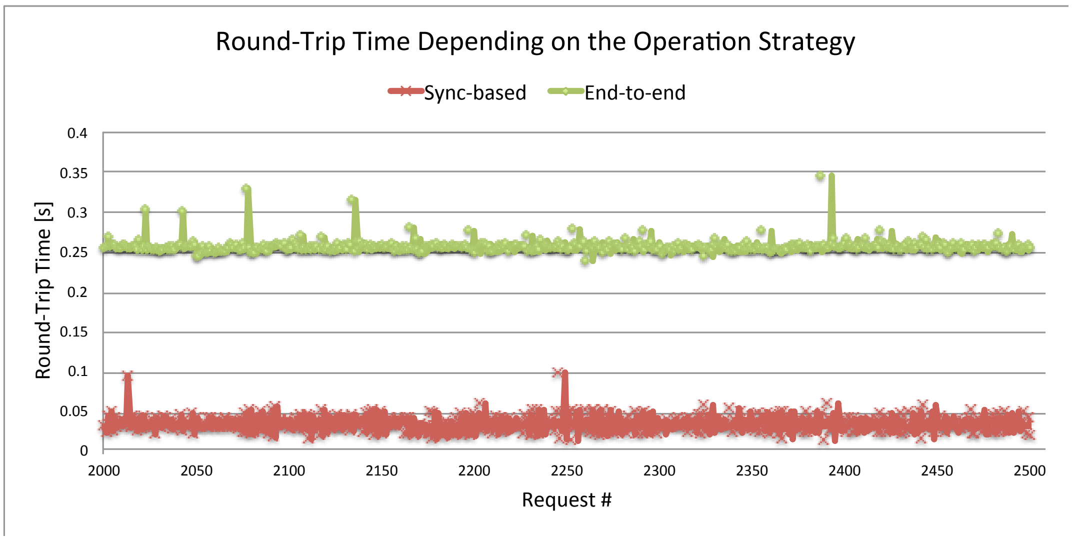


Figure 4.39: Round-trip time for 500 consecutive `GET` requests for retrieving the current winning class.

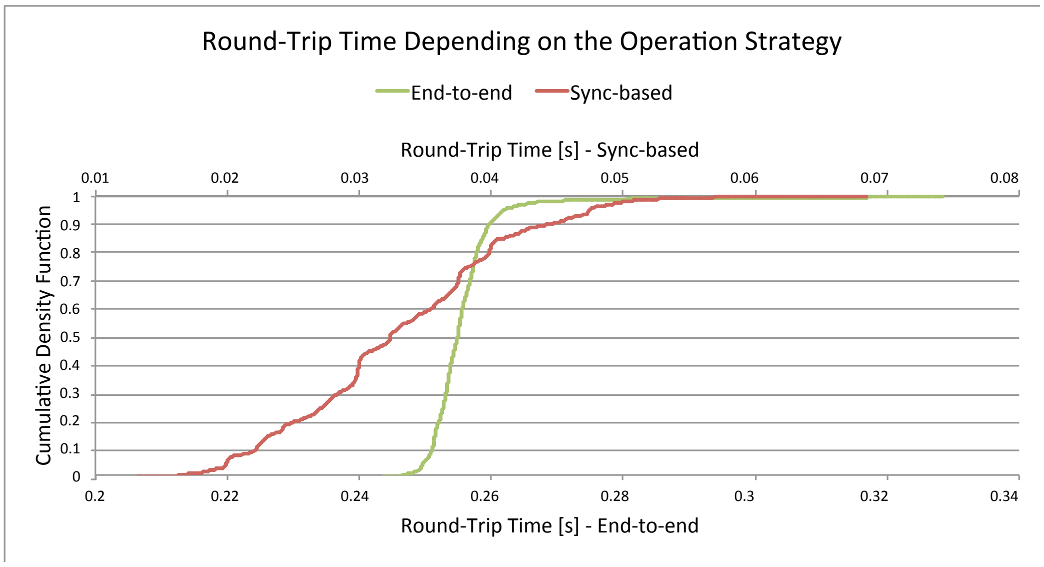


Figure 4.40: Round-trip time cumulative density function for the end-to-end and sync-based modes.

cision. In this case, the average round-trip time for 3000 consecutive requests is 256 milliseconds. From the cumulative density function, we can see that 90% of the requests are answered within 260 milliseconds.

The network traffic depends on several parameters that are neither predictable nor controllable such as the number of clients, the requests frequency, the number of Virtual Classes and their polling interval (computation of a new likelihood). From a general perspective, we can argue that the sync-based mode should be favoured when the notifications issued to inform the Virtual Sensor about new likelihoods have a lower impact than sequentially retrieving the value from the Virtual Classes.

Evaluating Concurrency In the second evaluation, we evaluate the difference between the operating modes in terms of concurrent requests. As for the previous evaluation, we performed requests using the end-to-end and sync-based modes. Figure 4.41 shows the round-trip time when having up to 100 concurrent clients each one issuing one request.

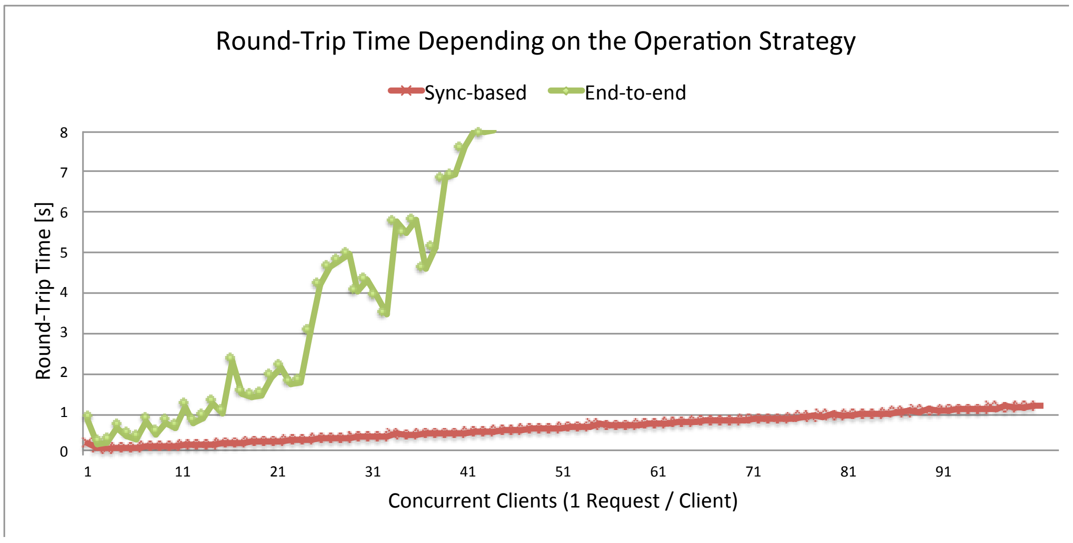


Figure 4.41: Round-trip time for up to 100 concurrent clients sending GET requests to retrieve the actual winning class.

Not astonishingly, the sync-based mode scales much better as the round-trip time of the end-to-end mode increases rapidly as the number of clients rises. With about 10 concurrent clients, the round-trip time reaches already one second for the end-to-end mode while this barrier is attained with 90 concurrent clients using the sync-based mode.

From Figure 4.42 we can see that the sync-based mode scales also much better. The success rate of the end-to-end approach drops abruptly as soon as the number of concurrent clients reaches 40. The sync-based mode, for its part, is not affected by the number of concurrent clients as the success rate does not go below 100%.

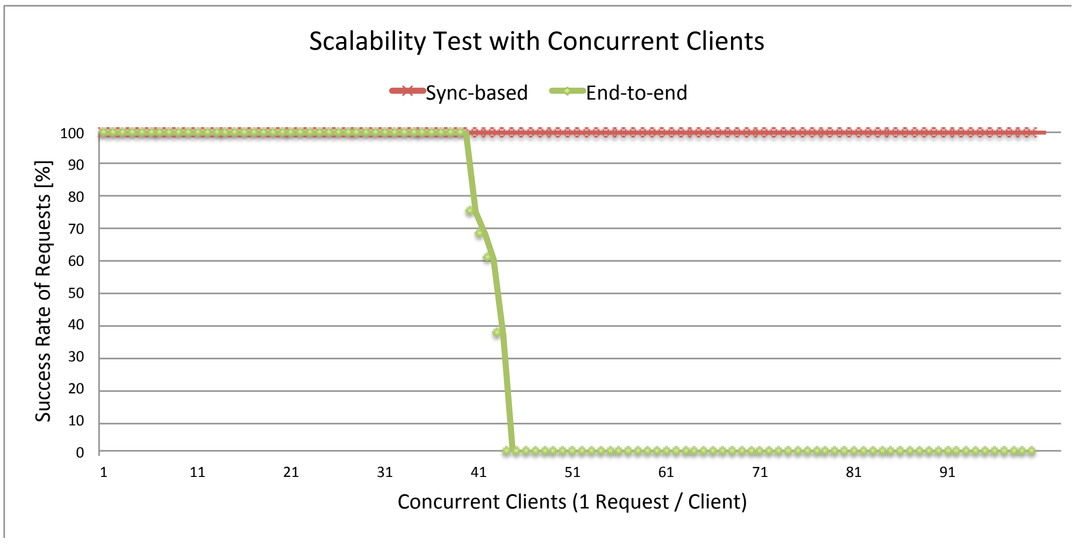


Figure 4.42: Comparing the scalability in terms of concurrent requests for the end-to-end and sync-based modes.

We can explain this limitation by the high amount of sub-requests that are generated by the Virtual Sensor to the Virtual Classes. Indeed each client request will result in five requests to Virtual Classes. The bottleneck are the Virtual Class agents running on the OpenPicus, having no threading and handling requests one at a time. This creates a long queue of requests having as result some timeouts on the client side.

Although it is unlikely that a large number of concurrent clients will access simultaneously a Virtual Sensor, the sync-based mode should be enabled in the case of concurrent accesses. Moreover, as Virtual Sensors can be used by mashups and other control applications to adapt the surrounding environment, the real-time aspect becomes important. The end-to-end mode performing worse in each of the scenarios, its use should therefore be avoided whenever possible.

4.3.4 A Mashup-Based Room Controller

In traditional building automation systems, engineers or administrators are usually creating control scenarios. Dedicated software such as the KNX ETS are used to define such interactions between automation devices. Unfortunately these applications are often developed by the alliance or company promoting a given network protocol and equipments. They are therefore limited to a single protocol so that it is not possible to mix technologies.

As explained in this thesis, relying on Web technologies and our concept of the Web of Buildings overcomes this limitation. Instead of implementing several protocols, control applications only need to be Web-compatible to interact with any kind of technology. Control scenarios also benefit from the Web's simplicity as it becomes easier to mix content provided by different sources, leading to *mashups*.

In this section we propose to rely on the concepts of Web 2.0 mashups and their physical adaptation to build control applications. We then come up with a building automation mashup architecture that is especially tailored for end-users.

Physical Mashups

Historically, mashups were defined by Yu et al. [255] as: *"Web applications generated by combining content, presentation, or application functionality from disparate Web sources. They aim at combining these sources to create useful new application or services."* Although mashups can be referred as composite applications, they come with their specific particularities [141, 43, 255, 48]:

Lightweightness and simplicity: Web standards such as HTTP, JavaScript and semantics are at the heart of Web mashups. Those technologies contribute to the lightweightness and simplicity as they are widespread and can possibly run on a large variety of clients including browsers.

Accessibility to a wide public: Thanks to their simplicity, Web mashups are consequently accessible to a larger public. While most composite applications are created by developers, graphical mashup editors tend to make application composition more understandable by end-users.

Rapid development: Traditional composite applications used in enterprise software business are often relying on heavy proprietary protocols or either use more standardised concepts such as the `WS-*` service stack. This makes composition hard to achieve as many barriers such as security, data representation and quality of service have to be negotiated first. Mashups are for their part making abstraction of all these hassles and are especially beneficial for ad-hoc applications, where time constraints are important. Moreover, mashups are intended to fit with personal needs of individuals or group of persons rather than enterprise constraints.

Mashups are combining data and services coming from smart things. As they can now involve any kind of object, this situation would benefit from a transition from code to graphical composition. End-users have to be an inherent part of the mashup creation process, allowing them to define by their own how the environment should behave. In this sense, mashup editors will play an important role in the creation of applications.

From Room Controllers to Building Automation Mashups

A building control depends on the type of building and the complexity of the strategies. Large buildings usually implement centralised optimisation strategies residing server-side, often relying on complex algorithms or specific applications. On the other hand, houses or apartments rely rather on local thermostats. In both cases, users have only little control over what happens in the room they use. In most situations, a device named the *room controller* placed in each room offers the possibility to adapt parameters such as temperature or lighting by adjusting thresholds.

With the emergence of smart things, more advanced scenarios can be imagined to adapt the environment according to certain conditions [256, 123, 63]. Room controllers have a limited graphical user interface if any. They are not suited to create mashups. We therefore propose to migrate towards virtual room controllers offering more convenient tools. In our vision, users will define their own scenarios with the help of a mashup editor requiring no programming skills. We dematerialise the concept of room controller from a dedicated physical device to mashups combining functionalities of several smart things that can be of any type.

The use of Web technologies advocated in the WoB plays a key role to build mashups. It essentially decouples the field technology from the automation. Furthermore, mashups bring several advantages over traditional room controllers:

Personalised scenarios: Providing a mashup editor that is able to graphically represent device capabilities enhances the potential scenarios that can be composed. Mashups can reuse the entire power of smart things to create higher-level applications that are personalised by the users controlling their entire environment.

Technology independence and evolvability: While room controllers depend on a specific building network technology, mashups are agnostic to this limitation. The Web indeed federates building technologies around the same paradigms and standards that are naturally supported by mashups. New devices can therefore be seamlessly added in existing mashups without worrying about potential incompatibilities.

Reusability: Mashups being small applications can be deployed themselves as Web resources. By following this concept, they become reusable components as any other Web resource like sensors and actuators.

Requirements

By considering the context of smart buildings and its general concepts, we propose a number of requirements to migrate towards mashup-based controllers:

User-friendly mashup editor: The creation of mashups must be kept as simple as possible without requiring any particular knowledge. Users should be able to compose their automation scenarios simply by combining building-blocks reflecting device capabilities.

Event-based interactions: In building automation system as well as in the WoB, data are exchanged between smart things with notifications. Mashups consuming data generated by sensors and other services have to use the same interaction principle relying on events to avoid polling and to minimise network traffic.

Automatic request generation: A composite application has to deal with various sources of information besides having to propagate its results. Along with the semantic descriptions of the resources (the notion of gates described in Section 4.1.3), a mashup has to understand by itself how resources should be consumed.

Lightweight scenario representation: While mashups are composed using a dedicated editor, they are in fact deployed on smart things endowed with a runtime application. In order to rely on a large variety of things, the exchange format should allow a deployment on constrained devices with limited memory and computational capacities.

In the following sections, we will propose a mashup editor as well as an architecture influenced by the aforementioned requirements, distributing and executing user-defined scenarios.

A Building Automation Mashup Editor

While several mashup editors are available, they do not integrate our concepts proposed in the Web of Buildings. We therefore opted to develop a new editor from scratch, basing its architecture and functionalities according to the WoB. This application is fully functional and uses state of the art technologies such as HTML 5 and JavaScript (jQuery and Bootstrap). Our solution is positioned to be easily usable by non-technical people. Lowering the entry barrier for building automation however has an impact on the complexity of the scenarios users can create. In order to keep it simple, our mashup editor restrains the possible composition to the condition/action pattern as follows *if <combination of several states> then <perform actions>*.

From Semantic Descriptions to Building Blocks Like other mashup editors, we represent device functionalities as building blocks. In order to make scenario creation as easy as possible, we opted for an approach that automatically generates a library of available sensors and actuators. We here contrasts with other editors requiring to manually define devices before being able to use them in mashups. To achieve this generation, we reuse the semantic descriptions of devices and their functionalities. As shown in Figure 4.43, the semantic description is used to generate the properties of building blocks, which are further combined to compose a scenario.

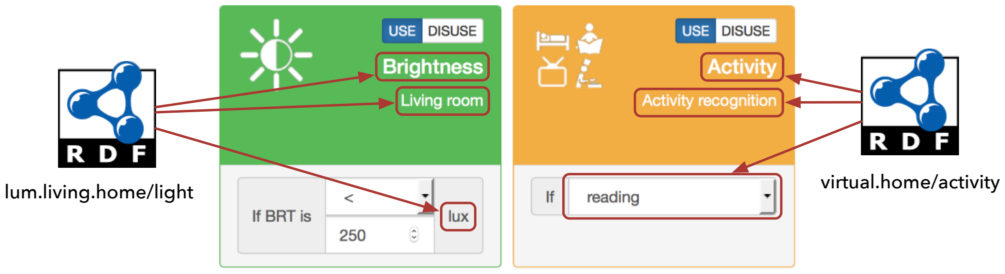
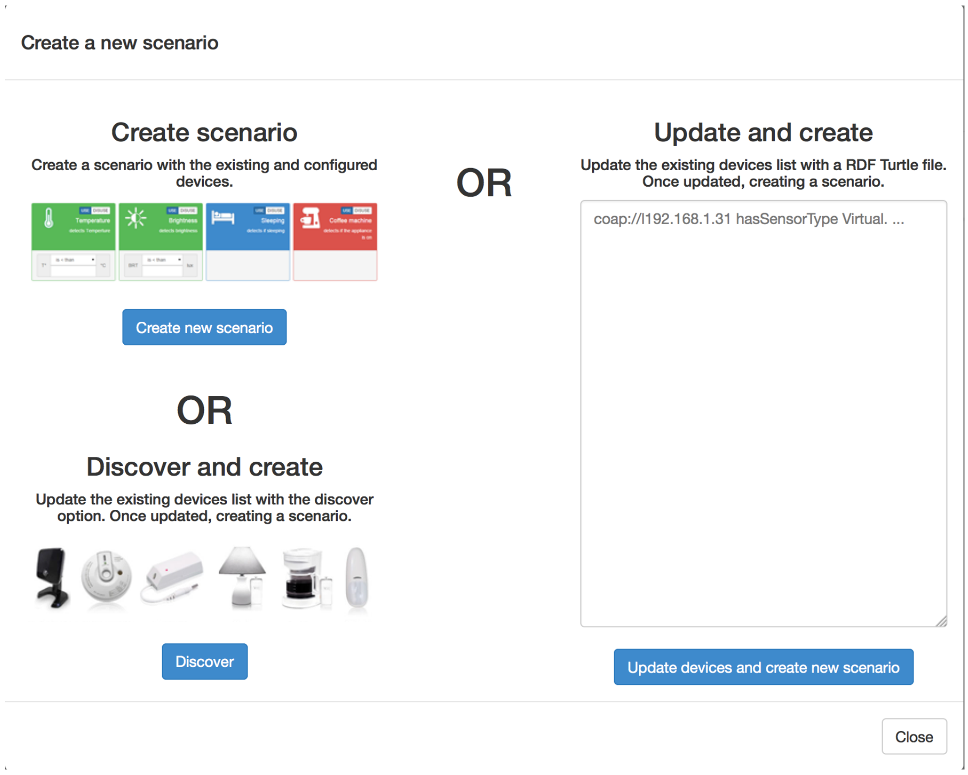


Figure 4.43: The graphical mashup editor automatically exposes building blocks to users according to the semantic description of Web resources.

The RDF semantic descriptions are translated into the application as generic JavaScript objects. These can then be graphically represented to the user. The major advantage of this approach lies in the wide range of devices that can be included in the library. Indeed any type of object or service offering its description using RDF that is compliant with our domain ontology is naturally compatible with the mashup editor. For example, Virtual Sensors which are an abstract representation of machine learning runtime models can be used as higher-level services like any sensor.

As shown in Figure 4.44, users have three different choices when creating a new scenario. If all the needed functionalities were already loaded in the application, then the users can reuse those from the library. This way requires that descriptions were previously loaded in the application. In the case of using a device that has been recently installed or not yet known, its semantic description has to be provided to the Web application.


Create a new scenario

Create scenario

Create a scenario with the existing and configured devices.

OR

Discover and create

Update the existing devices list with the discover option. Once updated, creating a scenario.

Update and create

Update the existing devices list with a RDF Turtle file. Once updated, creating a scenario.

coap://l192.168.1.31 hasSensorType Virtual. ...

Figure 4.44: Users can choose three different ways for creating a new scenario: reuse already known devices, discover the network or manually paste semantic descriptions.

Devices can be added in two ways. The first one requires more skills as the RDF description must be pasted inside the application. This mode is thought when users are defining scenarios including devices that are not yet installed. One could imagine that the descriptions would be provided on the manufacturers website. More easy to use, the included discovery module implements the concepts outlined in Section 4.1.4 to search inside the building network. Users can either detail what kind of device they are looking for, or simply discover the entire network.

In both methods the RDF description either manually provided or gathered through discovery will serve as foundation to create the building blocks shown to the user when creating scenarios.

Combining Building Blocks A scenario is composed of conditions and actions combined together. In contrast with other Web mashup editors such as Clickcript [305] requiring to link building blocks together, we decided to opt for a simpler interface, although limiting certain functionalities. In our mashup editor, users simply check whether a device has a role in a scenario. For defining the conditions, they set configurable values inside the building blocks such as thresholds or certain states, as shown in Figure 4.43.

The principle is identical for setting the actions that have to be performed, as shown in Figure

4.45. Participating devices are enabled by the users and the output value set. Actuators outputs are, whenever possible, represented as a list where the desired status can be chosen from. This is realised by relying on the datapoints defining what data is allowed by the endpoint.

Each scenario is a combination of logical *AND* relations between the building blocks. The actions will only be realised when all conditions of a scenario are fulfilled. In the case of *OR* relations needed, users have to define separate alternative scenarios by modifying the condition section. While this could be perceived as not user-friendly, we motivate this by the fact that other mashup editors including *ORs* in the conditions can become quickly incomprehensible, so that it is almost impossible to have an overview of what is performed.

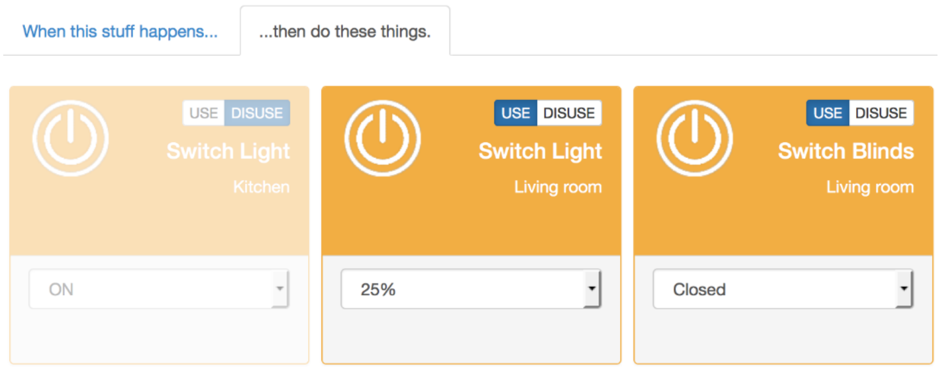


Figure 4.45: Users select the actions that have to be executed when the conditions are fulfilled. They can further parametrise these actions from a list of predefined values.

Scenario Exchange Format Once defined, the scenarios are exchanged with a mashup engine. More or less complex Web-oriented formats were already proposed [257, 258]. We observe that one of them fully satisfies our needs for simplicity. Indeed, as we aim to execute mashups on constrained devices, the representation format has to be as lightweight as possible to target devices with only little memory. For this reason, we defined our own exchange model based on JSON. The exchange format only comports information that is necessary to execute the mashup, as shown in Listing 4.9.

The structure holds two main parts where the conditions and actions are defined. Each building block is represented as an object inside the JSON with appropriate information. Conditions differ from actions as the conditions have an additional information indicating the rule applied to the measure. Typically, we indicate if the actual value should be lower, equal or greater.

```
{
   "creationDate":"8/1/2015 14:44:09",
   "name":"Reading",
   "desc":"Light when reading in the living room",
   "id":"xkghnqz1",
   "conditions":[
      {
         "url":"coap://lum.living.home/light",
         "unit":"lux",
         "rule":"<",
         "ruleValue":"250"
      },
      {
         "url":"coap://virtual.home/activity",
         "rule":"=",
         "ruleValue":"reading"
      }
   ],
   "actions":[
      {
         "url":"coap://light.living.home/dim",
         "unit":"%",
         "ruleValue":"25"
      },
      {
         "url":"coap://blinds.living.home/position",
         "unit":"%",
         "ruleValue":"100"
      }
   ]
}
```

Listing 4.9: Example of a scenario representation in JSON format for the conditions shown in Figure 4.43 and the actions shown in Figure 4.45.

A Mashup-Based Room Control Platform

The goal of the mashup based room control platform is to provide a framework that fulfils the previously discussed requirements. Rather than acting like other solutions only focusing on the graphical interface or the mashup engine [259, 260] [305], our proposition encompasses all these aspects. We here outline an architectural proposition that was not implemented.

As shown in Figure 4.46, the platform is a mixed environment between Web-enabled smart things and external services such as weather forecasting or room use scheduling. The mashup editor and runtime are decoupled from each other by a RESTful API. This allows to offer various kinds of editors that can run in browsers or as native mobile applications. The basic foundations of the WoB defining semantic descriptions and query are the only mandatory features.

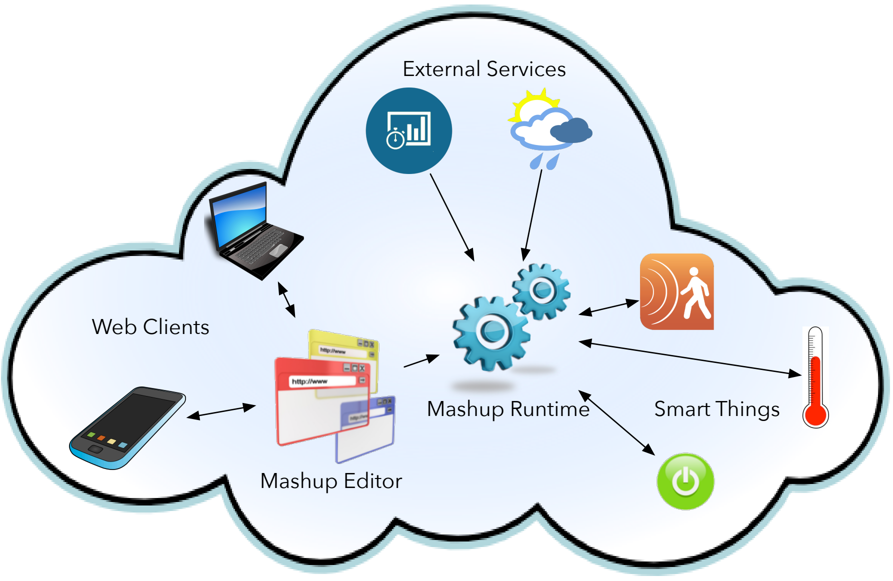


Figure 4.46: Platform overview of the mashup solution. Any kind of smart thing semantically describing its capabilities as well as external services can be used to define control scenarios.

The idea behind the room control platform is to delegate the execution of the scenarios to smart things able to understand the exchange format. Scenarios are therefore distributed throughout the buildings according to the location of where the actions have to be performed. This approach fosters autonomy and network independence by relying on a set of mashup runtime agents spread across the building. Deployment can be imagined from several points of view:

Autonomous: In the autonomous mode, the editor application automatically decides where a scenario should be deployed. According to parameters such as closeness with the devices involved in the scenario, it will choose the most appropriate smart thing able to execute mashups.

User-defined: Users may have preferences where mashups should be executed in very particular cases. In order to give them the opportunity to take this decision, users can look for mashup runtime agents available in the building network and manually select which one will execute a certain scenario.

Components Architecture

In this section we further describe the behaviour of the room control platform by focusing on the most important components of the architecture as shown in Figure 4.47.

Discovery Component This component implements useful functions for discovering several smart things involved in the creation of control scenarios. First, it is used to discover interesting resources available in the building network. The type plays no role as they can be sensors, actuators or higher-level services. The only requirement for them is to make their semantic description available, as it will be interpreted to dynamically create a graphical building block.

Furthermore, mashup-runtime capable devices are discovered during the deployment phase. As previously mentioned, this can happen autonomously with the application selecting which smart thing is best suited to execute the scenario. Otherwise, the discovery results are returned to the user, who will further choose a target.

Blocks Library The library contains already discovered resources that can be potentially used in scenarios. A block object is created by analysing the semantic description of the resource and is stored inside the Web application. Web applications can hereby use the concept of built-in Web storage of browsers to store the descriptions.

Blocks are classified in categories depending if they are a source of information for the mashup or if they provide ways to control the physical environment. *Condition blocks* include any kind of block making reference to a resource furnishing information. Sensors and external services are part of this category.

On the other hand, *action blocks* are made available to the user to react to certain conditions in the environment. The user will define their new state in the scenario. These blocks are essentially the actuators installed in the building. However, other kind of services allowing to update information can be considered. It encompasses enterprise services such as the ones dedicated to security, activity logging or even automatic billing [261, 262].

Deployment Component Once scenarios have been defined, the user can decide to deploy them on smart things running a mashup execution runtime agent. In a first step, the internal scenario representation is translated into the exchangeable JSON according to the schema.

Afterwards, the JSON will be deployed using CoAP on the most appropriate smart thing, either automatically chosen according to closeness, or selected by the user. The role of the deployment client is to manage the exchange of the JSON scenario with the mashup runtime agent.

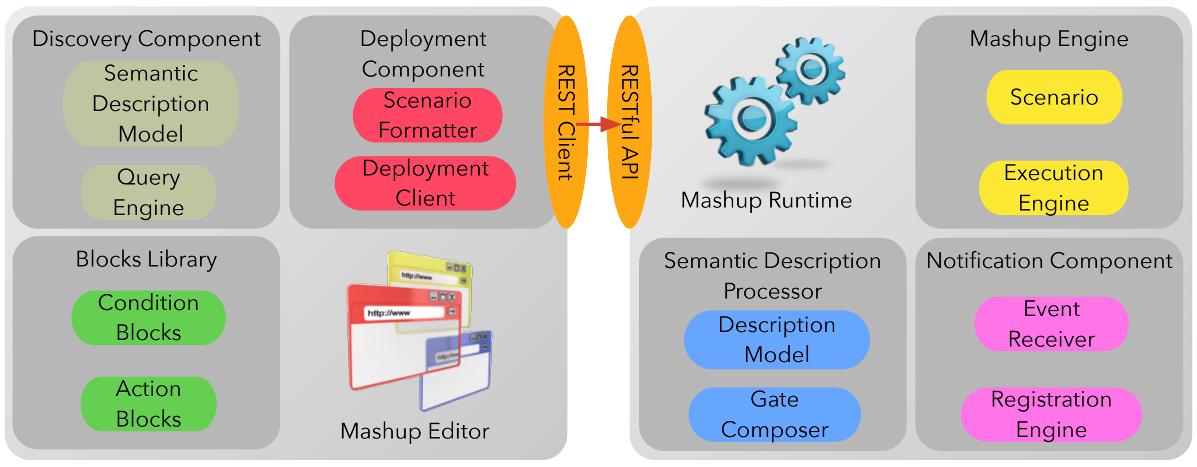


Figure 4.47: The mashup-based room control platform is divided into a series of components, each one performing a specific task. This architecture enables evolvability and allows components to be replaceable.

Semantic Description Processor Before executing a control scenario, the runtime needs to know how to consume the resources referred by the blocks. The semantic description processor uses for this purpose our domain ontology to understand the meaning of a resource.

With the notion of gates introduced in Section 4.1.3, a machine is capable to build requests by itself without requiring a human. For this purpose, abstract representations of the resources are automatically composed. These are used by the execution engine and by the notification component to communicate with a resource without worrying about how it has to be consumed.

Notification Component In order to optimise communications and to reduce network traffic, changes of state are forwarded by devices as notifications, avoiding polling. Before notifications can be received, the mashup runtime has to register itself as consumer at resources providing the input data. The registration engine is devoted to this task by reusing the semantic description processor to understand what notification methods are provided and what is the registration procedure.

Once registered, notifications will be received and handled by the event receiver. It is responsible to adapt the application to the type of notification used. In the case of HTTP callbacks, it will create a dedicated resource to be called every time the state on the producer changes.

Mashup Engine The mashup engine is at the heart of the runtime agent. It stores the JSON scenario that will be processed. The representations can either be kept in the original JSON format or transformed internally if necessary. However, the exchange format being kept as simple as possible makes the agent technology-agnostic. Indeed, how the engine is implemented should not impact the scenario processing as long as the JSON is interpreted correctly.

The engine will be provided with the values received by the notification component. Each time a new value is available, the scenario will be processed to determine the new states of the action building blocks. Requests to the actuators will be sent at the end of the processing to update the physical environment.

Runtime RESTful API Like every component of the Web of Buildings architecture, the mashup-based room control platform relies on RESTful APIs homogenising the interactions, shown in Figure 4.48. Mashup editors and any other type of clients can use it to deploy building control scenarios into the building network by providing their JSON representation.

Furthermore, the API offers services to manage mashup instances (creation and deletion), as well as to gather information about the current state of execution. The error log provides information about unexpected situations, such as an action building block not responding to a request. It is also possible to know what actions were performed on the actuators by consuming the output resources.

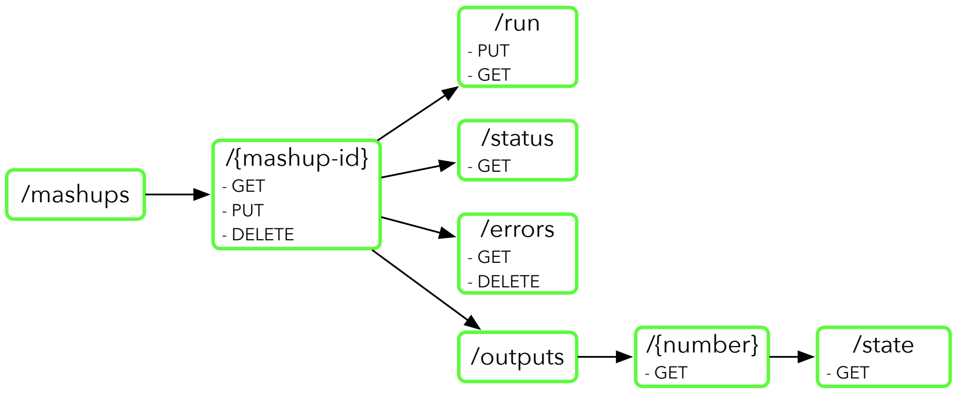


Figure 4.48: RESTful API of the mashup runtime agents in the room control platform.

We would like to remind that the mashup runtime has not been the subject of a prototype implementation. The architectural design that we have outlined should be more considered as a guideline, providing insights on the main components and their role.

4.3.5 Related Work

The idea to distribute machine learning algorithms for building automation purposes has already been explored in a few works. In [263], fuzzy logic is distributed across several agents by furnishing a XML document containing the model parameters. With this document, agents are able to automatically generate Java objects for the JADE agent framework, which is used for synchronisation between the entities. Inference agents collect the outputs of the fuzzy logic agents and make a decision before executing some predefined rules. Although going in the direction of decentralised systems, this solution requires a rather heavy infrastructure of powerful nodes and is probably not compatible with current IoT devices (a Java Virtual Machine is required). Fuzzy logic is also used in [264] where static agents perform the learning. The logic is then distributed on mobile agents executing the rules. Their fuzzy controllers were able to learn 128 rules in the first nine hours with a user spending all its time in the experimental room. The detection accuracy and the usefulness of the rules are however not mentioned.

While these architectures allow to deploy machine learning on distributed agents, they base their work on fuzzy logic, which has already been distributed in several project in the past [265]. In our approach, we target to deploy models issued from expectation-maximisation algorithms [266] achieving a better modelling of the density distribution. Moreover, the architectural choices often go in the opposite direction of an open and Web-based platform.

With an infrastructure composed of sensors and actuators accessible over Web protocols, Web mashups can be designed to associate functionalities. Graphical mashup editors allow end-users to define themselves higher-level applications. ClickScript [305] is a Web mashup editor showing building blocks that can be linked together. The JavaScript runtime is directly embedded in the Web page, so that mashups will not be executed if the Web page is not loaded into a browser. This editor has been adapted in [267, 51] to be compatible with HTTP REST and SOAP services. The Yahoo pipes [350] and the MashableLogic [351] Web mashup editors rely on preconfigured data sources that can be of several types: RSS feed, JSON and XML, all coming from Web pages. They are principally intended to build business-driven mashups rather than representing sensors and actuators.

Actinium [259] is a RESTful mashup runtime container developed with the Californium CoAP server and the Rhino JavaScript engine. Mashups written in JavaScript can be deployed and managed over a RESTful API. Each mashup runs in a separate container ensuring minimal security constraints. Although providing a Web-based infrastructure to execute mashups, the container requires a significant amount of CPU and memory, preventing from a deployment on smart things. The T-Res [260] project aims to deploy mashups on constrained devices working with the Contiki operating system. A RESTful API is also exposed to manage mashups provided in the Python language. Each mashup runs in a separated container avoiding access to other resources. Although their work goes in the direction of in-network processing, their implementation showed that only the runtime agent already takes 93% of the available memory of their WISMote device, leaving only little space for mashups.

In our approach, we did not deploy directly runnable code but a data structure in JSON representing mashups. Despite limiting the possibilities in terms of functionalities, this avoids executing containers. Moreover, the mashup representation is language-agnostic so that any kind of implementation can execute them.

4.4 Intelligent Adaptation Level

Intelligent Adaptation
Machine learning training
Machine learning adaptation

Smart buildings are equipped with many sensors to measure the physical environment and to further apply control strategies. All these data constitute an asset that can be used for more intelligent control. In this direction, we observe an increased use of data-driven approaches able to build more complex models. Machine learning applications are one of those emerging approaches. They are built around data analysis algorithms that are especially tailored for large datasets.

Machine learning provides ways to gather knowledge from historical data. In the context of smart buildings, this knowledge can include different domains such as activity detection/prediction, appliance recognition, and thermal inertia estimation. Before this can happen, mathematical models have to be trained on datasets. Due to the heavy infrastructure required to run these algorithms, training is usually performed server-side or using cloud services.

In our vision of a Web-enabled building architecture, we are investigating here the possibility to incorporate the training process of machine learning within the WoB. The feasibility will of course depend on the capabilities of the nodes composing the building network. In any case, such high-level process will constitute an interesting approach as our proposition reuses the functionalities provided by the other layers of the WoB. We also show how models can be adapted efficiently, hereby limiting the amount of data required and complying with in-network approaches. The solutions presented in this layer should be considered as guidelines towards a Web-enabled ML architecture for the training process. The concepts have so far not been implemented and tested in real situations.

4.4.1 Training

In the machine learning process, the training task consists of analysing datasets in order to find relations or patterns between the data. These datasets can be especially prepared to represent a problem or can be composed of real observations. The training task can be either supervised (inputs and desired outputs are given), unsupervised (no labels are given) or a mix of both [268].

The training is typically a heavy task requiring a large amount of CPU and memory to process a rather large dataset. For example, the expectation-maximisation training technique used with HMMs and GMMs require to have the entire dataset at disposal [269, 270]. In order to find relations between the observations, the dataset indeed needs to be processed in one batch. The training needs therefore to run on a dedicated machine able to process large amounts of data.

Machine learning being mostly used in the academic research, dedicated data analysis software are used to train and test various algorithms, and to compare their results. These software are not intended to communicate with other systems in order to share their output models. This limits the deployment for applications in smart buildings. Furthermore, software such as Matlab among others are not open and cannot be supplemented with a RESTful server to homogenise the application layer.

In spite of the aforementioned limitations, we discuss in this section how training can be interfaced with a Web architecture, so that existing algorithms can be easily integrated with the Web of Buildings. We voluntarily make abstraction of several machine learning characteristics to offer a global solution that can be adapted to a wide range of applications. We start by exposing the concepts describing how the training process can interact with the Web of Buildings (distributed storage, Virtual Sensors and mashup platform). Then, we provide solutions to technically integrate data analysis software within the building network by means of proxies.

From Historical Data To Virtual Sensors

The machine learning process analyses datasets in order to find patterns between the observations. The output of this process is a set of mathematical models further used for classification purposes. In the context of smart buildings, the historical data of sensors represents the training dataset that has to be processed. The data analysis software has therefore to retrieve the useful data it needs. This is realised by accessing the distributed data storage presented in Section 4.2.3 and by performing the appropriate requests (assuming storage has been previously activated for the resources). The historical data will be copied in the memory by using the applicable internal format. Once those steps completed, the data analysis software can execute the training algorithm.

After having trained the models, the data analysis software transforms them according to the MaLeX JSON schema. The obtained JSON document including the general information and the models is then further used to either create new Virtual Sensors or to update existing ones if parameters have changed. From that moment, the models will be deployed on compatible smart things. Figure 4.49 shows the general concept where other levels composing the WoB are used in the training process.

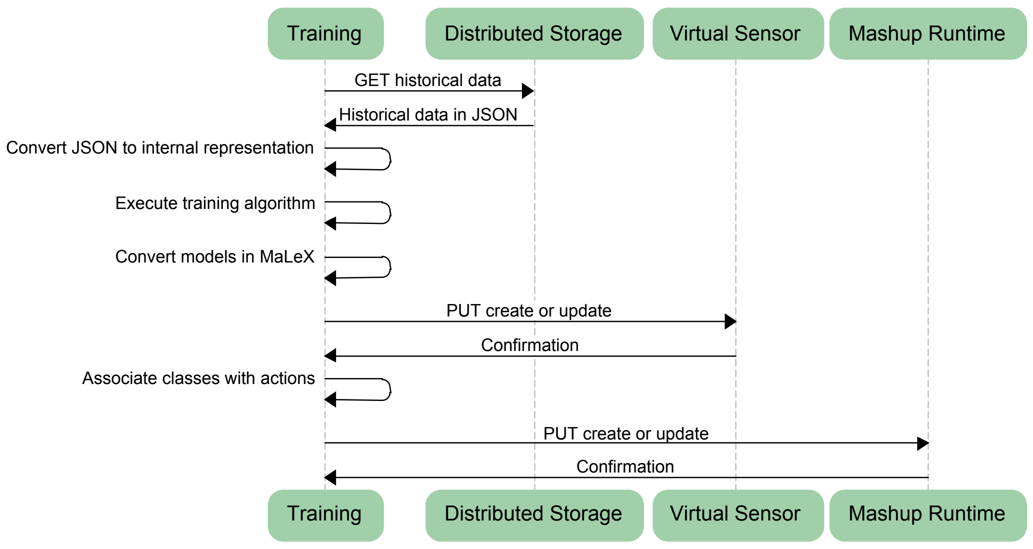


Figure 4.49: The machine learning training process is positioned on top of other levels. It retrieves the needed historical data from the in-network cloud storage. Once the models are trained, they can be deployed as Virtual Sensors. Furthermore, the training process can automatically define control scenarios in terms of mashups.

Automatic Scenario Composition

In addition to train classification models according to live data, the classes defined in the models can be used to automatically adapt the environment. The state of actuators also gives interesting information that can be reused to determine in which state the environment should be when particular conditions are observed. To mention an example, the training could find relations between the values of motion and illumination sensors and the state of light switches. This could result in one or several scenarios automatically managing the lighting in a room. We can therefore endow the training process with the task of creating control scenarios.

To achieve such a level of automation, each class is associated with a set of actuators along with the state in which they should be. This information is deployed separately according to the concerns. Mathematical models will be abstracted in the form of Virtual Sensors to provide control scenarios with classification algorithms. The automatically generated mashups will consume information from physical sensors and Virtual Sensors, and according to the conditions, drive actuators. This process of automatic control scenario composition is actually depicted in Figure 4.49.

This approach can be seen as an automation of the building automation, as scenarios are learned during operation rather than being manually defined. The main advantage resides in the continuity of learning and the automatic adaptation to changes of user habits or seasonal conditions. Users are not required to define control scenarios by themselves, but rely on machine learning intelligence. After a building having been equipped with machine learning, it would start in a collecting phase, awaiting to have sufficient historical data to begin the training.

We have to note that many research is going in this field. Our intention is here to foresee a compatibility of our proposal (and even a continuity) with what will be applicable soon.

A RESTful Training Platform

Before the training can be performed (and consequently control scenarios automatically created), the data analysis software has to be integrated into the Web. The Web of Buildings makes an intensive use of technologies dedicated to sensor networks, such as CoAP. While the strength of CoAP is to allow constrained devices interacting over a RESTful protocol, it has not yet made its way into a lot of APIs. HTTP still predominates in business and server-side applications. We will consider a concrete scenario where the training application is written with Matlab and will propose an architecture overcoming the limitations. The concepts that will be presented are however intended to be applied to any other data analysis software.

The Virtual Sensors presented in Section 4.3.3 are especially thought for machine learning applications and already provide a HTTP interface. This is not the case for the distributed data storage introduced in Section 4.2.3 and the mashups runtime discussed in Section 4.3.4, which are entirely CoAP-based. Although providing a CoAP library would solve this shortcoming, each data analysis software would require its own one, case by case, therefore complicating the integration task. In our solution shown in Figure 4.50, we aim at providing a set of proxies directly compatible with HTTP, which makes them integrable with almost any software or language. Proxies have already proven their suitability in several Web architectures [50, 105, 123, 51].

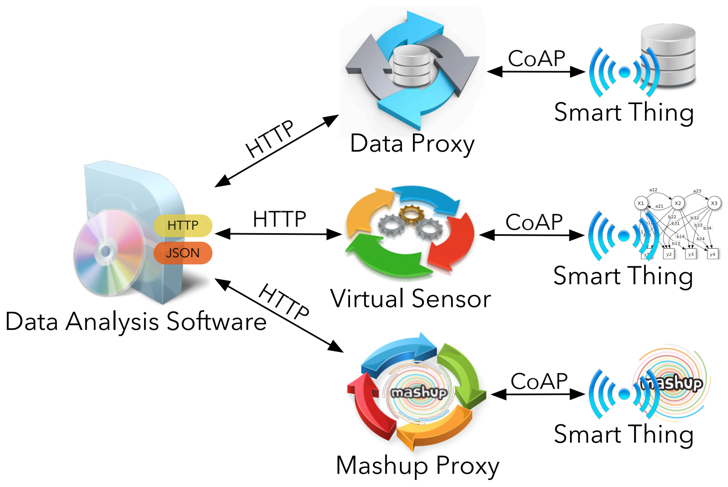


Figure 4.50: Data analysis software can interact with other services of the WoB through Virtual Sensors or dedicated proxies.

The role of the proxies is to offer RESTful Web services accessible over HTTP, hiding particularities related to CoAP and the use of multicast. As the building blocks presented in this thesis rely on multicast communications (HTTP is connection oriented and does not support multicast), the proxies have to perform tasks related to discovery using CoAP. For this purpose, we introduce two proxies dedicated to storage and mashups:

Data proxy provides a RESTful API to use the cloud-like storage. The interface is a simplified version of the CoAP one as the multicast-related task (storage space discovery) is performed by the proxy itself and not the by client. This means that only two services are exposed: storage need announcement and historical data retrieval. For each request, the proxy first discovers (if not already known) which storage space manages the resources and then uses the corresponding group multicast address.

Mashup proxy allows to create and manage control scenarios over a HTTP RESTful API also hiding specific parts related to discovery and deployment. From a client point of view, it appears as the device that will actually execute the mashups. The proxy offers the same API as for native mashup runtime agents. Control scenarios will however be deployed on smart things running the runtime agent. Requests are translated into CoAP and relayed to smart things. The proxy therefore holds internally a table mapping the ID of a mashup with its actual URI pointing to a runtime agent. By following this approach, the proxy is entirely transparent for clients.

Considering a machine learning application running with Matlab, it becomes quite easy to communicate with Virtual Sensors and proxies over HTTP. The urlread2 [352] library comes with a set of Matlab functions to build HTTP requests and to parse responses. The response is provided as an object containing all the headers and payload. By using this library, Matlab applications have the entire control over the headers required in the REST architectural style, including content negotiation and access to response codes.

```
[output,extras] = urlread2('http://data-proxy/storage/home/kitchen/temp', '
    GET', '', [], 'from', date1, 'to', date2);
if strfind(extras.firstHeaders.Response, '200 OK')
   json2data = loadjson(output);
   % process the historical data
end
```

Listing 4.10: Matlab code excerpt using the urlread2 and JSONlab libraries to retrieve historical data through the Data Proxy.

The handling of JSON data whether to parse incoming data (historical payload) or when creating a document (formatting models according to MaLeX) is made easy by using the JSONlab [353] library. It translates JSON into Matlab structures. In Listing 4.10 we provide an example how both libraries can be used to perform HTTP requests.

4.4.2 Adaptation

Machine learning allows to generate models according to how the building is used by people, or how to react to thermal changes. The accuracy of these models depends mainly on the amount of available data and the associated algorithms. It may however happen that models are not reflecting the right behaviour or are leading to wrong decisions. In this case, users may feel the automation system as inconvenient. In order to increase the accuracy of the learning system, some machine learning algorithms allow to perform an *adaptation* of the models. It implies the collection of a time window of past data including the correction performed by the users [271]. The adaptation is then an extra part of the learning that adapts the models to the dynamic behaviour of the users or to changing conditions of use of a building.

Adapting Models To The Context

In the retraining approach, models are adapted at some frequency, as for example every night for smart buildings. This frequency has an impact on:

- The accuracy of the models
- The quantity of data required and thus the impact on the network traffic
- The computational power and memory required

The frequency of adaptation is also bound to user satisfaction. In some buildings, adaptation to change of seasons (sunlight and temperature) will be enough. In some other buildings, adaptation will have to catch up more quickly with change of activities.

In order to detect inconvenient actions made by the system, such as automatically generated control scenarios, it is necessary to give what is called the *ground truth* as input to the training process. In the context of building automation, with corrective feedbacks [272, 273, 274], the ground truth will be provided by the users in form of reactions against the choices (classification) made by the automation layer. It mainly includes the management of HVAC and lighting. Let us illustrate this with a concrete example related to lighting. After the data collection phase, models will be trained and executed by the ML runtime. It could however happen that models are inaccurate and wrong decisions are performed on the lighting. The user will react to this by acting on the lighting control. This reaction is a key information indicating that the model led to a wrong decision and has to be relayed at the training level.

The role of the adaptation level is therefore to detect these situations and to report them to the training level. This information will serve to trigger the training by augmenting the dataset with new measurements describing the states of sensors and actuators when the situation arose. The training process will consequently try to refine the models by incorporating these observations.

An Adaptive Agent Architecture

In this section, we propose an architecture based on an agent behaviour that will detect user actions going against system decisions. We reuse the concepts presented in the other levels, such as observation, to get informed as soon as an actuator changes its state, or when a control scenario drives an actuator. The interaction with the other levels is shown in Figure 4.51.

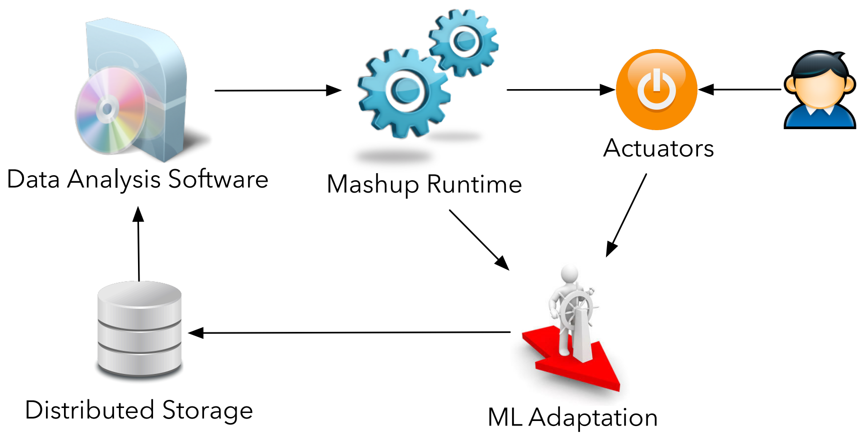


Figure 4.51: The adaptation layer will use actuators states and the actions performed by control scenarios to determine if an action was performed by the user. Conflicts will be notified in the distributed database so that the training process can refine its classes.

In our architecture, an agent will be responsible to watch a specific actuator that is concerned by one or several control scenarios. In order to detect actions performed by the user, it will observe the specific actuator and the outputs of mashups driving this actuator. In the case of the actuator changing its state, it will await a certain amount of time to be possibly notified by a control scenario, informing that it performed the change. In the case no notification is sent by a mashup, the adaptation agent considers the change as performed by a human.

The training level can decide to deploy adaptation agents into the building network by relying on an adaptation proxy. It offers a transparent API to manage resources representing agents. The proxy will handle all the discovery part of adaptation agents by relying on CoAP and multicast. The roles will be distributed to agents according to their location in order to minimise the forwarding costs in the network. Figure 4.52 shows the API exposed by the proxy. Data analysis software can create new adaptation agents by PUTting to the `/adaptations/{id}` resource, giving in the payload a JSON containing the necessary information. It has to include the URI of the concerned actuator and one or several references to the mashups controlling it. Additionally, the training should provide a URI referring to a distributed storage resource that will store the conflict notifications. The API exposes a set of sub-resources for each adaptation instance, so that, all these information can be modified during runtime by the training if, for example, new control scenarios drive the actuator.

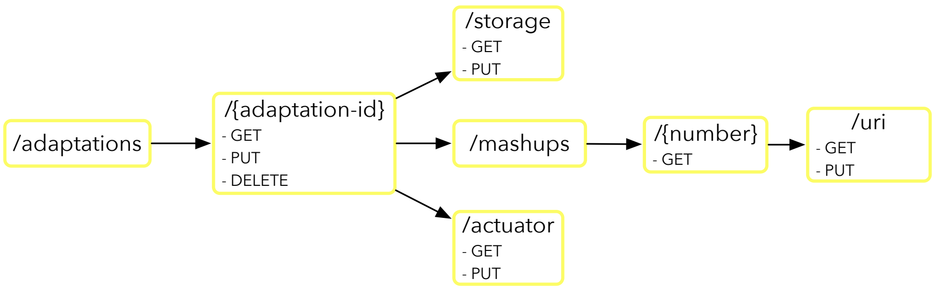


Figure 4.52: RESTful API of the adaptation proxy.

4.4.3 Related Work

Machine learning has been applied to data generated by sensors placed in buildings in several projects. Models are trained over temperature time-series in [252] to predict future temperatures in the building, and combined with comfort preferences in [275]. In the same manner, expectation-maximisation algorithms are applied in [276] to the optimisation of the energy consumption for thermal regulation by using weather, comfort and occupancy informations. In [277], a modelling of the thermal inertia of rooms is included in the models to achieve even greater precision in the thermal optimisation.

While these works use the same types of algorithms as in our proposition, they all perform the training in a centralised way, using dedicated powerful hardware. Moreover, the datasets used for the training are furnished in batches either for simulation or copied from building automation systems installations. Contrarily to our approach, no infrastructure is proposed that easily integrates with the building network.

The Web has met machine learning through some cloud services such as the Google Prediction API [346] and BigML [347], offering services to train models. Both expose a RESTful API to train and to test models. Although easily integrable in our WoB architecture, their use remain questionable from several points of view. First, it is not possible to select the type of algorithm nor to customise the models. In addition, the training datasets can sometimes hold confidential information (e.g. presence information and access control credentials) that have to be transferred to the cloud platforms. How these data are used internally is not specified and is hidden to the users. Moreover, it is not possible to know where they are stored and who exactly has access to these data.

4.5 Summary

In this section, we summarise our proposition of the Web of Buildings, We start by emphasising the main previous works in relation with our architecture. Second, we discuss the key points and limitations of our vision of Web-based smart buildings. Finally, we highlight the principal contributions of each level.

4.5.1 Related Work

The idea to use RESTful APIs in relation with physical objects (also called smart things) has been deeply investigated by Guinard [48] and Trifa [123], leading to the *Web of Things*. Since then, many projects have successfully applied this paradigm to interact with things [36, 137]. Due to the rather high verbosity of HTTP and the mandatory use of TCP [184], CoAP has recently emerged and positions itself as a lightweight version of HTTP. This protocol follows the REST guidelines and is especially optimised for constrained environments, hence its deployment in sensor networks [73, 148, 247, 132]. Devices which cannot run a Web server can be made compatible with a Web middleware through the use of smart gateways, translating functionalities to RESTful APIs [123, 129, 131].

Applying Web technologies goes far beyond the REST architectural style. The semantic Web offers powerful concepts and technologies in order to let Web resources be understandable by machines. The Resource Description Framework (RDF) comes with the notion of ontologies describing knowledge and the associated semantic descriptions [217]. Part of RDF, SPARQL is a query language for descriptions that can be used as discovery mechanism for smart environments [225]. Augmenting RESTful APIs and the Web resource with the notion of semantics allows to achieve some interoperability between systems [218, 215].

With an ecosystem of RESTful APIs, it becomes quite easy to build opportunistic applications composing from miscellaneous sources [255]. These applications, called *mashups* can be graphically designed by end-users to reflect their desired behaviour. In this direction, the ClickScript editor [305] has been especially adapted for the use with smart things [267, 51]. The Actinium runtime container [259] proposes a server infrastructure to dynamically deploy and run mashups. It makes an intensive use of CoAP to communicate with the devices. In a similar approach for executing mashups, T-Res [260] avoids the necessity of dedicated servers by executing mashups directly on nodes. Although their performance is satisfying, their sandboxed runtime container requisitions almost all the devices' memory.

In order to take advantage of all the data provided by the sensors, data-analysis approaches can be implemented in buildings. They can target a better management of the HVAC [252, 276] or can increase the user comfort [275]. Recently machine learning has also been applied for modelling the thermal inertia of buildings [277] and activity recognition [89]. Although providing useful information, data-driven systems remain only accessible to expert people. In order to overcome this limitation, the Google Prediction API [346] and BigML [347] are initiatives striving to lower the required knowledge for developing ML applications. This is achieved by offering ML as a service through RESTful APIs. Unfortunately these commercial solutions are rather opaque, so that it is not possible to drive the learning and testing by selecting the most appropriate algorithm.

4.5.2 Discussion

A central concern with automation systems in smart buildings, is the interoperability between devices of different technologies and manufacturers. Using a RESTful architecture to interact with smart things promotes loose coupling while relying on Web standards. Different parts of the building automation system can be implemented independently according to their specific requirements without affecting the overall operation. While RESTful APIs were suffering from a lack of formal service contracts, the semantic Web has brought an elegant solution to this shortcoming. More than just describing capabilities, it comes with a powerful set of technologies and methods ranging from languages up to query mechanisms. Combined with the lightweightness, simplicity and reusability of REST, we believe it forms an application layer that fosters homogeneity.

We can however point a major drawback preventing from real deployments in buildings: the lack of security and privacy. Smart buildings are requiring a high level of security to prevent abuses from people with bad intentions. In addition of causing economical damage by perturbing the proper working of processes in buildings, the existence of humans could be directly threatened. Because of the quite youth of the used protocols such as CoAP, security has not been a major topic so far. The research community has become aware of this problematic and is currently defining step by step security layers especially thought for constrained environments [354].

In our framework, we temporarily get around this shortcoming by advocating autonomy and self-sufficiency, avoiding the use of external services. This allows the building network to be disconnected from the rest of the world, and thus limits potential threats. Pushing services usually located server-side directly in the network in a cloud-like manner has other advantages. First, we demonstrated that the available and underexploited computational power and storage capacities of smart things can be used instead of a dedicated infrastructure, with as consequence to lower the costs. Second, we could promote self-organisation behaviour of our framework, so that building automation systems require less human intervention with potentials to reduce the time-to-market for new applications. Finally, the ease of installation following the plug-and-play concept opens the path for domestic use cases where no particular skills are expected.

Current frameworks dedicated to building automation only exploits static control rules. In our architecture, we make a first step towards the integration of machine learning applications. The field of machine learning is a rather large domain where plenty of algorithms and methods are available to find patterns. To include the largest range of applications, we focused on the integration of expectation-maximisation algorithms, which are widespread. Integrating machine learning as an inherent part of the Web is a novel challenge. Training algorithms require a large amount of CPU and memory and are therefore naturally not distributable on smart things. Data analysis software can meanwhile be part of the Web of Buildings by reusing services provided by other levels.

In contrast with other works focusing on smart buildings, our proposition encompasses all underlying services such as naming and discovery. Although each component can be used individually, they form together an ecosystem around REST and especially CoAP. This ecosystem comes with one major advantage that is particularly noticeable in constrained environments: participants only need to be RESTful as all services are provided this way. Implementing several underlying applications stacks such as DNS, UPnP among others not only tighten coupling but also wastes memory. The REST architectural style plays therefore a major role in the homogenisation of all services that can be found in a building.

4.5.3 Contributions

In this chapter, we presented our Web of Buildings framework and its four layers. Our proposition should not be considered as a strictly layered architecture but, it should be perceived as a set of architectural guidelines helping and facilitating the development of smart buildings applications around smart things.

In the Management Level, we provide a set of general purpose services that are the foundations of other levels. First, we propose a hybrid REST server library serving HTTP and CoAP requests transparently. Furthermore, we show how a Web-based naming can be distributed autonomously within the building network. Then, we discuss the importance of semantic descriptions to achieve interoperability between devices by relying on a domain ontology. The descriptions are further used to provide a discovery and query mechanism leveraging on the Resource Description Framework.

In the Field Level, we discuss how the REST architectural style should be applied to sensors and actuators. For devices that are not IP-based and cannot offer their services through a Web server, such as legacy building automation systems, we propose to rely on smart gateways exposing REST APIs to allow a transparent communication. We also emphasise on the importance to store historical data where it is produced. Following this approach, we provide a cloud-like storage reusing capacities of smart things. The comparison with commercial cloud services shows that our solution is competitive by means of performance.

In the Automation Level, we emphasise on providing a platform to distribute machine learning models inside the building network. By abstracting their specificities, classification tasks are represented as Virtual Sensors behaving like traditional resources. The machine learning runtime can be further integrated in the mashup architecture aiming to control the physical environment. To allow users easily defining control scenarios, we implemented a Web application fully integrating the fundamental concepts of the Management and Field Levels. Thanks to the powerful semantic descriptions of smart things, the application can automatically generate graphical blocks.

Finally, in the Intelligent Adaptation Level, we explore how the training part of machine learning can be integrated into the Web. Because current implementations are realised with dedicated data analysis software are far from field constraints such as CoAP, they cannot directly use our platform. We overcome this shortcoming through a set of proxies allowing a direct integration over HTTP. We then propose concepts automating the generation of control scenarios from the training. Moreover, we introduce a layer dedicated to the refinement of models that have generated actions perceived as inconvenient by users.

In the next chapter, we investigate ways to optimise our architecture in terms of network traffic and energy consumption. We also propose the use of transparent smart gateways to integrate the KNX and EnOcean legacy building networks, becoming first-class citizen of the WoB.

Chapter 5

Challenging the Web of Buildings Architecture

Contents

In Chapter 4 we presented our vision of an entirely Web-based middleware for smart buildings. While the building blocks and architectural guidelines provide a powerful ecosystem around smart things, this architecture can be challenged from several angles. We can for example mention security and large scale deployments among others. In this chapter, we focus on two aspects related to the Web of Buildings, which appear to us as the most needful relative to the requirements outlined in Section 2.5.

The first point is about interoperability with legacy building automation systems. We investigate the possibility to use smart gateways in order to expose components of smart building networks in our WoB. More specifically, we study two of the most used networks: KNX and EnOcean. Legacy devices become inherent components of the core of the WoB with no distinction to smart things, therefore opening the door for an integration in existing facilities.

Then, we question our architecture and especially the event-based model from an energy consumption point of view. By analysing the energy impact of the notification mechanisms, we will show that there are potentials to reduce the energy consumption of smart things in particular conditions. Moreover, we come with concrete proposition to enable reliable multicast notifications with CoAP, also having a beneficial impact on energy consumption.

5.1 Integrating Legacy Building Automation Systems in the Web of Buildings

In the Field Level of the Web of Buildings architecture discussed in Section 4.2, we outlined how sensors and actuators have to expose their functionalities in order to become part of the Web. However, a large number of buildings is currently equipped with legacy technologies using custom physical connections. Fundamentally critical in a Web context, they work over proprietary protocol stacks that do not carry IP packets. For this reason, legacy devices cannot naturally embed a RESTful Web server. To allow seamless communications between smart things and legacy systems, the legacy protocols need to be translated and the functionalities have to be made available over RESTful APIs.

In this section we propose to rely on smart gateways allowing to integrate transparently legacy protocols into the WoB as it was exposed in [11, 13, 10, 12]. We start by considering the KNX building automation system that is widespread in Europe. Then, we apply the same architecture to the EnOcean energy harvesting building automation system, gaining momentum in installations where wiring is an issue. We conclude this section by providing an exhaustive review of related work and by discussing our smart gateways.

5.1.1 KNX

In this section we outline a smart gateway architecture allowing to integrate legacy twisted pair KNX networks into the Web of Buildings over a KNXnet/IP bridge. We first discuss the specific KNX architectural components that have to be mapped to Web resources. Then, we show a mechanism to automatically configure the gateway from the configuration file of a KNX network. We then provide a reference architecture decomposed in modules to increase evolvability. Finally, we evaluate the performance of the gateway implementation in a real situation at the LESO-PB building of the EPFL.

Making KNX WoB-Compatible

In order to make KNX devices accessible from smart things using the WoB as application layer, specific features have to be mapped to the Web. More specifically, the smart gateway's role is to comply with the Field Level of our architecture, homogenising the access to functionalities over RESTful APIs.

Physical Interface The gateway needs to interface both worlds: IP and KNX. While the connection to an IP network can be achieved over various technologies such as Ethernet, Wi-Fi or 6LoWPAN, KNX is for its part more restrictive. A physical connection with the KNX twisted pair can be realised in two different ways:

TP/UART board: The TP/UART board forwards the packets to an UART serial interface using a dedicated protocol [355]. This module is especially intended for embedded systems such as micro-controllers. It is principally used by manufacturers developing new products.

KNXnet/IP bridge: KNXnet/IP is an application protocol developed by the KNX Association to transport KNX telegrams over IP networks [356]. The protocol defines a set of headers containing specific KNX data. Commercial bridges are proposed by manufacturers to interface the twisted pair through an Ethernet port. Tunnel connections can be created with the bridges to access the KNX network.

Although a gateway directly connected to the twisted pair allows to avoid the necessity of a bridge, we opted in the development of our gateway for the second solution. As it is not our goal to develop a commercial product, we privileged to lower the development time through the use of a KNXnet/IP bridge not requiring any skills in electronic.

Field Level Once having a physical connection with the KNX twisted pair, a RESTful API enabling access to the device functionalities has to be exposed. The gateway not only has to provide services in a RESTful manner, but also has to promote homogeneity through semantic descriptions allowing a mutual comprehension between different systems.

Each functionality will be associated with a resource that either accepts `GET`, `PUT` requests, or both. URLs will therefore identify a particular Group Object (GO) on the KNX side. For this purpose, resources will be mapped with KNX group object addresses. These addresses will be used to communicate with KNX devices. If applicable, the returned data will be provided in the payload of the CoAP response. By following this approach, the gateway is able to comply with the requirement to expose one resource for every functionality.

The WoB was designed to easily encompass features of legacy building automation systems as it works as a wrapper around the notion of datapoints (DPs). To achieve interoperability with other systems, the smart gateway has to provide semantic descriptions of the KNX devices. This is achieved by automatically composing descriptions from the KNX datapoints. As our model is very flexible and does not predefine data structures, every type of KNX datapoint can be transposed in a structure defined by our domain ontology. Therefore, only our shared semantic descriptions are used instead of the specific KNX datapoints. This contributes in hiding the legacy network.

From Group Objects to Resources

The KNX association has developed the *Engineering Tool Software* (ETS) to configure a KNX infrastructure. With this software, administrators and engineers have the possibility to create the building hierarchy, the network topology, and finally to create GOs representing functionalities between devices. At the time of writing this thesis, ETS is the most used tool for KNX configuration in professional installations. As shown in Figure 5.1, ETS exports projects in an archive composed of multiple XML files. The `knx_master.xml` file contains the description of all the DP types. The network topology, building organisation and group addresses are stored in the `0.xml` file. Finally, there is a folder for every manufacturer, containing a XML file for each device type composing the network. The device file informs about the available DPs on the device. This archive is zipped without security. It contains easily understandable XML files, allowing to import all the network knowledge into other applications. As illustrated in Figure 5.1, we apply a XSL transformation to the different XML files inside the archive in order to centralise all the necessary information for our gateway into one single XML file.

As previously explained, access to functionalities in a KNX network is done via GOs, which are compositions of datapoints and a group address. In the elaboration of the gateway, we match GOs to REST services allowing the interaction with KNX devices. For doing this, we use the XSLT output resulting from the XSL transformation. By taking into consideration the corresponding fields of the XML, we are able to compose a URL identifying a specific GO. For example, the datapoint shown in Listing 5.1 would result in the following URL: `coap://heating.office005.ground.leso.epfl.ch/dpt_switch`. The domain part is composed of the physical location of the device inside the building, completed by the domain name of the organisation. The last part of the URL representing the action to perform is the datapoint type

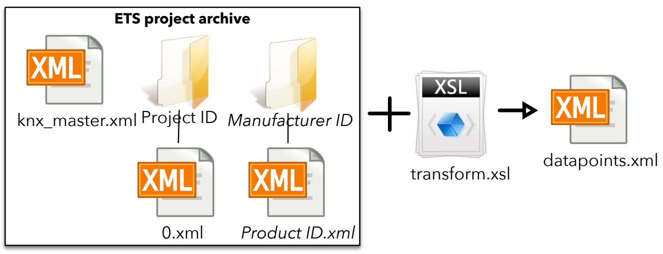


Figure 5.1: ETS project archive structure and XSL transformation.

name. We can now easily link group objects to URLs by following this generic rule: `coap://<group_name>.<location>.<organisation_domain>/<datapoint>`.

```
<datapoint stateBased="true" name="Heating" desc="Status" mainNumber="1"
    priority="Low" actionName="DPT_Switch" actionDesc="on/off" dptDesc="1-
    bit" dptBitsSize="1" location="Office005.ground.LESO">
    <knxAddress type="group">
        6195
    </knxAddress>
</datapoint>
```

Listing 5.1: Datapoint XML representation after the XSL transformation.

Our objective is here to allow smart things to pull state values and to perform actions on the KNX network by sending CoAP requests to the gateway. A `GET` request will result in the gateway reading the KNX group object and putting the actual value in the response. For changing a state, one will send a `PUT` request containing the new value inside the payload data. Representing the structural organisation of the building inside the domain part of the URL opens a new dimension. By acting this way, we can hide the fact that the device is actually in a KNX network and not directly connected to an IP network. For users of the system, the device seems to be an IP one with its own name entry directly pointing to it. However, this brings a certain complexity for the naming system as it musts contain entries matching with KNX groups. For example, the name equivalence of the group `heating` located in room `office005` of the `ground` floor in the `leso` building, giving the name entry *heating.office005.ground.leso* has to redirect requests to the gateway. This means that several name entries will point to the smart gateway. GOs can however be differentiated in the requests as the CoAP header provides the `Uri-Host` option, containing the name entry.

Architecture

Our architecture relies on the *Calimero 2.0* Java library [357]. This library provides Java classes and methods for KNXnet/IP tunnel communications, and datapoint object representation, allowing developers to build applications dedicated to KNX infrastructures. The Web part of our implementation is built around the JCoAP [337] library that was extended to support multicast. The database storing historical values is running on MySQL. All the technologies used in the implementation are open source and free.

As shown in Figure 5.2, we based our implementation on several logical modules shared in different scenarios of use. The first one is the configuration of the gateway, where the administrator will provide all the necessary information for proper running, such as the ETS archive that will be further processed. Once configured, the gateway enters in its normal operation where it can

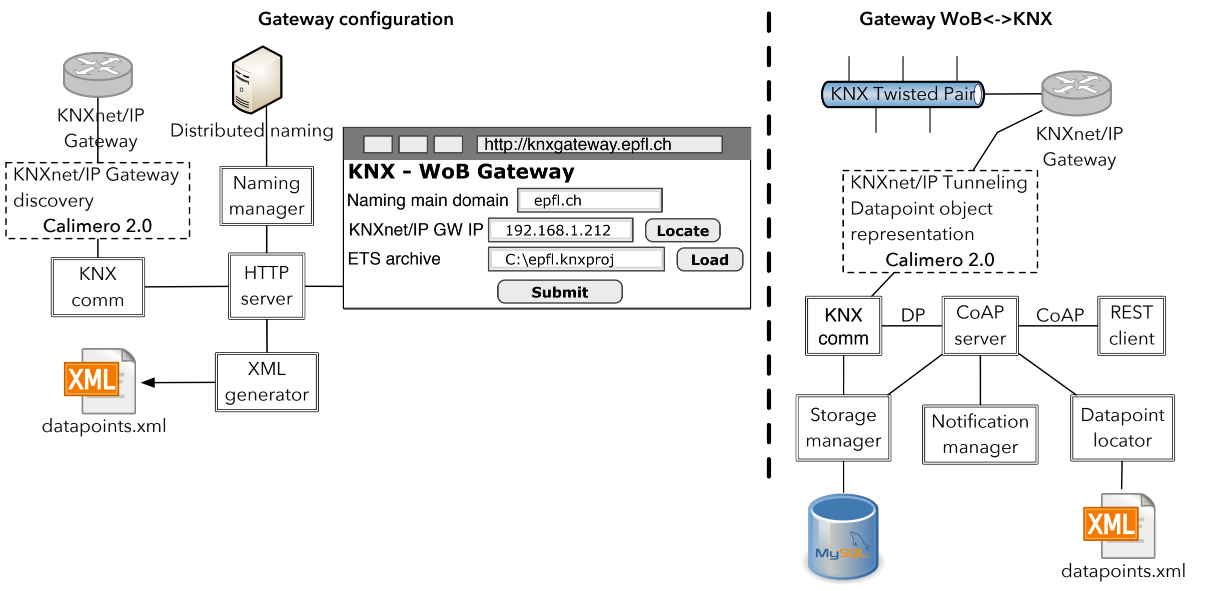


Figure 5.2: Overall KNX-WoB gateway architecture illustrating the logical modules for two scenarios: gateway configuration (left part) and gateway normal operation (right part).

serve requests to manipulate group objects. We provide here more details about the role of the different logical modules.

Datapoints file acts as database even if it only consists of XML tags. This file is at the heart of the application and holds all the mandatory information to communicate with the KNX devices. It contains a description of all the GOs reachable on the network, indicating the datapoint type, the group address, and other kind of data. This file also stores configuration data provided through the Web configuration page.

CoAP server stands as entry point of the application. It implements the *doGet()* and *doPut()* methods to handle the CoAP requests. The first step is to decode the URL in order to identify which action is requested on which GO. Once the operation identified, it acts as controller and dispatches the request to the right modules. At the end of the processing, it will respond either with a value corresponding to the `GET` read request, or only with the CoAP response code in the case of a `PUT`.

XML generator processes the ETS project archive to generate the XML datapoints file. It first decompresses the archive and then applies the XSL stylesheet to the project. All special characters are removed during this processing as they cannot be present in URLs. The XML file is saved inside the resource directory of the Web server.

Naming manager is responsible to add name record entries identifying functionalities. We reuse the approach of distributed naming system presented in Section 4.1.2. This module only implements the client specification and does not act as a server.

Datapoint locator acts as query engine for the datapoints file. It can look up specific GOs according to the datapoint type, group name and location, and then returns them in the Calimero datapoint object representation. It is also used to retrieve all the possible domain names corresponding to group objects, used by the Naming manager to add entries in the distributed naming system.

KNX comm represents an interface to the KNXnet/IP network. This module can discover KNXnet/IP bridge by sending multicast messages, thus avoiding administrators to look after the IP address of the bridge. This feature is useful when DHCP is used to configure the IP addresses. The KNX comm module handles the tunnel connection, allowing to talk with the KNX network. By listening to the network, it will notify the Notification manager of incoming telegrams that might concern some consumers. Activating a cache feature can avoid to overload the KNX network when too many clients try to read simultaneously the same GO.

Storage manager manages the storage of historical data produced by KNX devices. It participates in the overall cloud-like storage by acting as an agent providing storage according to our specification proposed in Section 4.2.3. It is however not mandatory for a smart gateway to provide internal data storage.

Notification manager stores in an associative table the clients having registered on resources. Triggered by the KNX comm module, it will lookup if consumers are registered for the GO having undergo a change of state, and then launches the notification by forwarding the new value.

Evaluation

To establish the performances of our gateway, we decided to measure various key-values such as: round-trip time for consecutive requests, maximum concurrent requests, notification reaction time (from the action on the KNX device until client notification) and processing time of the ETS project archive (during configuration). All our measurements were realised on an existing KNX installation of the four floors office LESO building located on the EPFL campus in Lausanne, Switzerland. The installation features 265 devices, distributed in 765 groups, with a total of 795 GOs and represents an average installation that can be found in many buildings.

We selected as hardware platform the Raspberry Pi which is low-cost and offers enough computional power to run our gateway. This module can be considered as low-power with a consumption of about 4W, powered at 5V over a micro USB port. The Raspberry Pi can be operated by many Linux systems installed on a SD card plugged into the device. Any kind of software compatible with the ARM processor can be installed on it, like Java, MySQL and many others.

Measure type	Result
ETS archive processing time	30 [min]
Maximum CoAP requests per second	45
Average round-trip time for 500 consecutive requests	22 [ms]
Maximum concurrent client requests	620
Average event reaction time for 500 notifications	33 [ms]

Table 5.1: Gateway performance measured on a real-life KNX installation running 265 devices.

Table 5.1 summarises the performance of the gateway implemented on a Raspberry Pi. The ETS archive processing time is quite long, mainly due to the XSLT that is extremely resource consuming. However, as this operation has only to be performed during the configuration of the gateway, we can assume that it is not an important issue. In order to avoid this long processing, we developed a tool in Java allowing to perform the XSLT on a computer and to deploy the resulting XML on the gateway. In this case, the processing of the ETS archive is much faster than performed on the gateway. We observed a processing time of 15 seconds on a regular PC.

The maximum CoAP requests per seconds is actually bottlenecked by the twisted pair of the KNX network offering only 9600b/s. The average response time for consecutive `GET` requests is for its part entirely satisfying although also limited by the twisted pair.

The maximum simultaneous CoAP requests is limited by the Raspberry Pi. Nonetheless, we believe that the measured value is largely sufficient for common building automation systems operation. Finally, we observed a fast event response time allowing to link KNX sensors with non-KNX actuators. We can see that all the results are suitable for such an installation and that the Raspberry Pi has to be considered as an alternative to classical PCs running gateways or serving as middleware.

5.1.2 EnOcean

In this section we propose an implementation of a smart gateway to connect EnOcean networks to the WoB through an USB dongle. Some parts of the implementations have been published in [13, 12]. We start by providing insights on how properties of EnOcean can be mapped to Web resources. Then, we expose two different modes of operation to integrate more or less transparently the gateway in a network. Due to the absence of management software, we explain how devices have to be added in the gateway. We then provide a modular reference architecture, and we end by evaluating the performance of the gateway in a real scenario.

Making EnOcean WoB-Compatible

The EnOcean application protocol is naturally not compatible with Web paradigms, and can therefore not be used by smart things. The gateway aims at translating the EnOcean data structures and functionalities into ones that are shared among WoB devices through RESTful APIs and semantic descriptions as defined in the Field Level.

Physical Interface In order to provide a Web compatibility, EnOcean functionalities have to be offered through an IP network. The specific EnOcean protocol stack has therefore to be interfaced with either Ethernet, Wi-Fi or 6LoWPAN. Two main ways to connect with the EnOcean RF network are available:

USB dongle: Several USB dongles are available on the market that can be plugged in a computer without requiring specific drivers. The dongle is detected as a serial port by the system and forwards telegrams between both mediums. The EnOcean Serial Protocol [358] (ESP) is used to format the telegrams that are exchanged through the dongle.

IP bridge: Like for KNX, some IP bridges are interconnecting the EnOcean RF and an Ethernet network. The main difference here is that no standard protocol has been defined how to forward telegrams over IP. This results in manufacturer-specific solutions which are not exchangeable and use proprietary stacks.

In our solution, we opted for the EnOcean USB300 dongle that is far cheaper than an IP bridge and can be directly connected to the gateway. Furthermore, the standard ESP allows the dongle to be exchangeable with others if required.

Unlike KNX where a single gateway is sufficient for an entire network, EnOcean requires the use of several ones depending on the building configuration. Working with radio frequency, the range only permits to cover a few rooms, even when using repeaters. Gateways have therefore to be spread in the building. Our experience with EnOcean shows that a gateway has to be installed every 15 meters in large buildings.

Field Level Once being able to communicate with the EnOcean network, functionalities have to be exposed as RESTful APIs. Here comes a first serious limitation of EnOcean. As the objective is to power the devices through energy harvesting, sensors are only equipped with a radio transmitter and sleep most of the time. This means that it is not possible to read their actual value. Services will therefore return to clients the last value transmitted by the sensor. Even more limiting, actuators endow only a radio receiver and cannot communicate their current state. It has to be guessed by observing the sensors driving them. Furthermore, observing the state of an actuator is difficult to achieve as there are no EnOcean Equipment Profiles (EEPs) defining their interface. More precisely, we cannot determine which measurement in a sensor telegram drives the actuator. Because of these lacks, our gateway can only be used to retrieve sensor information. Each time a sensors wakes up and sends a telegram, the values contained in it will be stored on the gateway, and will be considered as the current statutes.

Sensor functionalities can still be mapped to RESTful APIs accepting `GET` requests. We hereby rely on device properties such as name and location provided by users and the EEPs. The URL of a service identifies a particular device and a measure. As sensors can usually measure several physical properties or give insights about their internal states (e.g voltage of the internal battery), an URL has to point to what is called a shortcut in the EEPs. The last received value will then be returned in the payload of the response.

In order to make sensor measurements understandable by WoB devices, semantic descriptions of the functionalities and the data representations have to be provided. The WoB offers for this a flexible ontology to describe datapoints. Descriptions can be automatically generated from the EEP corresponding to a device. They contain all the necessary information and can be translated by using our domain ontology. By following this, EnOcean devices are hidden and appear as reusable resources that are understandable by all other smart things.

Modes of Operation

Due to the pairing principle of EnOcean, where sensors are learned on actuators, our gateway has to provide two modes of working: active and transparent. Those behaviours are shown in Figure 5.3. In the *active mode*, the gateway drives the actuators according to virtual groups created through a Web configuration application. This simplifies the physical pairing of devices by needing only to pair the gateway with the actuators. In this mode, the gateway will listen to telegrams sent by sensors, and if the sender of an incoming telegram is configured in active mode and is part of a group, the gateway will then retransmit the original telegram by replacing the sender ID with its own. The drawback of this mode is to add some latency and to weaken the whole area if the gateway fails (e.g. hardware problem, power outage, etc.). In such cases, users will no more be able to actuate (e.g switching lights, moving blinds, etc.) because of sensors being not directly paired with actuators.

In order to avoid those problems, we implemented a second mode of working, the *transparent mode*. This time, not only the gateway is paired with the actuators, but also the sensors. Both gateway and sensors can then drive the actuators. By having a direct link between sensors and actuators, we can provide a better user experience. As for the *active mode*, the gateway will store each captured telegram to provide sensor state values to clients.

Configuration Procedure

Mapping EnOcean capabilities is much more complicated than KNX, even if the configuration of the network is very simple by just pairing devices together. Indeed, there is no central

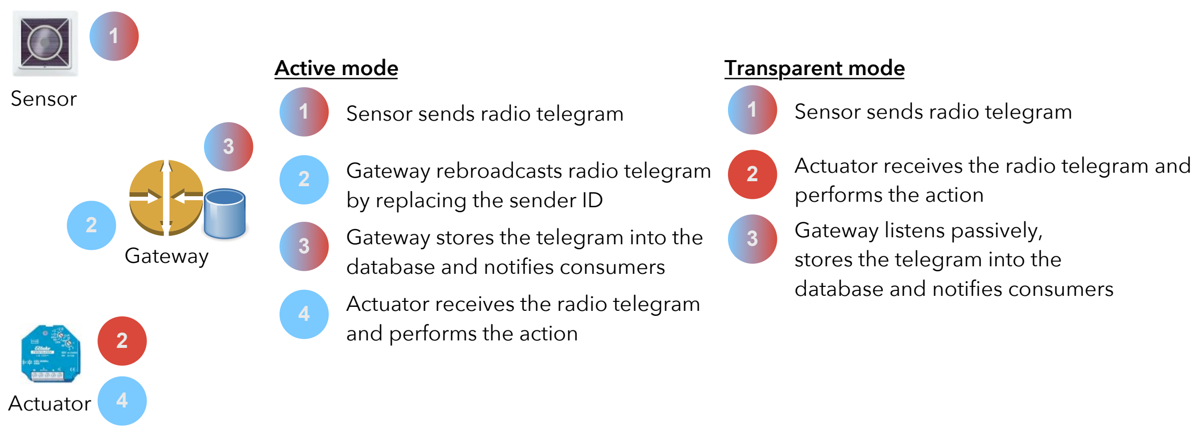

Sensor
1
Gateway
2
3
Actuator
2
4
Active mode
1 Sensor sends radio telegram
2 Gateway rebroadcasts radio telegram by replacing the sender ID
3 Gateway stores the telegram into the database and notifies consumers
4 Actuator receives the radio telegram and performs the action
Transparent mode
1 Sensor sends radio telegram
2 Actuator receives the radio telegram and performs the action
3 Gateway listens passively, stores the telegram into the database and notifies consumers

Figure 5.3: The EnOcean gateway can work according to two operation modes. In the *active mode*, actuators will only be paired with the gateway. In the *transparent mode*, sensors and the gateway will be paired with actuators.

management of the network like with ETS. Forming groups of functionalities by pairing devices is achieved by the user putting the actuator in a teach-in mode, and triggering a learn telegram on the sensor that has to drive the actuator. All the knowledge is distributed in the actuators and it exists unfortunately no way to retrieve it, because of the sensors sending broadcast telegrams and most actuators being only listeners.

The major drawback of this concept is the fact that it is therefore not possible to automate the mapping of the EnOcean network capabilities to REST APIs. An alternative is to let the user reproduce the pairing of devices by using a Web application hosted on the gateway, which is the approach taken in our implementation. The process can be simplified for the user by the gateway listening to teach-in telegrams. Those telegrams hold the EEP information, so that the gateway is now able to parse and to interpret the telegrams. The user can then add the detected devices to the gateway by furnishing additional information such as a name and the location. This information is then used to map URLs with shortcuts of EEPs by following this rule: `coap://<sensor_name>.<location>.<organisation_domain>/<shortcut>`. This allows clients to retrieve values of sensors by sending a CoAP `GET` request. However, as sensors are only senders and cannot listen to command telegrams, the gateway will return the last captured data.

Architecture

As at the time of implementing the gateway no library for the EnOcean Serial Protocol (ESP) was available, we developed our own. Our library is capable to capture incoming telegrams, to send outgoing ones, and to marshal and unmarshal the ESP. It is written in Java and can be ported to any platform running a Java Virtual Machine and having a USB dongle connected. Regarding the REST server, we reused the JCoAP library that was already used for the KNX gateway. The configuration, current and historical values are stored in a MySQL database.

The general architecture shown in Figure 5.4 is decomposed in logical modules either used during configuration or normal operation. During configuration, users manage the devices that have to be accessible through the gateway by providing basic information (name, location and operating mode). The gateway is able to automatically detect the type of sensor and its corresponding EEP by capturing the teach-in telegrams. Once configured, the gateway will start to catch telegrams and to store the values that will be returned to `GET` requests. We here provide more details on the modules that are different to the ones of the KNX gateway.

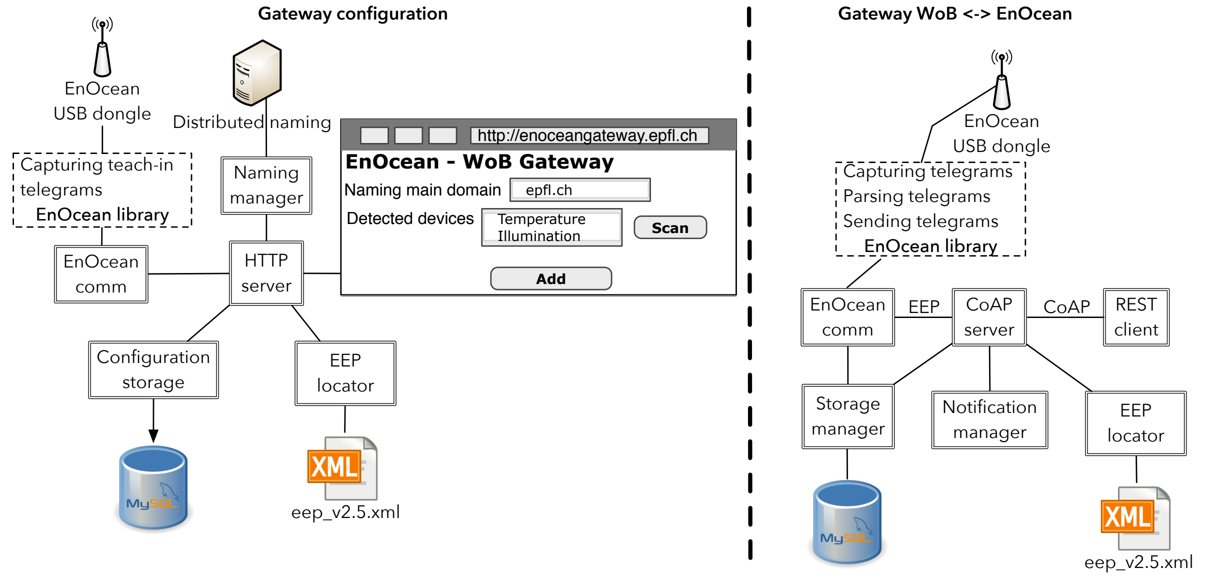


Figure 5.4: Overall EnOcean-WoB gateway architecture illustrating the logical modules for two scenarios: gateway configuration (left part) and gateway normal operation (right part).

EEP file is provided by the EnOcean Alliance and holds all the EEP descriptions in the XML format. Each EEP is detailed in terms of measures, their location within the telegram and the data properties (type, size and unit). These information serve to build the semantic description of the Web resources exposed by the RESTful API.

EEP locator is a lookup engine to parse the EEP XML file. It allows to list EEPs for a specific category of devices (e.g. a temperature sensor) so that users can choose the type of device they want to add. Furthermore, it allows to retrieve the properties of a particular EEP during operation to parse the telegrams.

EnOcean comm is the interface to the EnOcean network. It offers methods to send telegrams and watches for incoming ones. If the sensor having sent the telegram was previously configured in the gateway, then the values will be passed to the storage manager and will replace the current ones. Additionally, in the case of clients having registered as observers on a resource, it will forward the data to the notification manager.

Configuration storage is responsible to store and to manage the configuration of the gateway. This not only encompasses general properties, but is also essentially related to the EnOcean devices that must be accessible through the gateway. The necessary information such as their sender ID, EEP number, name and location are stored.

Evaluation

For the EnOcean gateway, we measured the same key-values as for KNX apart the ETS archive processing file. The gateway has been installed at the College of engineering, Fribourg, Switzerland, where an office room of 20m^2 has been equipped with 7 sensors, 3 lights/blinds switches and 4 actuators. This lab office was occupied by three people and represents a typical room that can be found in office buildings. The same hardware platform was used for the EnOcean gateway, namely a Raspberry Pi.

From Table 5.2, we can clearly see that all the results are also satisfying for EnOcean. All limitations are due to the Raspberry Pi reaching its limits of processing capabilities. Indeed,

Measure type	Result
Maximum CoAP requests per second	82
Average round-trip time for 500 consecutive requests	12 [ms]
Maximum concurrent client requests	631
Average event reaction time for 500 notifications	29 [ms]

Table 5.2: EnOcean gateway performance measured on a real-life installation running 14 devices.

no direct interaction with the EnOcean network is required as values are cached in the gateway. The event reaction time is sufficient to be used by other smart things such as Virtual sensors or in control scenarios.

5.1.3 Related Work

Trying to ease the development of applications using KNX devices has been explored in different works. A first attempt was realised with the BCU SDK [278], which consists of a script generating C++ classes representing devices capabilities. A more Web-oriented approach has been undertaken in [94]. The principle was to expose KNX functionalities as Web services by using the oBIX (Open Building Information Exchange) standard, which is a special XML schema to represent building data and operations. Unfortunately, oBIX is not at all widespread on smart things, probably because of its relatively complex XML schema. In addition to this, the proposed implementation does not allow an easy integration of the gateway in an existing environment, requiring an important configuration effort for large networks.

The openhab [359] project is a building management system offering interfaces to various building networks, including KNX. An integration of EnOcean is actually under development. Although being a complete solution, it does not allow to import a KNX configuration nor to expose stored data through REST services. Additionally, the project is quite complex and asks to have a deep knowledge of its components. Our approach tackles these limitations by taking advantage of the Web's simplicity and by being highly integrable in existing KNX and EnOcean infrastructures.

5.1.4 Discussion

In this section, we have described how the Field Level of the WoB proposed in Section 4.2 can be applied to legacy building automation systems through the use of smart gateways. Smart gateways allow to extend IP-based automation networks by encompassing legacy devices that have not the possibility to embed a Web server exposing a RESTful API. By allowing transparent interactions between different technologies, we have shown that buildings already equipped with legacy systems can be extended with novel technologies that are relying on the IP protocol. Gateways therefore contribute in the overall evolvability through the possibility to connect as well legacy technologies as new ones.

Legacy device functionalities can be seamlessly integrated in the core of the Web in a natural and efficient manner thanks to RESTful APIs. REST offers well-defined and self-describing mechanisms, which lower the overall system complexity and promote loose coupling. Such gateways can be implemented independently without influencing the general communication scheme. Clients reuse the same interaction style that is basing on URLs to identify resources that are located behind a gateway, as no difference is noticeable with native Web smart things.

The WoB and particularly the semantic description approach based on the domain ontology are generic enough to integrate various building automation systems. The datapoint and EEP defining the data format and properties of functionality endpoints can be translated in our shared ontology. The overall comprehension of the exchanged data and device capabilities is therefore homogenised amongst the different participants in a smart building. The specific particularities of technologies are hidden behind a generalised knowledge description model.

We have shown and experimented that any KNX device and functionality can be exposed as REST service. This is not the case for EnOcean, inducing several limitations. Because of the pairing mechanism and the objective to minimise the energy consumption of devices, only sensors can report their state. Devices being either deaf or silent, it is not possible to communicate with them, aside from listening to telegrams. From our point of view, we consider this a major conceptual mistake, as EnOcean devices cannot be integrated in building management systems. The EnOcean Alliance has realised that this is a considerable limitation and pushes new devices endowed with a transceiver, and therefore able to communicate bi-directionally. This has however a consequence on the self-powering ability as energy harvesting can no more satisfy the power requirements. Batteries or even power supplies have therefore to be connected to these devices to ensure their proper working.

Finally, both gateways have been successfully deployed in the LESO building on the EPFL campus. The KNX gateway allows students and researchers of the LESO to retrieve sensor values and to actuate devices with no particular KNX knowledge, only by performing REST requests. It is particularly used to test novel control algorithms. Four EnOcean gateways have been spread in the building in the context of a research focusing on the illumination.

5.2 Making the Web of Buildings More Energy Efficient

Building automation systems intend to reduce the energy consumption of buildings by exploiting sensor information and driving actuators. This requires to exchange data at relatively high rates between the components of the system. These communications have costs in terms of energy, as devices need to send data and network equipments have to forward it to the destination. From this observation, we can deduce that lowering the network traffic will also have a beneficial impact on energy consumption.

It is therefore meaningful that the automation system should not work against the overall goal by itself consuming a substantial amount of energy. The underlying system must obviously be optimised and must have the lowest possible footprint. As mentioned throughout this thesis, most communications are happening through the triggering of events themselves generating notifications. In this section, we propose two different mechanisms to reduce the impact of our Web of Buildings framework on energy consumption of devices by optimising notifications. We start by analysing the impact from an energy point of view of the main notification techniques presented in Section 3.3, namely CoAP observation, HTTP callback and WebSockets. We will use models computing the energy consumption of each technique to allow data producers selecting the most appropriate one. Then, in order to address several clients with a single message, we show how notifications can be sent over multicast with CoAP. For this purpose, we propose solutions combining observation and group communication, and this by keeping the CoAP reliability. We then conclude this section by providing an overview on previous research and by discussing our propositions.

5.2.1 Selecting the Appropriate Protocol

The protocols used for notifications have different structures and work either on top of TCP or UDP. This results in packets of different sizes that may have an impact on the energy consumption of devices. In order to verify this assumption, we will evaluate the notifications mechanisms to find out which one is the most adapted.

We would like to emphasise that the results and models proposed in this section are related to very specific conditions and are only applicable to Wi-Fi communications and to the OpenPicus Flyport. Using another Wi-Fi module will probably lead to different energy consumption results.

We start by performing measurements on a Wi-Fi module generating notifications in order to observe the impact of the protocols. We also provide mathematical models allowing to compute the energy consumption. Then, we propose an adaptation of the registration procedure so that the producer is able to choose the most appropriate notification technique. The energy consumption models will then be integrated in our hybrid REST library in the form of an optimisation module aiming to reduce the energy consumption of the device. The benefits of the module on the energy consumption are then evaluated. Finally, we discuss the advantages and drawbacks of our solution. The work presented in this section has been the subject of several publications [14, 15, 16].

From Measures to Consumption Models

In this section, we aim at quantifying the energy consumption of the CoAP observation, HTTP callback and WebSocket notification mechanisms. We start by outlining the experimental test bed that was set up to measure the energy consumption of the aforementioned mechanisms. Then, we analyse these results by identifying what are the characteristics of the protocols. Finally, we propose mathematical models allowing to compute the expected energy consumption for each notification method.

Procedure We set up an isolated test environment built of an OpenPicus Flyport module acting as the producer, a Wi-Fi access point, a PC acting as the consumer, a Hameg HM8115-2 programmable power meter to measure the energy consumption of the Flyport and a PC to record the measurements. The configuration of the access point was as follows: 802.11g, no encryption and long preamble. It is necessary to set up a dedicated test bench to ensure that no other device will be disturbing the proper running of the experiment, as it would be in a public network. All unnecessary services were deactivated on the participants of the network to guarantee that no other traffic can influence our measurements. The Flyport Wi-Fi module generated events that were sent to the consumer PC either through WebSocket, CoAP observation or by HTTP callback. The recorded measurements were then saved in a CSV format file to be further processed with Matlab.

To perform accurate measurements and to observe the influence on the power consumption of each protocol, only a minimal program sending events was running on the Flyport. Some pre-measurements showed that the energy consumption was lower when the module was supplied with 3.3V than 5V.

Our measurement campaign consists of tests of 30 seconds each, where the producer sends packets with a fixed payload size at a specific interval, applied to the three notification methods separately. Originally, we selected rather high intervals (from 1 to 5 seconds) between the generation of the events. However, it came out that no difference between the protocols in terms of energy consumption was visible. This imposed us to choose new reference values. We made the payload size in bytes vary as follows: 1, 10, 50, 100, 200 and 400, and the intervals in milliseconds between the sending of each notification as follows: 50, 100, 200, 400 and 800. The combination of the payload sizes and intervals gives us a campaign of 30 measurements. The 800 milliseconds interval limitation comes from the fact that it is not possible to observe significant differences of power consumption at higher intervals. This limitation is due to the insignificant repercussion on energy consumption at such high intervals. Nevertheless, we were able to observe a difference of consumption as WebSocket consumes more than HTTP and CoAP for intervals higher than 10 seconds. This difference is due to the keep-alive packets sent by TCP.

Power Consumption Measurements From the Figure 5.5 and the measurement tables in Section A.1, we can retrieve that HTTP is overall more energy consuming than WebSocket and CoAP. When examining the obtained measurements, WebSocket reveals to be on average 3.98% less consuming than HTTP, but only 0.69% less than CoAP. The maximal gain of WebSocket compared to HTTP is of 9.52% while the minimal one is 0.76%.

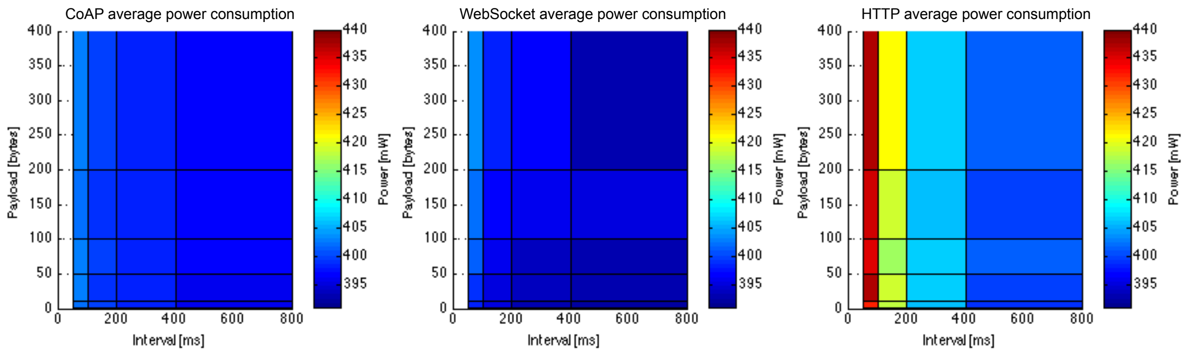


Figure 5.5: Measurements of the average power consumption for CoAP, WebSocket and HTTP notifications on the Flyport module.

CoAP performs almost as well as WebSocket, being in average 3.76% better than HTTP, with boundaries values of 8.70% and 0.76%. This can be explained by the number of packets exchanged, more precisely the total number of transmitted data. In the case of WebSocket, once the connection is established, only one packet is necessary to send the JSON payload. It is exactly the same for CoAP, except that no connection is required.

The communication is meanwhile more complex for HTTP. Every time a payload has to be sent, a connection has to be established first. This includes the inherent TCP window negotiation, the HTTP header, the HTTP response, and finally the connection closing. An increase of consumption is caused by this overhead. The amount of payload data does not play a major role in the power consumption. This is especially true when observing the results for HTTP and CoAP. The influencing factor of the consumption is mainly the sending interval.

Modelling The Energy Consumption To determine which method is the most efficient in particular conditions, we propose for this to build mathematical models allowing to compute the power consumption. As previously pointed out, the consumption depends on the payload and the sending interval. To achieve this, we have to find a function expressing the consumption, with the payload and the sending interval as parameters. We can calculate the time needed to send data, including all underlying protocols from the global composition of a 802.11g frame, taken from [279] and shown in Figure 5.6.

Approximation for WebSockets By relying on the composition of a TCP frame over Wi-Fi, we are able to define a formula to compute the energy consumption needed to send one packet of data.

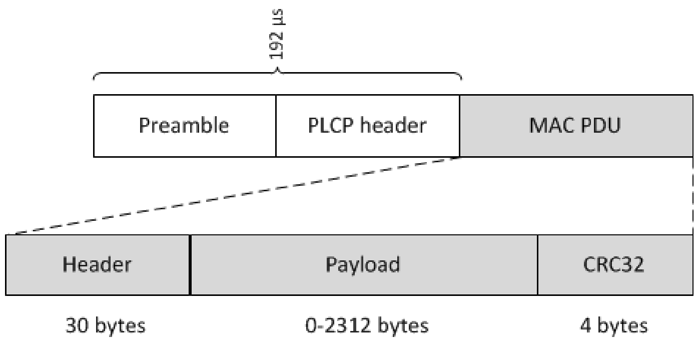


Figure 5.6: Composition of a 802.11 Wi-Fi frame taken from [279].

*E(payload) = (PLCP_preamble + (MAC_header + IP_header + TCP_header + WebSocket_header + payload) * ByteRate) * TransmitPower*

with IP_header, TCP_header and ByteRate known from [280, 281] and WebSocket_header from [152]. The transmission power was previously measured.

If we compare the energy consumption measurements with the theoretical values that can be computed with the approximation function, we can observe that the function is accurate enough to compute the energy consumption of the Flyport over WebSockets, with an average error of 0.86%.

Approximation for HTTP Computing the energy consumption for HTTP is more complicated. Due to the connection and window negotiation, due to several acknowledgements, and due to the connection closing, there are a lot more packets exchanged with HTTP than WebSockets. The model is even more complex to establish considering that the number of packets may vary from one connection to another. Instead of using a theoretical model, we opted for a parametric model where the parameters are fit to the observation. We converged to an exponential function which mathematical properties match at best with our measurements. The resulting function is as follows:

*P(interval) = a * exp(b * interval) + c * exp(d * interval)*

The parameters a , b , c and d were computed through a numerical fitting algorithm for every case of payload (1, 10, 50, 100, 200 and 400), ending up with 6 functions relative to the payload size. The resulting parameters allow the functions to be highly precise with an average error of 0.05% compared to the measured values.

Approximation for CoAP As for WebSockets, it is possible to approximate the consumption with a function taking as parameter the payload. Based on UDP, the header size is smaller than TCP [280]. Being optimised for constrained environments, the CoAP header consists of only a few bytes. The function is as follows:

*E (payload) = (PLCP_preamble + (MAC_header + IP_header + UDP_header + CoAP_header + payload) * ByteRate) * TransmitPower*

This function is also accurate enough to approximate the energy consumption of CoAP. If we confront the theoretical values we are able to compute to the measurements, we obtain an average error of 1.44%.

A Novel Notification Paradigm

In the current notification approaches, it is the role of the consumer to initiate the communication by registering itself as an observer. The producer will then send the new value each time the state changes by using the same protocol. While this procedure is obvious and can hardly change, there are some situations that question this behaviour. It could indeed happen that a client potentially supports several notification techniques. In this case, it randomly selects one that will be used until it does not require to be notified any more.

In the context of smart buildings, actuators and mashups will need during their entire operation to be informed about changes of sensors, as the event-based model is the major communication method. The notification mechanism used will therefore have an impact on the devices, and can lead to a relatively high footprint if chosen randomly. It is therefore necessary to select the most appropriate protocol. As previously seen, the energy consumption depends on the frequency of the notifications and the payload size. A client interested in being notified does not know these parameters in advance as they depend on the producer, making him unable to choose the appropriate mechanism.

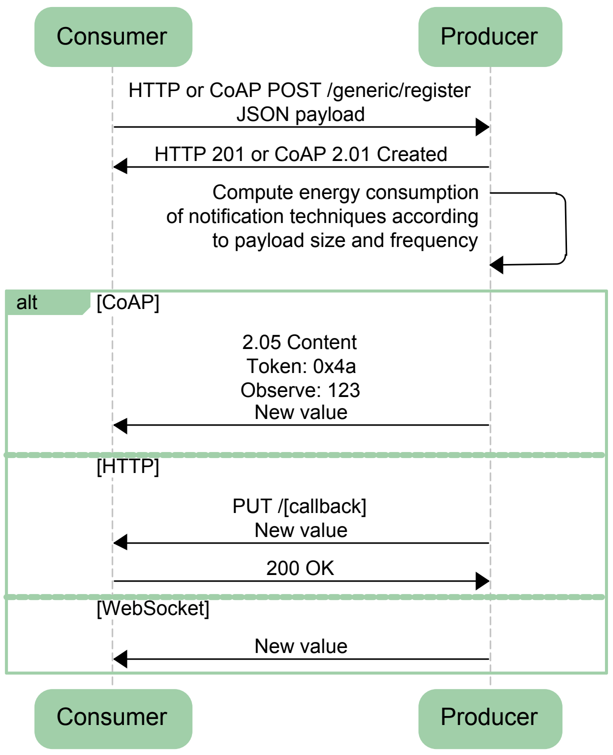

Consumer
Producer
HTTP or CoAP POST /generic/register
JSON payload
HTTP 201 or CoAP 2.01 Created
Compute energy consumption of notification techniques according to payload size and frequency
alt
[CoAP]
2.05 Content
Token: 0x4a
Observe: 123
New value
[HTTP]
PUT /[callback]
New value
200 OK
[WebSocket]
New value
Consumer
Producer

Figure 5.7: Clients register as consumers using a generic service that is either available over CoAP or HTTP. The notifications will then be sent according to the most appropriate notification mechanism.

We therefore propose to let the producer decide what notification mechanism has to be used, according to the frequency of notifications and the payload size. This requires however to adapt the registration procedure that has to be generic for all protocols. Notification producers supporting several notification methods will expose a specific resource dedicated to registration, accessible over CoAP and HTTP as shown in Figure 5.7.

Clients have to provide information about what notification methods they support and how they can be used during registration. In Listing 5.2 we show an example of JSON containing the necessary data. With this information, producers will be able to select the notification mechanism that has the lowest footprint in the current conditions.

```
{
   "websocket":{
      "port":26841,
      "key":"x3JJHMbDL1EzLkh9GBhXDw=="
   },
   "coap":{
      "port":91268,
      "token":12345
   },
   "http":{
      "callback":"htttp://consumer.home:8080/callback"
   }
}
```

Listing 5.2: Example of JSON furnished during the registration of a client supporting WebSocket, CoAP observation and HTTP callback.

The Selection Intelligence Module

To determine if an optimisation of the energy is achievable by choosing the best protocol, we have modified our hybrid REST server library presented in Section 4.1.1 to include the aforementioned models and concepts. The Core API now includes the *Selection Intelligence* module, as shown in Figure 5.8.

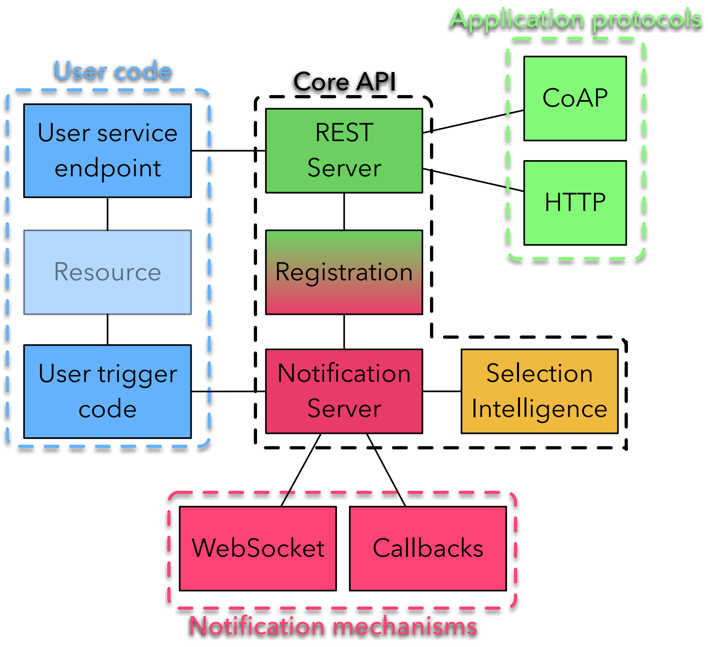


Figure 5.8: The Selection Intelligence module embeds the consumption models and is able to compute the energy of each notification technique. It is used by the Notification Server to decide which mechanism has to be used.

This module is responsible to select the most energy efficient protocol between HTTP, CoAP and WebSocket when sending notifications to clients. We implemented a straightforward technique to decide which protocol should be preferred. For this, we rely on a history of previous notifications specific for each resource that can generate events. Each time a notification is sent, a new entry containing the timestamp and the size of the payload will be added to the history. The module will use this history to compute a value indicating how much energy would have been used for

CoAP, HTTP and WebSocket to send all the previous notifications. The least consuming one will then be chosen to send the next notification. The selection process is summarised in Figure 5.9. We are aware that this approach is not optimal and is more reactive than proactive. It supposes that the future events have a relation with the past ones.

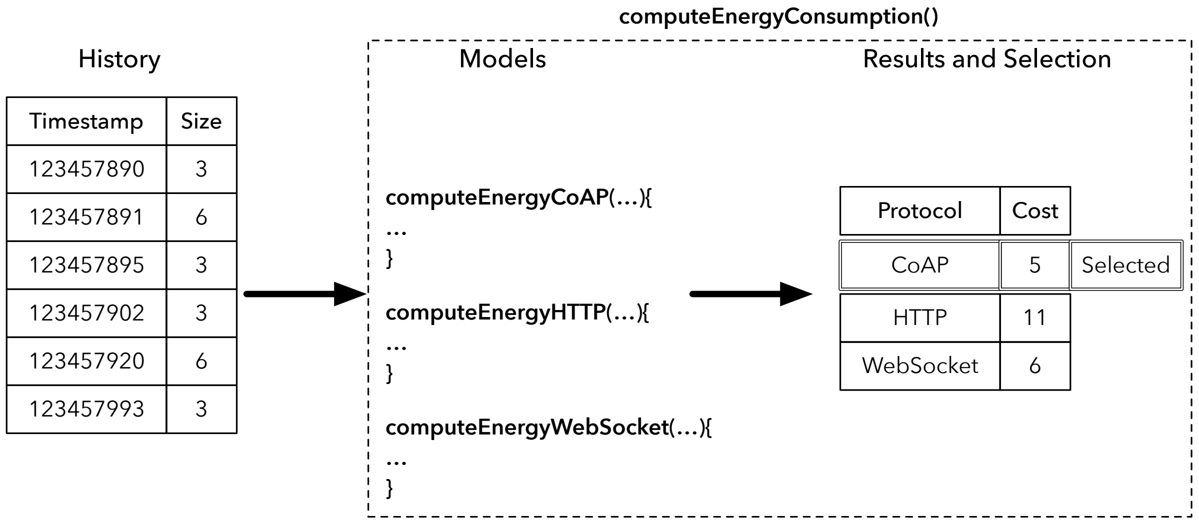


Figure 5.9: The history of previous notifications is used to compute the cost of each notification mechanism. The one with the lowest footprint will be selected for the next transmission.

The computation is done every time before some data has to be sent, and initiated by the Notification Server. This requires to know the costs in terms of energy to transmit bytes over the radio. Therefore, the module must include a consumption model specific to the device for each mechanism. These models can be derived from measurements or could in the future be directly provided by the manufacturer. The size of the history will play a major role on how the layer will respond to changes of the sending interval. A small history size will allow quickly responding to changes of interval, but could switch too fast for outlier values. At opposite, a large history will filter those outlier values, but will require a long time before switching.

Quantitative Evaluation

Table 5.3 shows the results we obtained after having performed the measurement campaign a second time with the Selection Intelligence module. The results are compared to the reference measurements when not running the intelligence module. We here remind that in Web of Things applications, HTTP and CoAP are often considered as mutually exclusive, so that a smart thing will prefer using CoAP than HTTP. This is the reason why in our evaluation we do not compare CoAP and HTTP together, but only with WebSocket.

The column *Gain* shows the percentage of energy saved relative to the highest value of the reference measurements. At opposite, the column *Loss* shows the percentage of energy lost relative to the lowest value of the reference measurements. A negative value in column *Gain* means the hybrid server with the Selection Intelligence consumes more than without. This can be explained by the consumption due to the module itself.

Nevertheless, the Selection Intelligence module has proven its usefulness by allowing to save 6.17% of energy in the best case and 2.10% on average when using WebSocket and HTTP. When using CoAP instead of HTTP, the results are much less impressive, because WebSocket and CoAP being very close in terms of consumption, leaving very little room for optimisation. The

Payload	Interval	WebSocket–HTTP		WebSocket–CoAP	
[bytes]	[ms]	Gain[%]	Loss[%]	Gain[%]	Loss[%]
1	50	5.41	0.25	0.00	0.00
1	100	3.45	0.49	0.00	0.00
1	200	1.49	0.25	0.00	0.00
1	400	0.00	0.25	0.00	0.00
1	800	0.00	0.50	0.00	0.00
10	50	6.17	0.00	0.00	0.00
10	100	4.20	0.00	0.25	0.00
10	200	1.73	0.25	0.25	0.00
10	400	0.50	0.25	0.50	0.00
10	800	-0.25	0.25	0.25	0.00
50	50	5.62	0.24	0.49	0.00
50	100	4.46	0.00	0.25	0.00
50	200	1.49	0.25	0.25	0.25
50	400	0.25	0.25	0.50	0.00
50	800	-0.25	0.50	0.00	0.00
100	50	5.39	0.25	0.00	0.00
100	100	4.46	0.25	0.50	0.00
100	200	2.24	0.00	0.00	0.25
100	400	0.50	0.25	0.25	0.00
100	800	-0.25	0.25	0.00	0.00
200	50	4.11	0.72	0.49	0.00
200	100	4.19	0.00	0.25	0.00
200	200	1.73	0.00	0.00	0.00
200	400	0.00	0.25	0.00	0.00
200	800	-0.25	0.25	0.25	0.00
400	50	2.63	0.72	0.48	0.00
400	100	2.67	0.49	0.00	0.00
400	200	0.98	0.00	0.25	0.00
400	400	0.25	0.00	0.00	0.00
400	800	0.00	0.50	0.00	0.00

Table 5.3: Power consumption efficiency of the Selection Intelligence module.

Selection Intelligence module also chooses the best method for higher intervals above 10 seconds as it selects HTTP or CoAP, which are theoretically the best ones.

Our energy savings being quite low for a single consumer, they become much more interesting with multiple ones. Indeed, in the case of multiple consumers registered (limited to three in our case due to memory limitations of the Flyport), the positive gains were almost multiplied by the number of consumers, up to 15% in the best case.

Discussion

Consequences on energy are seldom taken into account by developers implementing notifications. We provide here a Selection Intelligence module within the hybrid REST library doing the job at their place. Doing so, we hope to contribute to a reduction of the overall energy consumption of our WoB framework without using any duty-cycle or synchronisation technique. By relying on the impact of WebSocket, HTTP and CoAP on the Wi-Fi, we are able to reduce the energy consumption of event-based systems. Even if the purpose of those protocols is the same, i.e. carrying data, our measurements showed that they are indeed not equivalent in terms of energy footprint. However, this study pointed out that WebSocket and CoAP are almost identical.

Although our Selection Intelligence module allows energy savings for sensors sending at a fixed interval, its behaviour remains open for varying intervals. The reaction time of the module will be significantly influenced by the number of records stored in the history. The rate of symbols sent over Wi-Fi is another issue, as it is part of the approximation functions. This rate is continuously adapting itself to the surrounding environment. In our test infrastructure, it was forced to 2Mb/s. If the variations of the rate are too random, it could be that the gains of our proposed approach drops significantly.

Furthermore, savings are only reachable with notifications sent at a quite high frequency, the radio transmitter being not used otherwise. Although many building automation devices generate events at lower frequencies, we observed that some specific sensors, e.g. measuring properties of HVAC installations are working in the frequency range of this study.

At last, we would like to insist on the fact that, our research has not been deeply investigated, so that there is still room for improvements. One could save even more energy by caching and grouping events. Another way, instead of being only reactive, could be by predicting the behaviour of devices with self-learning algorithms, and to choose the method in a proactive manner. Finally, using multicast capabilities to notify more than one consumer with a single packet can also contribute in a reduction of the network traffic, as outlined in the next section.

5.2.2 Towards Multicast Notifications

In the Web of Buildings architecture, we use several times CoAP notifications [158] in combination with group communication [159] (multicast) to create distributed applications. Being able to notify several clients with one single message can contribute in a reduction of the network traffic. The current specifications of both techniques explicitly say that they are not mutually compatible. The reason for this conflict is due to the reliability mechanism that is part of CoAP observation.

The reliability is a technique similar to TCP acknowledgements ensuring the delivery of notification messages. It requires the producer to know explicitly to which consumer it is sending the notification, and is not the case with multicast that impersonates communications. In this section, we provide two solutions allowing multicast communications by keeping reliability. For this, we start by discussing the notion of multicast groups that have an influence on the architecture. Then, we outline our propositions that are tailored for two distinct use cases: autonomous and supervised group assignment. These propositions are further evaluated in terms of network traffic. Finally, we discuss the consequences of our architecture.

The propositions shown in this section are currently discussed at the IETF through the official mailing list and in meetings of the Core working group [360]. They will be the subject of IETF drafts in a near future. Moreover, we would like to put a particular attention on the fact that the achievable savings by using multicast principally depend on the type of network technology and topology (multicast routing protocols, mesh or star architecture).

Multicast Groups

Multicast groups are shared amongst network participants to exchange information that is destined to multiple recipients. A group is identified by an IP address and a destination port. No announcements are required in order to be part of a group. It is sufficient for nodes participating in group communication to listen to a particular or several groups. Individual responses to requests are always sent back in unicast.

In a context of notifications, a group serves to address multiple observers at one time. Instead of sending one CoAP response to each observer, only one will be sent to the multicast group. How groups are composed can influence the network load. We identify two kinds of group compositions:

Global groups are shared amongst all nodes of a network. Each potential observer has to listen to the predefined global group to receive notifications. This means that all notifications are sent to this unique group. Observers can identify the resource that generated the message through the token within the header. Although facilitating group management, the notifications have to be propagated to all nodes residing in the network. This can potentially cause some congestions for mesh networks. In this particular situation, the multicast becomes a broadcast. In addition, the messages will have to pass to the top of the application stack before they are potentially dismissed. This solution however has the advantage to limit the number of groups to one. Even constrained devices not able to manage multiple IP stacks due to memory limitations can use multicast notifications.

Resource groups are associated with a particular observed resource. This means that each resource supporting multicast observation owns its specific group. During registration, observers will be informed about the group they will have to listen to in order to receive the notifications. The token that is mandatory with observation plays no particular role as the observers can identify the resource through the group properties (IP address and port). This technique however requires observers to dynamically join multicast groups. In the case of a client observing several resources, it will have to listen to all the groups associated with the resources. This technique limits the number of potential compatible clients as some smart things are not able to join more than one multicast group due to IP stack and memory limitations. Combined with multicast routing, resource groups allow to concentrate the message forwarding to the only parts of the network where devices have joined the group, and therefore limits the traffic propagation.

In the remainder of this section, the aforementioned group composition approaches will be the subject of concrete proposals on how the CoAP protocol has to be adapted.

Combining Observation and Group Communication

The main reason for the incompatibility between observation and group communication is due to the impersonal communications. When a producer sends a notification over multicast, it does not know who are the potential observers. It is therefore unable to determine which producer has missed the notification. Nodes indeed do not announce themselves when joining a multicast group. We therefore in this section provide two different ways for observers to announce themselves to the producer, by adapting the CoAP protocol. Then, with this information, we show how reliability can be ensured.

Autonomous Group Assignment In this first solution, the group assignment is managed by the producer itself. The attributed multicast group can be either general or preferably resource-based. In Figure 5.10 we show the proposed registration procedure. The observer will issue as usual a `GET` request over unicast to the resource, with the *observe* property set accordingly. Two changes are required compared to the standard procedure. First, the observer does not provide a token in the request, as this will be generated by the producer. All the observers of an identical resource have indeed to share the same token. Secondly, the observer must provide its *NodeID* [282], which is a unique ID identifying itself. This last information is required to ensure reliability, in order to determine which observers have received a notification.

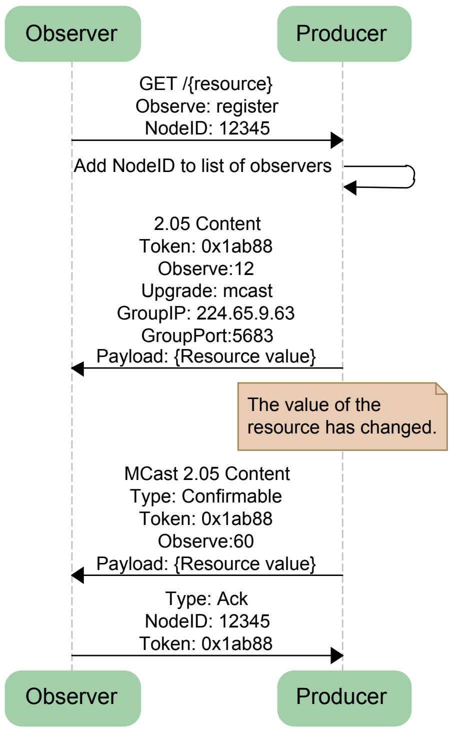


Figure 5.10: Observers register over unicast with the observable resource. Several additional options are required to handle the multicast negotiation.

The producer will as usual respond with a `2.05 Content` response that will however contain additional information. We introduce new headers that are used to inform the observer that from now on the notifications will be sent over multicast. First, the `Upgrade` option is similar to the one in HTTP for switching to WebSocket, and signifies that multicast will be used. Then, as the producer is imposing a multicast group, this information has to be shared with the observer. For this, the `GroupIP` and `GroupPort` options are included in the response.

After the registration process, notifications will be sent over multicast instead of unicast. The notifications can be either of non-confirmable type or requiring an acknowledgement from the observers. In the case of a confirmable notification, observers will respond with an acknowledgement containing their NodeID. The NodeID is here useful to determine which observers have missed the notification.

Although no more imposed by the client, the token however plays an important role. Its use is necessary to match notifications with resources, especially with general groups. With resource groups, the token could theoretically be omitted as the resource is identified by the multicast group. Nevertheless, as resource groups are generated randomly, it could happen that duplicated groups are used for different resources in a same network. In this case, the token will also allow to filter notifications.

Supervised Group Assignment In the supervised group assignment, we aim at providing a procedure that handles the management of the groups. This approach is used in the cloud-like data storage of the Field Level discussed in Section 4.2.3. It is intended for environments where a central entity (called the broker) organises the group assignment. The broker is in charge of linking producers with observers together. Like for the autonomous approach, either global or resource groups can be used. We focus only on the registration mechanism and assume that the broker already knows the observers and producers.

The general procedure is shown in Figure 5.11, where the broker first links the observer with the producer. This step is required so that the observer knows it has to consider notifications sent by a particular producer. To do so, observers have to expose a service dedicated to such assignment. The payload must contain the token that will be used to identify the resource and the group information (IP address and port). This procedure will be repeated for each observer concerned by the notifications of a resource.

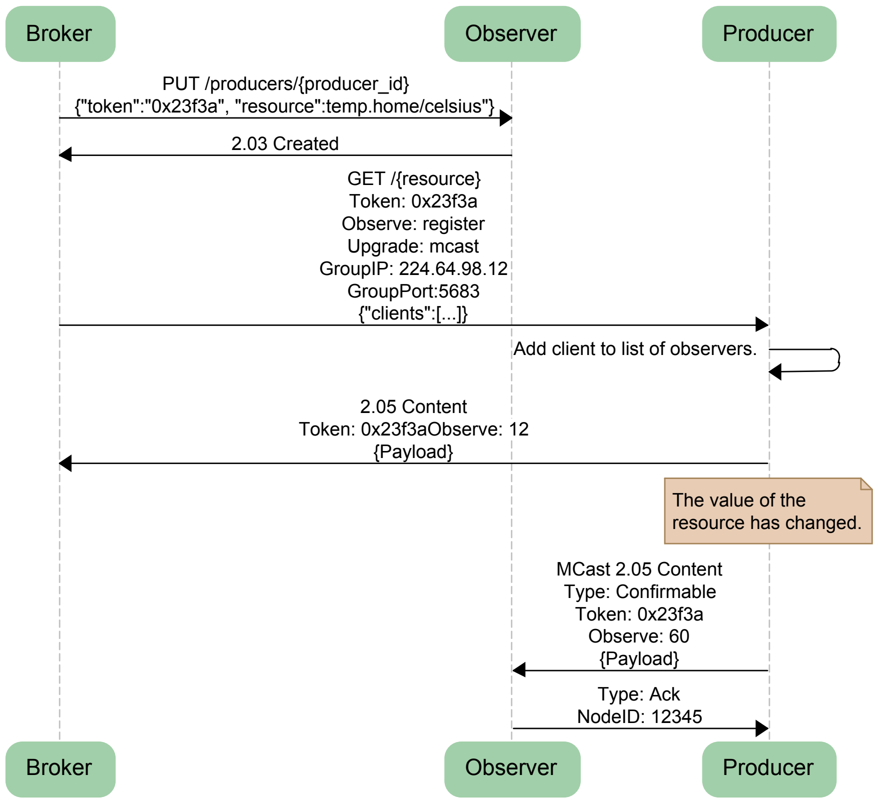


Figure 5.11: In a supervised environment, a broker will manage the group assignment between producers and consumers.

After having linked the observers with the resource through a randomly generated token, the broker will activate multicast notifications on the producer. The usual registration mechanism is slightly adapted to the context. While the token is provided by the broker (necessary to link the observers with the resource), some other information have to be enclosed in the request. First, the broker selects which multicast group (IP address and port) will be used to send the notifications, by reusing the new header options indicating an upgrade to multicast. Additionally, it will provide in the payload the list of observers in terms of NodeID. These NodeID will serve to determine if all clients have received a notification. Regarding the response, it does not include additional information compared to the standard observe procedure.

Once this procedure done, the producer will send the notifications over the multicast group imposed by the broker. Only smart things having been linked with that resource will consider the notifications. The token is once again used to identify a resource and to filter notifications.

Reliability Methods The previously proposed group assignment approaches are especially thought so that the producer knows who are the observers. This is a mandatory condition to ensure reliability. When notifications are sent with the confirmable flag, observers must respond with an acknowledgement containing their NodeID. The producer will use this information to check which observers have acknowledged and which have missed.

We here propose to rely on the same mechanism described in the traditional CoAP observation in terms of timeouts and retries. The *MessageID* for example is also used to identify repeated notifications. Observers having received the notification can discard the retransmissions by comparing the MessageID.

The use of multicast, however, raises questions in terms of how to forward the retransmissions. We identify two different ways how to send them:

Multicast: The simplest way to send retransmissions is over multicast. Observers having received the notification will discard the retransmission while the other ones having missed the original one will consider it. It is however questionable if a retransmission should be sent over multicast if only one or a few observers have not acknowledged. Multicast nevertheless remains the best alternative if a large number of observers have missed a notification.

Unicast: Sending retransmissions over unicast allows to address specifically observers having missed notifications. Moreover, it eliminates the potential forwarding to all smart things having joined the group. This however requires the producer to be aware of the observers' unicast IP address. Although a producer can keep track of unicast IP addresses of observers when receiving the acknowledgements, it can happen that an IP address has meanwhile changed.

From the above observations, we can argue that unicast retransmissions should be privileged for global multicast groups whenever possible. Unicast retransmissions have the advantage to address solely observers having missed the notification, and thus avoid the message to be forwarded to all observers throughout the entire network.

Quantitative Evaluation

To evaluate the impact of multicast notifications in a building automation context, we set up a test environment of six OpenPicus Flyport modules. Two of them were connected over Ethernet while the others were connected to a Wi-Fi network. One of the Wi-Fi modules was simulating a HVAC air flow sensor. The internal program of the sensor triggered an event randomly at an average interval of 100 milliseconds. The other five modules were acting as observers.

The experiment was run two times over 24 hours. In the first scenario, we used multicast notifications with the autonomous group assignment. In the second one, we used the classical unicast observation pattern. Reliability was enabled in both cases. The energy consumption of the producer was measured with a Hameg HM8115-2 programmable power meter.

	Multicast	Unicast
Energy consumption [kJ]	34.90	35.68
Number of messages	17280	86400
Success rate [%]	99.95	99.96
Number of retransmissions	8	33
Success rate after retransmissions [%]	100	100

Table 5.4: This table summarises the results from the 24 hours simulation using the classical CoAP observation and our proposition of multicast notifications.

Table 5.4 shows the differences between both approaches. We can observe that the number of messages is obviously five times higher when using unicast. Those numbers were expected as a separate notification is sent for each observer. The number of notifications received by the observers is substantially similar with only 0.01% difference. Note that with both notification methods, 100% of the messages were correctly received after retransmissions.

Regarding the energy consumption, the experiment showed a difference of 2.25% between both approaches. Using multicast allows to save some energy as less messages have to be sent. The radio is indeed less often in the sending mode and therefore lessens the energy consumption.

Discussion

The results we obtained show limited energy savings when sending notifications over multicast. More important savings can theoretically be achieved as the number of observers growths. We can observe a significant impact on the number of messages. With multicast notifications, only one message will be sent, whereas one for each observer is sent with unicast. This can contribute in avoiding network congestions. Indeed, the number of overall notifications (network traffic) will be much lower with multicast than unicast when considering a smart building with hundreds or thousands of producers.

We would like to emphasise that our evaluation was performed in specific conditions with Wi-Fi modules. The impact of multicast communications within a mesh network would depend on several factors such as the repartition of the nodes and the multicast routing protocol. However, multicast is an interesting alternative when it comes to notify multiple clients at the same time. Moreover, our two group-assignment techniques open the door for new applications in sensor networks, like our cloud-like in-network storage.

5.2.3 Related Work

The difference between SOAP, SOAP over HTTP and unstructured data has been analysed in [283] from an energy point of view. Their results show that unstructured data is the most efficient one, while SOAP over HTTP is the worst one. SOAP Web Services are compared to REST services in [284]. REST services come out as the winner in this comparison by being faster and having less overhead.

Close to our work, a comparison of HTTP and CoAP in terms of energy is performed in [184]. With notifications sent at 10 seconds interval over a 6LoWPAN network, they measured a difference of energy consumption of about 40%. This great difference is obtained by relying on a radio duty-cycling technique and is due to the fact that several packets must be sent over the radio for HTTP. Moreover, the energy consumption was only measured during the transmission of packets. Similar results are obtained in the same conditions in [285]. Their measurement approach is however questionable as it was measured over a short period of time, namely the packet transmission time, and does not give insights in real situations.

Yazar et al. [150] performed energy consumption measures of CoAP REST services by varying the returned payload. Although using 6LoWPAN modules, their results are similar to ours and show that the payload is not a key-parameter influencing energy consumption.

To the best of our survey, there are no literature works comparing unicast and multicast communications from an energetic point of view for notifications. However, many research have been realised, proposing efficient routing algorithms for multicast communications [286, 287, 288], which support our idea to use multicast communications in sensor networks.

5.2.4 Discussion

In this section, we proposed two techniques that aim at reducing the footprint of smart things on energy consumption and network traffic. We would like to remind that the results we obtained are relative to particular devices and technologies in specific conditions. Our work should be considered as preliminary studies paving the way for deeper research in this field. Furthermore, no duty-cycling technique turning off the radio was used in our approaches. This would require synchronisation methods and complicates the interaction between the devices.

We first analysed the differences in terms of energy between HTTP callback, WebSocket and CoAP observation. Our results show that HTTP should be avoided whenever possible as it consumes more energy than the others. The use of the wrong application protocol can lead to higher energy consumption. We then modelled the energy consumption of each technique to introduce an intelligent selection module that chooses the most appropriate notification method according to the history. Developers do not have to preconfigure the mechanism that will be used as this choice is led to the intelligence module. Thy can further concentrate on the application than the energy.

We then proposed to adapt the CoAP protocol so that observation and group communication are no more mutually exclusive, leading to multicast notifications. The autonomous group assignment technique allows devices to dynamically attribute multicast groups to resources, whereas this task is realised by a central node in the supervised approach. Although reducing the energy consumption of smart things by only 2.25%, this value must be relayed at a higher level where smart buildings are composed of thousands of devices. More important, multicast notifications are significantly reducing the amount of network traffic awarded to notifications. This technique can therefore be used in installations where sensors have to notify several clients simultaneously.

5.3 Summary

In this chapter we tackled the Web of Buildings architecture from two distinct points of view. First, we provided interoperability with legacy building automation systems by means of smart gateways exposing RESTfuls APIs. These interfaces allow to interact with legacy functionalities by following the guidelines of the WoB. This approach not only fosters interoperability, but lets existing installations seamlessly integrate into the Web. Furthermore, our KNX and EnOcean gateways were successfully installed and proved their suitability while reducing the effort required during the configuration phase.

Second, we addressed the energy consumption of smart things. Our experiments showed that applicative protocols are not equivalent in terms of energy footprint and that savings can be obtained by selecting the most appropriate one. In the same direction, we proposed concrete solutions to make CoAP ready for reliable multicast notifications by introducing new header options and slightly changing the registration procedure. All these techniques not only reduce the overall energy consumption, but also lower the network traffic. We would like to remind that the achievable savings depend on several parameters, which are, for some of them, little manageable. Future works should address these limitations by considering other network technologies such as 6LoWPAN.

Chapter 6

Conclusions and Outlook

Contents

With fossil fuels becoming increasingly scarce and expensive, and in a context where reducing our carbon footprint has become a major planetary goal, the reduction of buildings' energy consumption has gained focus. At the same time, buildings are offering more and more comfort with new types of automation and services. We believe that a better management of building facilities can largely contribute in limiting energy consumption while increasing user comfort.

In order to achieve such a combined management, we propose in this thesis work to rely on what we call the *Web of Buildings*. This framework is based on the adoption of proven Internet and Web technologies to accelerate the development and the integration of smart applications dedicated to building control. We believe that the advantages of the Web of Buildings are numerous:

Federating the technologies: IP and Web technologies are de-facto standards for communicating objects. Sensors and actuators installed in buildings are not left apart with the arrival of native IP devices. Thanks to their rich expressiveness, Web protocols are also good candidates to federate heterogeneous technologies often present in buildings.

Easing the development of applications: The use of lightweight application programming interfaces (APIs) as RESTful Web resources will allow developers to speed up their developments. Another benefit is the mashup pattern offered by such APIs, where services are exposed in a modular and reusable fashion, allowing to compose higher-level composite applications.

Lowering the barrier to entry: Web APIs allow to standardise the exposition of services. Patterns can be used to let a device describe its functionalities that can then be seamlessly consumed by machines or humans. The configuration of such smart things is then reduced, lowering the barrier to entry for applications, for example in houses or apartments.

The potentials of the Web of Buildings are large, actually following the trends of the Internet of Things and Web of Things. However, the context of buildings and the constraints of building automation systems are specific. The underlying objectives of this thesis were to analyse, to design and to evaluate IP and Web-based solutions in such a context, with respect to the specificities of smart buildings.

6.1 Contributions

The core of this thesis is the proposal of an architecture focusing on the homogenisation of the application layer of building automation systems. Our architecture is decomposed in four layers (shown in Figure 6.1) that we analysed and conceptualised by leveraging on Web technologies. The design of each layer was also the subject of proof-of-concept implementations that were then evaluated.

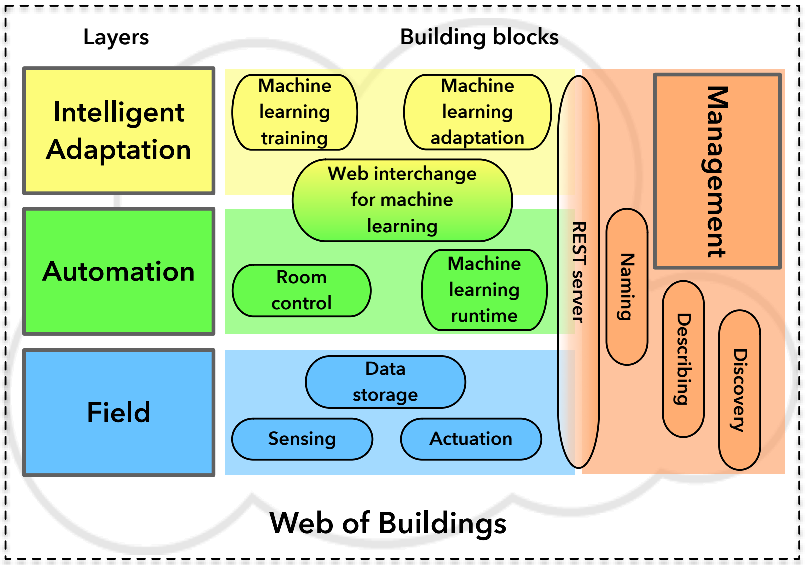


Figure 6.1: The Web of Buildings architecture.

1. **Management Level** In the Management Level we addressed the common features that are shared and reused by the other layers. We designed a hybrid REST server library facilitating the development of applications on smart things. Essential in a Web environment, we showed how a hierarchical naming system can be distributed in an autonomous manner. Further, we employed the semantic Web to provide an extensible domain ontology describing functionalities. We also introduced the notion of *gates* allowing to automate service consumption. Along with the powerful languages and concepts of the semantic Web, we were able to provide a distributed query system to discover devices spread in the building network.

2. **Field Level** In the Field Level we proposed guidelines outlining how functionalities of sensors and actuators have to be exposed as RESTful APIs. Furthermore, we demonstrated how interoperability between devices of different technologies and manufacturers can be achieved by relying on URIs, services and semantic descriptions. In order to limit the propagation of historical data, we presented the conception and evaluation of a cloud-like storage reusing the memory of smart things.

3. **Automation Level** In the Automation Level we focused on the control logic of buildings from a novel angle. By including machine learning and especially generative algorithms such as Hidden Markov Models as core components of the Web, we opened the path to execute data-driven control approaches in a simple way. The MaLeX format contributes to this achievement by providing a standardised way to deploy mathematical models. Our concept of *Virtual Sensor* makes abstraction of machine learning specificities and exposes the runtime as a common sensor. Additionally, we provided a mashup Web application

that seamlessly integrates into the WoB, able to deploy control scenarios on compatible smart things.

4. **Intelligent Adaptation Level** In the Intelligent Adaptation Level we contributed to integrate the machine learning training process into the Web through the proposal of an architectural platform based on proxies. We then showed an approach to improve the accuracy of models by encompassing user feedback. For this, we came up with a distributed Web architecture distinguishing actions performed by users from those issued by the control logic.

In the thesis, we also performed deeper incursions in two important challenges that the Web of Buildings technologies will face: compatibility with legacy systems and energy efficiency.

Compatibility with Legacy Systems We proposed the use of smart gateways for interfacing with legacy building automation systems and the WoB. We provided two concrete implementations seamlessly integrating KNX and EnOcean into the Web. Device functionalities are automatically mapped to RESTful APIs along with the mandatory semantic descriptions. Smart things can therefore transparently use functionalities residing on non-IP networks, which fosters reusability.

Energy Efficiency We first studied the impact of notifications on the energy consumption of smart things and applied a technique selecting the most appropriate protocol. Then, we suggested two different approaches to merge CoAP group communication and observation, leading to the ability to notify clients simultaneously over multicast. Both techniques allow to reduce the overall network traffic load and also have a beneficial impact on energy consumption.

For each building block, we evaluated the implementation first in an empirical manner by means of several prototypes and applications, then in quantitative terms with performance studies. Overall, the results demonstrated that the WoB is scalable enough to support thousands of devices while limiting its footprint on network traffic and energy consumption. More important, our architecture significantly simplifies the integration of heterogeneous technologies by federating them around Web paradigms and especially the REST architectural style. Additionally, our ecosystem contributes in lowering the development time of applications by relying on open and widespread standards.

6.2 Discussion and Future Challenges

This thesis should be considered as following an exploratory approach striving to integrate building automation systems in an homogeneous Web architecture. Instead of focusing on a particular shortcoming, we took a wider angle regarding an overall integration including all underlying infrastructures, and tried to understand and experience the related implications. As a consequence, this thesis provides an holistic view of this emerging domain, but also emphasises several challenges that will face the realisation of the Web of Buildings.

Although the work presented in this thesis provides answers to the questions that were driving our research, several new issues were raised:

Enhancing Security The building-blocks presented throughout this thesis do not address security nor privacy. These issues are however crucial in a context where the access to physical objects and the possibility to change their state are feasible. Although techniques such as DTLS can be employed to secure communications, no general security concept that tackles specificities of distributed sensor applications has been proposed by the Internet of Things community. Fortunately, this lack has been recognised and is now subject to discussions in standardisation groups such as the IETF.

Smart buildings are particularly challenging from a security point of view as the management has to be kept as simple as possible. For example, maintaining access lists on every device is very probably too cumbersome. Moreover, because of the real-time aspect of interactions between sensors and actuators, the security layer should not penalise the communications by introducing perceptible delays.

Deploying the Web of Buildings We gave a particular attention to test the proposed concepts by implementing concrete prototypes and to evaluate them in real-life scenarios whenever possible. However, we were not able to test our contributions in large-sized buildings except for the KNX and EnOcean gateways. The other building-blocks were tested in a lab environment installed with only a few dozen of devices. It is therefore obvious that we are missing feedback regarding the behaviour of our implementations in large-scale deployments. However, the vision behind the Web of Buildings is to provide an architecture adapting itself to the infrastructure in a self-organised way. We therefore believe that the concepts of zone grouping according to the room structure allow to bear a large amount of devices, as shown by our simulations.

Future work should focus on testing our concepts and associated technologies on larger deployments. New challenges will probably emerge or may point out difficulties for the scalability of the Web architecture applied to buildings. The fact of using open technologies should also increase the interest of potential test sites or system manufacturers.

Towards Semantic Unity In this thesis we relied on the semantic Web to provide understandable and machine-readable descriptions enabling automatic service discovery and consumption. While the highlighted technologies of the semantic Web, such as RDF and SPARQL, are standardised and thus understandable by any other machine, it is quite different for the knowledge base. Domain knowledge is centralised in ontologies that must be shared to achieve a mutual understanding. While this does not represent a limitation in closed systems, it can be penalising in architectures aiming to include new systems or participants, like it is the case for the Web of Buildings. It can happen that systems which should be interoperable have not the same domain knowledge. This heterogeneity at the knowledge level therefore restrains evolvability and interoperability.

In our approach we tried to limit this shortcoming by composing a domain ontology from already existing knowledge bases. This however does not guarantee total interoperability if other systems do not reuse common ontologies. A major effort by the semantic community is required to concentrate knowledge in ontologies that would be considered as de-facto standard. We can cite as example HTML which is standardised and maintained by the World Wide Web Consortium (W3C), resulting in every browser understanding the same tags. A central knowledge base around building automation systems should be created by following this example. Interoperability between devices of different manufacturers and even automation technologies could be a benefit for end-customers that could replace or expand their installation with vendor-neutral systems.

Distributing Machine Learning A novelty in this thesis is to put machine learning at the core of the Web. As machine learning is gaining momentum in smart building applications, its integration as key component in architectures becomes unavoidable. We voluntarily focused on generative algorithms which are the most common ones. While they provide good results from a classification point of view, they suffer from requiring large amounts of memory and computational power during the training phase. As it was demonstrated in this thesis, the test phase (runtime) can be distributed on smart things. Its integration into the Web with the notion of *Virtual Sensor* exposing RESTful APIs opens new ways for deploying higher-level types of sensors.

The main limitation comes from the necessity to process historical data in one batch during the training, that can obviously not happen on smart things. In a vision of a cloud-like building network requiring no dedicated infrastructure, this last point prevents from deploying totally autonomous systems reusing available resources. A particular effort must be provided on generative algorithms in order to distribute the training process in a same manner as for the test phase. Once this realised, the path will be open to fully include machine learning as core component of the Web, enabling new types of applications.

Final Thoughts This thesis illustrates how Web technologies in smart buildings are beneficial to homogenise the application level, resulting in comprehensible and reusable entities. We point out how the REST architectural style can be applied to all levels, so that the entire ecosystem speaks the same language, including underlying services. Thanks to the semantic Web and its related concepts and languages (e.g. RDF, ontologies and SPARQL), smart things are able to autonomously discover each other through understandable and machine-readable descriptions, opening the door for automatic service composition. Moreover, buildings are an interesting playground for self-organising systems as they already provide a natural structural organisation around the notion of their own morphology (e.g. site, building, floor, room, etc.). Although we paved the way to integrate more intelligence in the Web and in buildings through the deployments of *Virtual Sensors*, machine learning needs still to be subject of greater attention by the scientific community from a Web point of view.

In this thesis, we demonstrate that the Web of Buildings is interesting not only because it promotes the same basic and interoperable standards and paradigms, but also because it significantly eases the development of composite applications such as control systems through the provision of useful building blocks: discovery, naming, storage, etc. Through the presented architecture, sensors, actuators and higher-level services become first-class citizen of the Web, which fosters reusability and evolvability. As a consequence, we believe that the Web of Buildings has great potential to generate momentum in the field of building automation by inspiring innovation, leading to an increasing number of interesting applications mixing smart things, legacy systems and machine learning.

Appendix A

Energy Consumption Measures

A.1 Raw Energy Consumption of the Protocols

Payload[bytes]	Interval[ms]	Min[mW]	Max[mW]	Avg[mW]	Med[mW]
1	50	391	411	402	402
1	100	398	406	400	400
1	200	394	401	398	398
1	400	390	399	396	396
1	800	388	399	396	396
10	50	396	405	403	403
10	100	397	400	399	399
10	200	395	400	398	397
10	400	394	399	396	396
10	800	386	399	394	394
50	50	400	407	403	403
50	100	391	405	400	400
50	200	391	403	398	398
50	400	385	401	397	397
50	800	384	398	395	395
100	50	392	405	402	402
100	100	395	404	399	399
100	200	389	404	398	398
100	400	388	401	397	397
100	800	392	399	396	396
200	50	395	410	403	403
200	100	397	406	400	400
200	200	391	401	398	398
200	400	390	404	397	397
200	800	389	413	396	396
400	50	404	419	408	407
400	100	394	442	404	404
400	200	392	437	399	398
400	400	389	417	397	396
400	800	386	403	396	396

Table A.1: Energy consumption measures for the CoAP observation notification mechanism.

Payload[bytes]	**Interval[ms]**	**Min[mW]**	**Max[mW]**	**Avg[mW]**	**Med[mW]**
1	50	408	456	434	432
1	100	406	434	418	417
1	200	391	423	406	407
1	400	385	424	399	394
1	800	381	416	396	393
10	50	425	457	437	435
10	100	409	448	419	417
10	200	394	421	407	405
10	400	385	426	400	395
10	800	383	431	395	392
50	50	418	451	435	434
50	100	407	432	417	415
50	200	396	423	407	406
50	400	386	427	401	396
50	800	372	442	395	392
100	50	386	462	436	435
100	100	410	435	419	418
100	200	386	429	406	405
100	400	385	433	400	397
100	800	378	426	396	393
200	50	423	458	437	437
200	100	399	447	421	418
200	200	396	428	407	406
200	400	385	433	401	398
200	800	386	434	397	393
400	50	418	468	440	439
400	100	411	445	426	423
400	200	394	440	410	411
400	400	384	436	402	396
400	800	386	449	397	392

Table A.2: Energy consumption measures for the HTTP callback notification mechanism.

Payload[bytes]	Interval[ms]	Min[mW]	Max[mW]	Avg[mW]	Med[mW]
1	50	381	407	398	398
1	100	391	401	395	395
1	200	386	399	393	393
1	400	389	402	392	392
1	800	380	397	391	391
10	50	395	406	399	399
10	100	384	405	395	395
10	200	391	398	394	394
10	400	389	398	393	392
10	800	380	398	392	392
50	50	398	409	401	400
50	100	392	401	396	396
50	200	390	399	394	394
50	400	381	399	393	393
50	800	386	396	392	392
100	50	393	411	402	401
100	100	388	407	397	396
100	200	384	404	396	395
100	400	392	400	395	394
100	800	382	399	393	392
200	50	395	420	404	403
200	100	393	418	398	396
200	200	378	408	397	397
200	400	383	402	393	392
200	800	380	400	392	392
400	50	393	425	407	405
400	100	394	430	403	401
400	200	394	415	399	398
400	400	384	416	395	394
400	800	388	416	394	392

Table A.3: Energy consumption measures for the WebSocket notification mechanism.

A.2 Energy Consumption Measures with the Selection Intelligence Module

Payload[bytes]	Interval[ms]	WebSocket[mW]	HTTP[mW]	Intelligence[mW]	Gain[%]	Loss[%]
1	50	406	429	407	**5.41**	**0.25**
1	100	404	420	406	**3.45**	**0.49**
1	200	402	409	403	**1.49**	**0.25**
1	400	402	403	403	**0.00**	**0.25**
1	800	402	400	402	**0.00**	**0.50**
10	50	405	430	405	**6.17**	**0.00**
10	100	405	422	405	**4.20**	**0.00**
10	200	403	411	404	**1.73**	**0.25**
10	400	400	403	401	**0.50**	**0.25**
10	800	401	401	402	**-0.25**	**0.25**
50	50	408	432	409	**5.62**	**0.24**
50	100	404	422	404	**4.46**	**0.00**
50	200	402	409	403	**1.49**	**0.25**
50	400	401	403	402	**0.25**	**0.25**
50	800	401	400	402	**-0.25**	**0.50**
100	50	407	430	408	**5.39**	**0.25**
100	100	403	422	404	**4.46**	**0.25**
100	200	402	411	402	**2.24**	**0.00**
100	400	401	404	402	**0.50**	**0.25**
100	800	401	401	402	**-0.25**	**0.25**
200	50	411	431	414	**4.11**	**0.72**
200	100	406	423	406	**4.19**	**0.00**
200	200	404	411	404	**1.73**	**0.00**
200	400	402	403	403	**0.00**	**0.25**
200	800	401	401	402	**-0.25**	**0.25**
400	50	415	429	418	**2.63**	**0.72**
400	100	410	423	412	**2.67**	**0.49**
400	200	407	411	407	**0.98**	**0.00**
400	400	403	404	403	**0.25**	**0.00**
400	800	402	400	402	**0.00**	**0.50**

Table A.4: Energy consumption measures comparing WebSocket, HTTP and the Selection Intelligence.

Payload[bytes]	Interval[ms]	WebSocket[mW]	CoAP[mW]	Intelligence[mW]	Gain[%]	Loss[%]
1	50	406	406	406	0.00	0.00
1	100	404	404	404	0.00	0.00
1	200	402	402	402	0.00	0.00
1	400	402	402	402	0.00	0.00
1	800	402	402	402	0.00	0.00
10	50	405	405	405	0.00	0.00
10	100	405	404	404	0.25	0.00
10	200	403	402	402	0.25	0.00
10	400	400	402	400	0.50	0.00
10	800	401	401	400	0.25	-0.25
50	50	408	407	406	0.49	-0.25
50	100	404	405	404	0.25	0.00
50	200	402	404	403	0.25	0.25
50	400	401	403	401	0.50	0.00
50	800	401	401	401	0.00	0.00
100	50	407	407	407	0.00	0.00
100	100	403	405	403	0.50	0.00
100	200	402	403	403	0.00	0.25
100	400	401	402	401	0.25	0.00
100	800	401	401	401	0.00	0.00
200	50	411	409	409	0.49	0.00
200	100	406	405	405	0.25	0.00
200	200	404	404	404	0.00	0.00
200	400	402	402	402	0.00	0.00
200	800	401	402	401	0.25	0.00
400	50	415	417	415	0.48	0.00
400	100	410	410	410	0.00	0.00
400	200	407	406	406	0.25	0.00
400	400	403	403	403	0.00	0.00
400	800	402	402	402	0.00	0.00

Table A.5: Energy consumption measures comparing WebSocket, CoAP and the Selection Intelligence.

Publications List

[1] G. Bovet and J. Hennebert, “Le web des objets à la conquête des bâtiments intelligents,” *Bulletin Electrosuisse*, vol. 10s, pp. 15–18, 2012.

[2] G. Bovet and J. Hennebert, “Will web technologies impact on building automation systems architecture?,” in *Proceedings of the 5th International Conference on Ambient Systems, Networks and Technologies (ANT 2014), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2014)*, pp. 985–990, 2014.

[3] G. Bovet, A. Ridi, and J. Hennebert, “Toward web enhanced building automation systems,” in *Big Data and Internet of Things: A Roadmap for Smart Environments*, Studies in Computational Intelligence, pp. 259–283, Springer International Publishing, 2014.

[4] G. Bovet and J. Hennebert, “Distributed semantic discovery for web-of-things enabled smart buildings,” in *New Technologies, Mobility and Security (NTMS), 2014 6th International Conference on*, pp. 1–5, March 2014.

[5] C. Gisler, G. Barchi, G. Bovet, E. Mugellini, and J. Hennebert, “Demonstration of a monitoring lamp to visualize the energy consumption in houses,” in *The 10th International Conference on Pervasive Computing (Pervasive)*, June 2012.

[6] G. Bovet and J. Hennebert, “A distributed web-based naming system for smart buildings,” in *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks, WoWMoM 2014*, pp. 1–6, 2014.

[7] G. Bovet, G. Briard, and J. Hennebert, “A scalable cloud storage for sensor networks,” in *Proc. of the 5th International Workshop on Web of Things*, WoT’ 14, pp. 22–27, ACM, 2014.

[8] G. Bovet, A. Ridi, and J. Hennebert, “Appliance recognition on internet of things devices,” in *To appear in Proc. of the 4th IEEE International Conference on the Internet of Things (IoT 2014)*, 2014.

[9] G. Bovet, A. Ridi, and J. Hennebert, “Virtual things for machine learning applications,” in *Proc. of the 5th International Workshop on Web of Things*, WoT’ 14, pp. 4–9, ACM, 2014.

[10] G. Bovet and J. Hennebert, “A web-of-things gateway for knx networks,” in *Smart Objects, Systems and Technologies (SmartSysTech), Proceedings of 2013 European Conference on*, pp. 1–8, 2013.

[11] G. Bovet and J. Hennebert, “Introducing the web-of-things in building automation: A gateway for knx installations,” in *ICINCO 2013 - Proceedings of the 10th International Conference on Informatics in Control, Automation and Robotics*, vol. 1, pp. 99–106, 2013.

[12] G. Bovet and J. Hennebert, “Web-of-things gateways for knx and enocean networks,” in *Cleantech for Smart Cities and Buildings from Nano to Urban Scale (CISBAT 2013)*, vol. 1, pp. 519–524, 2013.

[13] G. Bovet and J. Hennebert, “Offering web-of-things connectivity to building networks,” in *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Pub*, UbiComp ’13 Adjunct, pp. 1555–1564, ACM, 2013.

[14] G. Bovet and J. Hennebert, “Communicating with things - an energy consumption analysis,” in *The 10th International Conference on Pervasive Computing (Pervasive)*, June 2012.

[15] G. Bovet and J. Hennebert, “An energy efficient layer for event-based communications in web-of-things frameworks,” in *Multimedia and Ubiquitous Engineering, MUE 2013*, vol. 240 of *Lecture Notes in Electrical Engineering*, pp. 93–101, Springer Netherlands, 2013.

[16] G. Bovet and J. Hennebert, “Energy-efficient optimization layer for event-based communications on wi-fi things,” in *Proceedings of the 4th International Conference on Ambient Systems, Networks and Technologies (ANT 2013)*, pp. 256–264, 2013.

[17] A. Ridi, N. Zarkadis, G. Bovet, N. Morel, and J. Hennebert, “Towards reliable stochastic data-driven models applied to the energy saving in buildings,” in *Cleantech for Smart Cities and Buildings from Nano to Urban Scale (CISBAT 2013)*, vol. 1, pp. 501–506, 2013.

References

[18] B. Wesselink, R. Harmsen, and E. Wolfgang, “Energy Savings 2020: How to triple the impact of energy saving policies in Europe - A contributing study to Roadmap 2050,” tech. rep., ROADMAP 2050, 2010.

[19] L. Pérez-Lombard, J. Ortiz, and C. Pout, “A review on buildings energy consumption information,” *Energy and Buildings*, vol. 40, pp. 394–398, Jan. 2008.

[20] J. Zhu, D. A. S. Chew, S. Lv, and W. Wu, “Optimization method for building envelope design to minimize carbon emissions of building operational energy consumption using orthogonal experimental design (OED),” *Habitat International*, vol. 37, pp. 148–154, 2013.

[21] F. Mateo, J. J. Carrasco, A. Sellami, M. Millán-Giraldo, M. Domínguez, I. Díaz, and E. Soria-Olivas, “Forecasting Techniques for Energy Optimization in Buildings,” in *Encyclopedia of Information Science and Technology*, pp. 967–977, IGI Global, 2014.

[22] A. Dounis and C. Caraiscos, “Advanced control systems engineering for energy and comfort management in a building environment—A review,” *Renewable and Sustainable Energy Reviews*, vol. 13, pp. 1246–1261, Aug. 2009.

[23] KNX Association, *KNX System Specifications*. KNX Association, february ed., 2011.

[24] J. Hertel, “LONWORKS in industrial,” *ETFA 2001. 8th International Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No.01TH8597)*, 2001.

[25] F. I. Ferreira, A. L. I. Osório, C. S. . G. S. P. Pedro, and J. a. M. F. I. Calado, “Building automation interoperability – A review,” in *IWSSIP 2010 - 17th International Conference on Systems, Signals and Image Processing*, pp. 158–161, 2010.

[26] H. Doukas, K. D. Patlitzianas, K. Iatropoulos, and J. Psarras, “Intelligent building energy management system using rule sets,” *Building and Environment*, vol. 42, pp. 3562–3569, 2007.

[27] S. Wang and J. Xie, “Integrating Building Management System and facilities management on the Internet,” *Automation in Construction*, vol. 11, pp. 707–715, Oct. 2002.

[28] W. Kastner, G. Neugschwandtner, S. Soucek, and H. M. Newman, “Communication Systems for Building Automation and Control,” *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1178–1203, 2005.

[29] W. Granzer, “Gateway-free integration of BACnet and KNX using multi-protocol devices,” *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, 2008.

[30] J.-P. Vasseur and A. Dunkels, *Interconnecting Smart Objects with IP*. Elsevier, 2010.

[31] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, and E. Jansen, "The Gator tech smart house: A programmable pervasive space," *Computer*, vol. 38, pp. 50–60, 2005.

[32] E. Fleisch and F. Mattern, *Das Internet der Dinge*, vol. 2007. Springer-Verlag Berlin Heidelberg, 2005.

[33] F. Mattern and C. Floerkemeier, "From the internet of computers to the internet of things," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6462 LNCS, pp. 242–259, 2010.

[34] N. Gershenfeld, R. Krikorian, and D. Cohen, "Internet 0 : Interdevice Internetworking," *Scientific American*, no. October, pp. 1–10, 2004.

[35] M. Kovatsch, M. Weiss, and D. Guinard, "Embedding Internet Technology for Home Automation," in *Proceedings of the 2010 IEEE Conference on Emerging Technologies and Factory Automation ETFA*, vol. 33, pp. 463–72, 2010.

[36] A. Kamilaris, V. Trifa, and A. Pitsillides, "The Smart Home meets the Web of Things," *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 7, no. 3/2011, pp. 145–154, 2010.

[37] L. Filipponi, A. Vitaletti, G. Landi, V. Memeo, G. Laura, and P. Pucci, "Smart City: An Event Driven Architecture for Monitoring Public Spaces with Heterogeneous Sensors," *2010 Fourth International Conference on Sensor Technologies and Applications*, pp. 281–286, July 2010.

[38] J. W. Hui and D. E. Culler, "Extending IP to low-power, wireless personal area networks," *IEEE Internet Computing*, vol. 12, pp. 37–45, 2008.

[39] J. W. Hui and D. E. Culler, "IP is dead, long live IP for wireless sensor networks," in *Proceedings of the 6th ACM conference on Embedded network sensor systems - SenSys '08*, p. 15, 2008.

[40] L. M. S. De Souza, P. Spiess, D. Guinard, M. Köhler, S. Karnouskos, and D. Savio, "SOCRADES: A Web service based shop floor integration infrastructure," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4952 LNCS, pp. 50–67, 2008.

[41] T. Riedel, N. Fantana, A. Genaid, D. Yordanov, H. R. Schmidtke, and M. Beigl, "Using web service gateways and code generation for sustainable IoT system development," in *2010 Internet of Things, IoT 2010*, 2010.

[42] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications.* Springer-Verlag Berlin Heidelberg, 1 ed., 2004.

[43] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. 'big'web services: making the right architectural decision," in *Proceeding of the 17th international conference on World Wide Web*, pp. 805–814, 2008.

[44] B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao, "Tiny web services for sensor device interoperability," in *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, pp. 253–266, 2008.

[45] F. Jammes, A. Mensch, and H. Smit, "Service-oriented device communications using the devices profile for Web services," in *Proceedings - 21st International Conference on Advanced Information Networking and Applications Workshops/Symposia, AINAW'07*, vol. 2, pp. 947–955, 2007.

[46] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, "Interacting with the SOA-based internet of things: Discovery, query, selection, and on-demand provisioning of web services," *IEEE Transactions on Services Computing*, vol. 3, pp. 223–235, 2010.

[47] D. Yazar and A. Dunkels, "Efficient application integration in IP-based sensor networks," *Proceedings of the First ACM Workshop on Embedded Sensing Systems for EnergyEfficiency in Buildings BuildSys 09*, p. 43, 2009.

[48] D. Guinard, *A Web of Things Application Architecture – Integrating the Real-World into the Web.* Ph.d., ETH Zurich, 2011.

[49] D. Guinard, V. Trifa, and E. Wilde, "A resource oriented architecture for the Web of Things," *Internet of Things (IOT), 2010*, 2010.

[50] D. Guinard and V. Trifa, "Towards the Web of Things: Web Mashups for Embedded Devices," in *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences)*, vol. 51, pp. 1506–1518, ACM Press, 2009.

[51] D. Guinard, V. Trifa, F. Mattern, and E. Wilde, "From the Internet of Things to the Web of Things : Resource Oriented Architecture and Best Practices," *Architecting the Internet of Things (2011)*, pp. 97–129, 2011.

[52] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[53] S. Vinoski, "RESTful web services development checklist," *IEEE Internet Computing*, vol. 12, no. 6, p. 440, 2008.

[54] E. Finch, "Is IP everywhere the way ahead for building automation ?," *Facilities*, vol. 19, no. 11, pp. 396–403, 2001.

[55] Z. Shelby and C. Bormann, *6LoWPAN: The Wireless Embedded Internet.* WILEY, 2009 ed., 2009.

[56] G. Kortuem, F. Kawsar, D. Fitton, and V. Sundramoorthy, "Smart Objects as Building Blocks for the Internet of Things," *IEEE Computer Society*, vol. 10, pp. 1089–7801, 2010.

[57] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, pp. 1645–1660, Sept. 2013.

[58] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," *Computer Networks*, vol. 52, pp. 2292–2330, 2008.

[59] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: A survey," *Computer Networks*, vol. 38, pp. 393–422, 2002.

[60] M. A. Kafi, Y. Challal, D. Djenouri, M. Doudou, A. Bouabdallah, and N. Badache, "A study of wireless sensor networks for urban traffic monitoring: Applications and architectures," in *Procedia Computer Science*, vol. 19, pp. 617–626, 2013.

[61] Z. Fan, P. Kulkarni, S. Gormus, C. Efthymiou, G. Kalogridis, M. Sooriyabandara, Z. Zhu, S. Lambotharan, and W. H. Chin, "Smart grid communications: Overview of research challenges, solutions, and standardization activities," *IEEE Communications Surveys and Tutorials*, vol. 15, pp. 21–38, 2013.

[62] H. Alemdar and C. Ersoy, "Wireless sensor networks for healthcare: A survey," *Computer Networks*, vol. 54, pp. 2688–2710, 2010.

[63] M. S. Familiar, J. F. Martínez, and L. López, "Pervasive Smart Spaces and Environments: A Service-Oriented Middleware Architecture for Wireless Ad Hoc and Sensor Networks," *International Journal of Distributed Sensor Networks*, vol. 2012, pp. 1–11, 2012.

[64] J. Byun and S. Park, "Development of a self-adapting intelligent system for building energy saving and context-aware smart services," *IEEE Transactions on Consumer Electronics*, vol. 57, pp. 90–98, Feb. 2011.

[65] A. Giridhar and P. R. Kumar, "Toward a theory of in-network computation in wireless sensor networks," *IEEE Communications Magazine*, vol. 44, pp. 98–107, 2006.

[66] R. V. Kulkarni, A. Forster, and G. K. Venayagamoorthy, "Computational Intelligence in Wireless Sensor Networks: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 13, pp. 68–96, 2011.

[67] W. Li, J. Bao, and W. Shen, "Collaborative wireless sensor networks: A survey," in *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*, pp. 2614–2619, 2011.

[68] M. Satyanarayanan, P. Bahl, R. Cáceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, pp. 14–23, 2009.

[69] M. Satyanarayanan, Z. Chen, K. Ha, and W. Hu, "Cloudlets: at the Leading Edge of Mobile-Cloud Convergence," in *Proceedings of MobiCASE 2014: Sixth International Conference on Mobile Computing, Applications and Services*, 2014.

[70] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and Its Role in the Internet of Things," *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pp. 13–16, 2012.

[71] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Generation Computer Systems*, vol. 29, pp. 84–106, Jan. 2013.

[72] P. Middleton, P. Kjeldsen, Tully, and Jim, "Forecast: The Internet of Things, Worldwide, 2013," tech. rep., Gartner, 2013.

[73] L. Mainetti, L. Patrono, and A. Vilei, "Evolution of wireless sensor networks towards the internet of things: A survey," in *Software, Telecommunications and Computer Networks (SoftCOM), 2011 19th International Conference on*, pp. 2–7, IEEE, 2011.

[74] S. Sojoudi and S. H. Low, "Optimal charging of plug-in hybrid electric vehicles in smart grids," *2011 IEEE Power and Energy Society General Meeting*, pp. 1–6, 2011.

[75] K. Balasubramanian and A. Cellatoglu, "Improvements in home automation strategies for designing apparatus for efficient smart home," *IEEE Transactions on Consumer Electronics*, vol. 54, pp. 1681–1687, 2008.

[76] L. Lu, W. Cai, Y. S. Chai, and L. Xie, "Global optimization for overall HVAC systems—Part I problem formulation and analysis," *Energy Conversion and Management*, vol. 46, pp. 999–1014, May 2005.

[77] H. Han, Y. Jeon, S. Lim, W. Kim, and K. Chen, "New developments in illumination, heating and cooling technologies for energy-efficient buildings," *Energy*, vol. 35, pp. 2647–2653, June 2010.

[78] T. Zhu, A. Mishra, D. Irwin, N. Sharma, P. Shenoy, and D. Towsley, "The case for efficient renewable energy management in smart homes," *Proceedings of the Third ACM Workshop*

on Embedded Sensing Systems for Energy-Efficiency in Buildings - BuildSys '11, p. 67, 2011.

[79] R. Z. Freire, G. H. Oliveira, and N. Mendes, "Predictive controllers for thermal comfort optimization and energy savings," *Energy and Buildings*, vol. 40, pp. 1353–1365, Jan. 2008.

[80] Z. Wang, R. Yang, and L. Wang, "Multi-agent control system with intelligent optimization for smart and energy-efficient buildings," in *IECON Proceedings (Industrial Electronics Conference)*, pp. 1144–1149, 2010.

[81] L. Wang, Z. Wang, and R. Yang, "Intelligent multiagent control system for energy and comfort management in smart and sustainable buildings," *IEEE Transactions on Smart Grid*, vol. 3, pp. 605–617, 2012.

[82] S. Biffl, A. Schatten, and A. Zoitl, "Integration of heterogeneous engineering environments for the automation systems lifecycle," in *IEEE International Conference on Industrial Informatics (INDIN)*, pp. 576–581, 2009.

[83] Y. Yang, Q. Zhu, M. Maasoumy, and A. Sangiovanni-Vincentelli, "Development of Building Automation and Control Systems," *IEEE Design & Test of Computers*, vol. 29, pp. 45–55, 2012.

[84] T. Salsbury, "A survey of control technologies in the building automation industry," *16th IFAC World Congress*, vol. 16, p. 1396, 2005.

[85] S. Prívara, J. Široký, L. Ferkl, and J. Cigler, "Model predictive control of a building heating system: The first experience," *Energy and Buildings*, vol. 43, pp. 564–572, Feb. 2011.

[86] Y. Ma, F. Borrelli, B. Hencey, B. Coffey, S. Bengea, and P. Haves, "Model Predictive Control for the Operation of Building Cooling Systems," *Control Systems Technology, IEEE Transactions on*, vol. 20, pp. 796–803, 2012.

[87] A. G. Ruzzelli, C. Nicolas, A. Schoofs, and G. M. P. O'Hare, "Real-time recognition and profiling of appliances through a single electricity sensor," in *SECON 2010 - 2010 7th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, 2010.

[88] A. Ridi, C. Gisler, and J. Hennebert, "Automatic identification of electrical appliances using smart plugs," in *2013 8th International Workshop on Systems, Signal Processing and Their Applications, WoSSPA 2013*, pp. 301–305, 2013.

[89] T. V. Kasteren, A. Noulas, G. Englebienne, and B. Kr, "Accurate Activity Recognition in a Home Setting," in *UbiComp '08 Proceedings of the 10th international conference on Ubiquitous computing*, pp. 1–9, 2008.

[90] K. Murphy, *Machine Learning: a Probabilistic Perspective.* MIT Press, 2012.

[91] P. Domingos, "A few useful things to know about machine learning," *Communications of the ACM*, vol. 55, p. 78, 2012.

[92] I. O. for Standardization, "Iso 16484-2:2004 building automation and control systems (BACS)," tech. rep., ISO, 2004.

[93] KNX Association, *KNX Advanced Course Specification.* KNX Association, february ed., 2012.

[94] M. Neugschwandtner, G. Neugschwandtner, and W. Kastner, "Web services in building automation: Mapping KNX to oBIX," in *IEEE International Conference on Industrial Informatics (INDIN)*, vol. 1, pp. 87–92, 2007.

[95] EnOcean, “EnOcean Radio Protocol V1.0,” tech. rep., EnOcean Alliance, 2012.

[96] EnOcean Alliance, “EnOcean Equipment Profiles V2.5,” tech. rep., EnOcean Alliance, 2013.

[97] ASHRAE, “Standard 135-2010 - BACnet A Data Communication Protocol for Building Automation and Control Networks (ANSI Approved),” tech. rep., ANSI/ASHRA, 2010.

[98] ISO, “Building automation and control systems – Part 5: Data communication protocol,” tech. rep., International Organization for Standardization, 2003.

[99] S. T. Bushby, “BACnet^{TM}: A standard communication infrastructure for intelligent buildings,” *Automation in Construction*, vol. 6, pp. 529–540, 1997.

[100] BACnet, “Introduction to BACnet - For Building Owners and Engineers,” tech. rep., BACnet International, 2014.

[101] L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: A survey,” *Computer Networks*, vol. 54, pp. 2787–2805, 2010.

[102] E. Fleisch, “What is the Internet of Things?,” *Economics, Management, and Financial Markets*, vol. 2, pp. 125–157, 2010.

[103] ITU, “The internet of things,” tech. rep., ITU Internet Reports - Executive Summary, 2005.

[104] M. Alahmad, W. Nader, Y. Cho, J. Shi, and J. Neal, “Integrating physical and virtual environments to conserve energy in buildings,” *Energy and Buildings*, vol. 43, pp. 3710–3717, Dec. 2011.

[105] S. Kyriazakos and F. Nieto, “Scenarios and Applications in a Things as a Service Environment,” in *Wireless Personal Multimedia Communications (WPMC), 2013 16th International Symposium on*, pp. 10–12, 2013.

[106] G. Fortino, a. Guerrieri, G. O’Hare, and a. Ruzzelli, “A flexible building management framework based on wireless sensor and actuator networks,” *Journal of Network and Computer Applications*, vol. 35, pp. 1934–1952, Nov. 2012.

[107] C. Gezer and C. Buratti, “A ZigBee Smart Energy Implementation for Energy Efficient Buildings,” *2011 IEEE 73rd Vehicular Technology Conference (VTC Spring)*, pp. 1–5, May 2011.

[108] S. J. Marinković, E. M. Popovici, C. Spagnol, S. Faul, and W. P. Marnane, “Energy-efficient low duty cycle MAC protocol for wireless body area networks.,” *IEEE transactions on information technology in biomedicine : a publication of the IEEE Engineering in Medicine and Biology Society*, vol. 13, pp. 915–925, 2009.

[109] P. Q. P. Qiu, Y. Z. Y. Zhao, U. H. U. Heo, D. Z. D. Zhang, and J. C. J. Choi, “Gateway architecture for zigbee sensor network for remote control over IP network,” *Information and Telecommunication Technologies (APSITT), 2010 8th Asia-Pacific Symposium on*, 2010.

[110] A. Dunkels, “Full TCP/IP for 8 Bit Architectures,” in *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*, 2003.

[111] J.-S. Lee, Y.-W. Su, and C.-C. Shen, “A Comparative Study of Wireless Protocols: Bluetooth, UWB, ZigBee, and Wi-Fi,” *IECON 2007 33rd Annual Conference of the IEEE Industrial Electronics Society*, pp. 46–51, 2007.

[112] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "RFC 4944 - Transmission of IPv6 Packets over IEEE 802.15.4 Networks," tech. rep., IETF, 2007.

[113] J. Hui and P. Thubert, "RFC 6282 - Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks," tech. rep., IETF, 2011.

[114] T. Winter, "RFC 6550 - RPL: IPv6 routing protocol for low-power and lossy networks," tech. rep., IETF, 2012.

[115] U. Herberg and T. Clausen, "A Comparative Performance Study of the Routing Protocols LOAD and RPL with Bi-directional Traffic in Low-power and Lossy Networks (LLN)," in *Proceedings of the 8th ACM Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, and Ubiquitous Networks*, PE-WASUN '11, (New York, NY, USA), pp. 73–80, ACM, 2011.

[116] T. Clausen, "The Lightweight On-demand Ad hoc Distance-vector Routing Protocol - Next Generation (LOADng)," tech. rep., IETF, 2014.

[117] G. del Campo, E. Montoya, J. Martín, I. Gómez, and A. Santamaría, "BatNet: A 6LoWPAN-Based Sensors and Actuators Network," in *Ubiquitous Computing and Ambient Intelligence SE - 8* (J. Bravo, D. López-de Ipiña, and F. Moya, eds.), vol. 7656 of *Lecture Notes in Computer Science*, pp. 58–65, Springer Berlin Heidelberg, 2012.

[118] A. H. Kazmi, M. J. O'grady, D. T. Delaney, A. G. Ruzzelli, and G. M. P. O'hare, "A Review of Wireless-Sensor-Network-Enabled Building Energy Management Systems," *ACM Trans. Sen. Netw.*, vol. 10, no. 4, pp. 66:1—-66:43, 2014.

[119] S. Han, Y.-H. Wei, A. K. Mok, D. Chen, M. Nixon, and E. Rotvold, "Building wireless embedded internet for industrial automation," in *Industrial Electronics Society, IECON 2013 - 39th Annual Conference of the IEEE*, pp. 5582–5587, Nov. 2013.

[120] V. C. Güngör, D. Sahin, T. Kocak, S. Ergüt, C. Buccella, C. Cecati, and G. P. Hancke, "Smart grid technologies: Communication technologies and standards," *IEEE Transactions on Industrial Informatics*, vol. 7, pp. 529–539, 2011.

[121] C.-W. Lu, S.-C. Li, and Q. Wu, "Interconnecting ZigBee and 6LoWPAN wireless sensor networks for smart grid applications," *2011 Fifth International Conference on Sensing Technology*, pp. 267–272, 2011.

[122] F. Mattern and P. Sturm, "From Distributed Systems to Ubiquitous Computing," *The State of the Art Trends and Prospects of Future Networked Systems*, pp. 3—-25, 2003.

[123] M. V. Trifa, *Building Blocks for a Participatory Web of Things : Devices , Infrastructures , and Programming Frameworks.* PhD thesis, ETHZ, 2011.

[124] X. G. X. Guo, J. S. J. Shen, and Z. Y. Z. Yin, "On software development based on SOA and ROA," *Control and Decision Conference (CCDC), 2010 Chinese*, 2010.

[125] E. Wilde, "Putting Things to REST," *Transport*, vol. 15, pp. 1 – 13, 2007.

[126] P. A. Bernstein, "Middleware: a model for distributed system services," *Communications of the ACM*, vol. 39, no. 2, pp. 86–98, 1996.

[127] F. Bellifemine and G. Fortino, "SPINE: a domain-specific framework for rapid prototyping of WBSN applications," *Software: Practice and Experience*, vol. 41, no. 3, pp. 237–265, 2011.

[128] L. Ducreux, C. Guyon-Gardeux, S. Lesecq, and F. Pacull, “Resource-based middleware in the context of heterogeneous building automation systems,” *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, pp. 4847–4852, Oct. 2012.

[129] O. Bergmann, K. T. Hillmann, and S. Gerdes, “A CoAP-gateway for smart homes,” *2012 International Conference on Computing, Networking and Communications (ICNC)*, pp. 446–450, Jan. 2012.

[130] M. Jung, C. Reinisch, and W. Kastner, “Integrating Building Automation Systems and IPv6 in the Internet of Things,” *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 683–688, July 2012.

[131] M. Jung, J. Weidinger, C. Reinisch, W. Kastner, C. Crettaz, A. Olivieri, and Y. Bocchi, “A Transparent IPv6 Multi-protocol Gateway to Integrate Building Automation Systems in the Internet of Things,” *2012 IEEE International Conference on Green Computing and Communications*, pp. 225–233, Nov. 2012.

[132] M. Jung, J. Weidinger, W. Kastner, and A. Olivieri, “Building Automation and Smart Cities: An Integration Approach Based on a Service-Oriented Architecture,” *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pp. 1361–1367, Mar. 2013.

[133] T. Perumal, A. Ramli, and C. Leong, “SOA-Based Framework for Home and Building Automation Systems (HBAS),” *International Journal of Smart Home*, vol. 8, no. 5, pp. 197–206, 2014.

[134] F. Bernier, J. Ploennigs, D. Pesch, S. Lesecq, T. Basten, M. Boubekeur, D. Denteneer, F. Oltmanns, F. Bonnard, M. Lehmann, T. L. Mai, A. McGibney, S. Rea, F. Pacull, C. Guyon-Gardeux, L.-F. Ducreux, S. Thior, M. Hendriks, J. Verriet, and S. Fedor, “Architecture for self-organizing, co-operative and robust Building Automation Systems,” *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*, pp. 7708–7713, Nov. 2013.

[135] M. Fazio, M. Paone, A. Puliafito, and M. Villari, “Heterogeneous Sensors Become Homogeneous Things in Smart Cities,” *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 775–780, July 2012.

[136] T. Kindberg, J. Barton, J. Morgan, and G. Becker, “People, places, things: Web presence for the real world,” *Mobile Networks and*, pp. 365–376, 2002.

[137] B. Ostermaier and F. Schlup, “WebPlug: a framework for the web of things,” in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, pp. 690–695, 2010.

[138] S. Vinosky, “Serendipitous reuse,” *IEEE Internet Computing*, vol. 12, pp. 84–87, 2008.

[139] S. Tilak, N. B. Abu-Ghazaleh, and W. Heinzelman, “A taxonomy of wireless micro-sensor network models,” in *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 6, pp. 28–36, 2002.

[140] L. Mottola and G. P. Picco, “Programming Wireless Sensor Networks : Fundamental Concepts and State of the Art,” *ACM Computing Surveys (CSUR)*, vol. 43, pp. 19:1–19:51, 2011.

[141] B. Hartmann, S. Doorley, and S. R. Klemmer, “Hacking, mashing, gluing: Understanding opportunistic design,” *IEEE Pervasive Computing*, vol. 7, pp. 46–54, 2008.

[142] T. Berners-Lee, "Information Management: A Proposal," *Word Journal Of The International Linguistic Association*, vol. February 2, pp. 1–10, 1989.

[143] T. O'reilly, "What is web 2.0," *Design patterns and business models for the next generation of software*, vol. 30, p. 2005, 2005.

[144] T. Berners-Lee and R. Fielding, "RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax," tech. rep., IETF, 2005.

[145] X. Ke, Z. Xiaoqi, S. Meina, and S. Junde, "Mobile mashup: Architecture, challenges and suggestions," in *Proceedings - International Conference on Management and Service Science, MASS 2009*, 2009.

[146] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, "Comparison of JSON and XML Data Interchange Formats: A Case Study.," *CAINE*, 2009.

[147] T. Bray, "RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format," tech. rep., IETF, 2014.

[148] A. Castellani and M. Gheda, "Web Services for the Internet of Things through CoAP and EXI," in *Communications Workshops (ICC), 2011 IEEE International Conference on*, no. Xml, pp. 1 – 6, 2011.

[149] R. T. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1," tech. rep., IETF, 1999.

[150] D. Yazar and A. Dunkels, "Efficient application integration in IP-based sensor networks," *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings - BuildSys '09*, p. 43, 2009.

[151] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins, "RFC 6202 - Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP," tech. rep., IETF, 2011.

[152] I. Fette and A. Melnikov, "RFC 6455 - The websocket protocol," tech. rep., IETF, 2011.

[153] I. F. Akyildiz, X. Wang, and W. Wang, "Wireless mesh networks: A survey," *Computer Networks*, vol. 47, pp. 445–487, 2005.

[154] K. Sohraby and M. Daneshmand, "A survey of transport protocols for wireless sensor networks," *IEEE Network*, vol. 20, pp. 34–40, 2006.

[155] Z. Shelby, K. Hartke, and C. Bormann, "RFC 7252 - Constrained Application Protocol (CoAP)," tech. rep., IETF, 2014.

[156] B. C. Villaverde, D. Pesch, R. De Paz Alberola, S. Fedor, and M. Boubekeur, "Constrained application protocol for low power embedded networks: A survey," in *Proceedings - 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2012*, pp. 702–707, 2012.

[157] C. Bormann, A. P. Castellani, and Z. Shelby, "CoAP: An application protocol for billions of tiny internet nodes," *IEEE Internet Computing*, vol. 16, pp. 62–67, 2012.

[158] K. Hartke, "Observing Resources in CoAP," tech. rep., IETF, 2014.

[159] A. Rahman and E. Dijk, "RFC 7390 - Group Communication for CoAP," tech. rep., IETF, 2014.

[160] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan, "RFC 3376 - Internet Group Management Protocol, Version 3," tech. rep., IETF, 2002.

[161] S. Deering, W. Fenner, and B. Haberman, "RFC 2710 - Multicast listener discovery (MLD) for IPv6," tech. rep., IETF, 1999.

[162] A. Adams, J. Nicholas, and W. Siadak, "RFC 3973 - Protocol Independent Multicast - Dense Mode," tech. rep., IETF, 2013.

[163] B. Fenner, M. Handley, H. Holbrook, and I. Kouvelas, "RFC 4601 - Protocol Independent Multicast - Sparse Mode," tech. rep., IETF, 2006.

[164] J. Franks, P. Hallam-Baker, and J. Hostetler, "RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication," tech. rep., IETF, 1999.

[165] E. Rescorla and N. Modadugu, "RFC 6347 - Datagram Transport Layer Security," tech. rep., IETF, 2012.

[166] T. Kothmayr, C. Schmitt, W. Hu, M. Brünig, and G. Carle, "DTLS based security and two-way authentication for the Internet of Things," *Ad Hoc Networks*, vol. 11, pp. 2710–2723, 2013.

[167] S. Raza, H. Shafagh, K. Hewage, R. Hummen, and T. Voigt, "Lithe: Lightweight secure CoAP for the internet of things," *IEEE Sensors Journal*, vol. 13, pp. 3711–3720, 2013.

[168] M. Brachmann, S. L. Keoh, O. G. Morchon, and S. S. Kumar, "End-to-end transport security in the IP-based internet of things," in *2012 21st International Conference on Computer Communications and Networks, ICCCN 2012 - Proceedings*, 2012.

[169] R. Hummen, J. H. Ziegeldorf, H. Shafagh, S. Raza, and K. Wehrle, "Towards viable certificate-based authentication for the internet of things," *Proceedings of the 2nd ACM workshop on Hot topics on wireless network security and privacy - HotWiSec '13*, p. 37, 2013.

[170] H. Tschofenig, "A Datagram Transport Layer Security (DTLS) 1.2 Profile for the Internet of Things," tech. rep., IETF, 2014.

[171] G. Peretti, "CoAP over DTLS TinyOS Implementation and Performance Analysis," tech. rep., University of Padosa, 2013.

[172] S. Jucker, "Securing the Constrained Application Protocol," Tech. Rep. October, ETHZ, 2012.

[173] L. Seitz, S. Gerdes, G. Selander, M. Mani, and S. Kumar, "ACE use cases," tech. rep., IETF, 2014.

[174] L. Seitz and G. Selander, "Problem Description for Authorization in Constrained Environments," tech. rep., IETF, 2014.

[175] E. Wahlstroem, "OAuth 2.0 Introspection over the Constrained Application Protocol (CoAP)," tech. rep., IETF, 2014.

[176] H. Tschofenig, "The OAuth 2.0 Internet of Things (IoT) Client Credentials Grant," tech. rep., IETF, 2014.

[177] H. Tschofenig, "The OAuth 2.0 Bearer Token Usage over the Constrained Application," tech. rep., IETF, 2014.

[178] R. Sanchez, R. Marin, and D. Garcia, "EAP-based Authentication Service for CoAP," tech. rep., IETF, 2014.

[179] M. L. Mazurek, J. P. Arsenault, J. Bresee, N. Gupta, I. Ion, C. Johns, D. Lee, Y. Liang, J. Olsen, B. Salmon, R. Shay, K. Vaniea, L. Bauer, L. F. Cranor, G. R. Ganger, and M. K.

Reiter, "Access Control for Home Data Sharing : Attitudes , Needs and Practices," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 645–654, 2009.

[180] W. Granzer, F. Praus, and W. Kastner, "Security in Building Automation Systems," *IEEE Transactions on Industrial Electronics*, vol. 57, pp. 3622–3630, Nov. 2010.

[181] M. Vucinic and B. Tourancheau, "OSCAR: Object Security Architecture for the Internet of Things," in *Proceedings of the IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2014)*, 2014.

[182] J. Liu, Y. Xiao, and C. P. Chen, "Authentication and Access Control in the Internet of Things," in *2012 32nd International Conference on Distributed Computing Systems Workshops*, pp. 588–592, 2012.

[183] M. Turkanović, B. Brumen, and M. Hölbl, "A novel user authentication and key agreement scheme for heterogeneous ad hoc wireless sensor networks, based on the Internet of Things notion," *Ad Hoc Networks*, vol. 20, pp. 96–112, 2014.

[184] T. Levä, O. Mazhelis, and H. Suomi, "Comparing the cost-efficiency of CoAP and HTTP in Web of Things applications," *Decision Support Systems*, vol. 63, pp. 23–38, 2014.

[185] A. Ludovici, P. Moreno, and A. Calveras, "TinyCoAP: A Novel Constrained Application Protocol (CoAP) Implementation for Embedding RESTful Web Services in Wireless Sensor Networks Based on TinyOS," *Journal of Sensor and Actuator Networks*, vol. 2, pp. 288–315, 2013.

[186] M. Kovatsch, S. Duquennoy, and A. Dunkels, "A low-power CoAP for Contiki," in *Proceedings - 8th IEEE International Conference on Mobile Ad-hoc and Sensor Systems, MASS 2011*, pp. 855–860, 2011.

[187] P. Mockapetris, "RFC 1035 - Domain names: Implementation and Specification," tech. rep., IETF, 1987.

[188] T. R. Henderson, "Host Mobility for IP Networks: A Comparison," *IEEE Network*, vol. 17, pp. 18–26, 2003.

[189] T. Tomonari and S. Kimura, "Mobility support in IPv6 networks with Dynamic DNS servers," in *Awareness Science and Technology and Ubi-Media Computing (iCAST-UMEDIA), 2013 International Joint Conference on*, pp. 729–735, Ieee, Nov. 2013.

[190] S. Cheshire and M. Krochmal, "RFC 6762 - Multicast DNS," tech. rep., IETF, 2013.

[191] M. Masdari, M. Maleknasab, and M. Bidaki, "A survey and taxonomy of name systems in mobile ad hoc networks," *Journal of Network and Computer Applications*, vol. 35, pp. 1493–1507, Sept. 2012.

[192] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Computing Surveys*, vol. 36, pp. 335–371, 2004.

[193] G. Fortino and A. Guerrieri, "Decentralized and embedded management for smart buildings," in *Proceedings of the workshop on applications of software agents*, pp. 3–7, 2011.

[194] L. Huang, "VIRGO P2P Based Distributed DNS Framework for IPv6 Network," *2008 Fourth International Conference on Networked Computing and Advanced Information Management*, pp. 698–702, Sept. 2008.

[195] R. Farha and A. Leon-Garcia, "Peer-to-peer naming architecture for integrated wireline / Wireless networks," in *GLOBECOM - IEEE Global Telecommunications Conference*, pp. 2019–2024, IEEE, Nov. 2007.

[196] N. Montavont and T. Noel, "Handover management for mobile nodes in IPv6 networks," *Communications Magazine, IEEE*, no. August, pp. 38–43, 2002.

[197] E. R. Koodli, "RFC 5568 - Mobile IPv6 Fast Handovers," tech. rep., IETF, 2009.

[198] R. Morera and A. McAuley, "Adapting DNS to dynamic ad hoc networks," in *Military Communications Conference, 2005. MILCOM 2005. IEEE*, no. 2, pp. 1303 – 1308, 2005.

[199] K. Sripanidkulchai, B. Maggs, and H. Zhang, "Efficient content location using interest-based locality in peer-to-peer systems," *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*, vol. 3, 2003.

[200] G. Brambilla, M. Picone, M. Amoretti, and F. Zanichelli, "An Adaptive Peer-to-Peer Overlay Scheme for Location-Based Services," *2014 IEEE 13th International Symposium on Network Computing and Applications*, pp. 181–188, Aug. 2014.

[201] S. Aomumpai, K. Kondee, C. Prommak, and K. Kaemarungsi, "Optimal placement of reference nodes for wireless indoor positioning systems," *2014 11th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, pp. 1–6, May 2014.

[202] J. Yicka, B. Mukherjeea, and D. Ghosal, "Wireless sensor network survey," *Computer Networks*, vol. 58, pp. 2292–2330, 2008.

[203] F. Silva, A. Boukerche, and T. Silva, "Content Replication and Delivery in Vehicular Networks," *Proceedings of the fourth ACM international symposium on Development and analysis of intelligent vehicular networks and applications*, pp. 127–132, 2014.

[204] J. Choi, J. Han, E. Cho, T. T. Kwon, and Y. Choi, "A survey on content-oriented networking for efficient content delivery," *IEEE Communications Magazine*, vol. 49, pp. 121–127, 2011.

[205] T. A. Butt, I. Phillips, L. Guan, and G. Oikonomou, "TRENDY," in *Proceedings of the Third International Workshop on the Web of Things - WOT '12*, pp. 1–6, 2012.

[206] C. Ge, S. Member, Z. Sun, and N. Wang, "A Survey of Power-Saving Techniques on Data Centers and Content Delivery Networks," *1334 IEEE Communications Survey & Tutorials*, vol. 15, pp. 1334–1354, 2013.

[207] P. Mockapetris, "RFC 1034 - Domain names: Concepts and Facilities," tech. rep., IETF, 1987.

[208] F. Zhu, M. W. Mutka, and L. M. Ni, "Service Discovery in Pervasive Computing Environments," *Pervasive Computing*, pp. 81–90, 2005.

[209] B. Sapkota, D. Roman, S. Kruk, and D. Fensel, "Distributed Web Service Discovery Architecture," *Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW'06)*, 2006.

[210] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, vol. 284, no. 5, pp. 28–37, 2001.

[211] C. Bettstetter and C. Renner, “A comparison of service discovery protocols and implementation of the service location protocol,” *Proceedings of the 6th EUNICE Open European Summer School: Innovative Internet Applications*, 2000.

[212] D. Chakraborty, A. Joshi, and S. Member, “Toward Distributed Service Discovery in Pervasive Computing Environments,” *IEEE Transactions on Mobile Computing*, vol. 5, no. 2, pp. 97–112, 2006.

[213] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003.

[214] A. Gomez-Perez, O. Corcho, and M. Fernandez-Lopez, *Ontological Engineering : with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web.* Advanced Information and Knowledge Processing, 2004.

[215] C. Reinisch, W. Granzer, F. Praus, and W. Kastner, “Integration of heterogeneous building automation systems using ontologies,” *2008 34th Annual Conference of IEEE Industrial Electronics*, pp. 2736–2741, Nov. 2008.

[216] M. Botts and A. Robin, “OpenGIS ® Sensor Model Language (SensorML) Implementation Specification,” tech. rep., Open Geospatial Consortium, 2007.

[217] G. Klyne and J. J. Carroll, “Resource Description Framework (RDF): Concepts and Abstract Syntax,” *W3C Recommendation*, vol. 10, pp. 1—-20, 2004.

[218] N. Choi, I.-Y. Song, and H. Han, “A survey on ontology mapping,” in *ACM SIGMOD Record*, vol. 35, pp. 34–41, 2006.

[219] F. Zablith, G. Antoniou, M. Aquin, G. O. S. Flouris, H. Kondylakis, E. Motta, and M. Sabou, *Ontology Evolution : A Process Centric Survey*, vol. 00. Cambridge University Press, 2013.

[220] R. Iannella and J. McKinney, “vCard Ontology for describing People and Organizations,” tech. rep., W3C, 2014.

[221] A. Donoho, J. Costa-requena, T. Mcgee, A. Messer, A. Fiddian-green, and J. Fuller, “UPnP$^{\mathrm{TM}}$ Device Architecture 1.1,” *Architecture*, pp. 1–136, 2008.

[222] E. Guttman, C. Perkins, J. Veizades, and M. Day, “RFC 2608 - Service Location Protocol, Version 2,” tech. rep., IETF, 1999.

[223] M. Sarnovský and P. Kostelník, “Device Description in HYDRA Middleware,” in *Proceedings of the 2nd Workshop on Intelligent and Knowledge oriented Technologies*, 2007.

[224] S. Han, G. Lee, and N. Crespi, “Semantic Context-aware Service Composition for Building Automation System,” *ieeexplore.ieee.org*, pp. 1–10, 2013.

[225] D. Pfisterer and K. Romer, “SPITFIRE: toward a semantic web of things,” *IEEE Communications Magazine*, vol. 49, no. 11, pp. 40–48, 2011.

[226] K. G. Clark, K. Grant, and E. Torres, “SPARQL Protocol for RDF,” tech. rep., W3C, 2008.

[227] S. Harris and A. Seaborne, “SPARQL 1.1 Query Language,” tech. rep., W3C, 2013.

[228] D. Boldt, H. Hasemann, and A. Kröller, “SPARQL for Networks of Embedded Systems,” *CoRR*, vol. abs/1402.7, 2014.

[229] J. Jeong, J. Park, and H. Kim, "DNS Name Service based on Secure Multicast DNS for IPv6 Mobile Ad Hoc Networks," *Advanced Communication Technology, 2004. The 6th International Conference on*, vol. 1, pp. 3–7, 2004.

[230] X. Hong, J. Liu, R. Smith, and Y. Lee, "Distributed naming system for mobile ad-hoc networks," *contract*, 2005.

[231] J. Jeong, J. Park, and H. Kim, "Name service in IPv6 mobile ad-hoc network connected to the internet," *Personal, Indoor and Mobile Radio Communications, 2003. PIMRC 2003. 14th IEEE Proceedings on*, vol. 2, pp. 1351–1355, 2003.

[232] Y. Gottlieb, "MOSS: Gathering Names in Networks of Mobile Nodes," *MILCOM 2007*, pp. 1–6, 2007.

[233] M. Nazeeruddin, G. Parr, and B. Scotney, "An efficient and robust name resolution protocol for dynamic MANETs," *Ad Hoc Networks*, vol. 8, pp. 842–856, Nov. 2010.

[234] I. Baumgart, "P2PNS: A Secure Distributed Name Service for P2PSIP," *2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pp. 480–485, Mar. 2008.

[235] F. Furfari, L. Sommaruga, C. Soria, and R. Fresco, "DomoML: the definition of a standard markup for interoperability of human home interactions," *Proceedings of the 2nd European Union symposium on Ambient intelligence*, pp. 41–44, 2004.

[236] J. Ploennigs and B. Hensel, "BASont-A modular, adaptive building automation system ontology," *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, pp. 4827–4833, 2012.

[237] R. Verborgh and T. Steiner, "Efficient runtime service discovery and consumption with hyperlinked RESTdesc," in *Next Generation Web Services Practices (NWeSP), 2011 7th International Conference on*, pp. 373–379, 2011.

[238] a. Kovacevic, J. Ansari, and P. Mahonen, "NanoSD: A Flexible Service Discovery Protocol for Dynamic and Heterogeneous Wireless Sensor Networks," *2010 Sixth International Conference on Mobile Ad-hoc and Sensor Networks*, pp. 14–19, Dec. 2010.

[239] S. Cheshire and M. Krochmal, "RFC 6763 - DNS-Based Service Discovery," tech. rep., IETF, 2013.

[240] L. Schor and P. Sommer, "Towards a zero-configuration wireless sensor network architecture for smart buildings," *for Energy-Efficiency in Buildings*, 2009.

[241] S. Mayer and D. Guinard, "An extensible discovery service for smart things," in *Proceedings of the Second International Workshop on Web of Things - WoT '11*, (New York, New York, USA), p. 1, ACM Press, 2011.

[242] G. Mathur, P. Desnoyers, P. Chukiu, D. Ganesan, and P. Shenoy, "Ultra-low power data storage for sensor networks," *ACM Transactions on Sensor Networks*, vol. 5, pp. 1–34, Nov. 2009.

[243] J. Ledlie, C. Ng, and D. Holland, "Provenance-aware sensor data storage," in *Data Engineering Workshops, 2005. 21st International Conference on*, p. 1189, 2005.

[244] B. Sheng, Q. Li, and W. Mao, "Data storage placement in sensor networks," *Proceedings of the seventh ACM international symposium on Mobile ad hoc networking and computing - MobiHoc '06*, p. 344, 2006.

[245] C. Williams, P. Huibonhoa, J. Holliday, A. Hospodor, and T. Schwarz, "Redundancy management for P2P storage," in *Proceedings - Seventh IEEE International Symposium on Cluster Computing and the Grid, CCGrid 2007*, pp. 15–22, 2007.

[246] L. Toka, P. Cataldi, M. Dell'Amico, and P. Michiardi, "Redundancy management for P2P backup," in *Proceedings - IEEE INFOCOM*, pp. 2986–2990, 2012.

[247] E. Mingozzi and G. Tanganelli, "An open framework for accessing Things as a service," in *Wireless Personal Multimedia Communications (WPMC), 2013 16th International Symposium on*, pp. 1–5, 2013.

[248] S. A. Aly, A. Ali-Eldin, and H. V. Poor, "A Distributed Data Collection Algorithm for Wireless Sensor Networks with Persistent Storage Nodes," *2011 4th IFIP International Conference on New Technologies, Mobility and Security*, pp. 1–5, 2011.

[249] H. Shen, L. Zhao, and Z. Li, "A distributed spatial-temporal similarity data storage scheme in wireless sensor networks," *IEEE Transactions on Mobile Computing*, vol. 10, pp. 982–996, 2011.

[250] R. Sarkar, W. Zeng, J. Gao, and X. D. Gu, "Covering space for in-network sensor data storage," *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks - IPSN '10*, p. 232, 2010.

[251] H. Sane, C. Haugstetter, and S. Bortoff, "Building HVAC control systems - role of controls and optimization," *2006 American Control Conference*, 2006.

[252] F. Zamora-Martínez, P. Romeu, P. Botella-Rocamora, and J. Pardo, "On-line learning of indoor temperature forecasting models towards energy efficiency," *Energy and Buildings*, vol. 83, pp. 162–172, Nov. 2014.

[253] A. Jordan, "On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes," *Advances in neural information processing systems*, 2002.

[254] C. E. Rasmussen and C. K. I. Williams, *Gaussian processes for machine learning*, vol. 14. The MIT Press, 2006.

[255] J. Yu, B. Benatallah, F. Casati, and F. Daniel, "Understanding mashup development," *IEEE Internet Computing*, vol. 12, pp. 44–52, 2008.

[256] S. Davidoff, M. Lee, and C. Yiu, "Principles of smart home control," in *UbiComp 2006*, pp. 19–34, Springer Berlin Heidelberg, 2006.

[257] L. Yangqun, "A Light-Weight Rule-Based Monitoring System for Web of Things," *2013 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pp. 251–254, Oct. 2013.

[258] V. Kumar, A. Fensel, and P. Fröhlich, "Context Based Adaptation of Semantic Rules in Smart Buildings," *Proceedings of International Conference on Information Integration and Web-based Applications & Services - IIWAS '13*, pp. 719–728, 2013.

[259] M. Kovatsch, M. Lanter, and S. Duquennoy, "Actinium: A RESTful runtime container for scriptable Internet of Things applications," *2012 3rd IEEE International Conference on the Internet of Things*, pp. 135–142, Oct. 2012.

[260] D. Alessandrelli, M. Petraccay, and P. Pagano, "T-Res: Enabling Reconfigurable In-network Processing in IoT-based WSNs," *2013 IEEE International Conference on Distributed Computing in Sensor Systems*, vol. 317671, pp. 337–344, May 2013.

[261] A. Zabasta, N. Kunicina, Y. Chaiko, and L. Ribickis, “Automatic wireless meters reading for water distribution network in Talsi city,” in *EUROCON 2011 - International Conference on Computer as a Tool - Joint with Conftele 2011*, 2011.

[262] O. Chudnovskyy, F. Weinhold, H. Gebhardt, and M. Gaedke, “Integration of telco services into enterprise mashup applications,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7059 LNCS, pp. 37–48, 2012.

[263] G. Acampora and V. Loia, “Fuzzy control interoperability and scalability for adaptive domotic framework,” *IEEE Transactions on Industrial Informatics*, vol. 1, pp. 97–111, May 2005.

[264] H. Hagras and V. Callaghan, “Creating an ambient-intelligence environment using embedded agents,” *Intelligent Systems*, vol. 19, no. 6, pp. 12–20, 2004.

[265] E. Alba and J. M. Troya, “A survey of parallel distributed genetic algorithms,” *Complexity*, vol. 4, pp. 31–52, 1999.

[266] C. F. J. Wu, “On the Convergence Properties of the EM Algorithm,” *The Annals of Statistics*, vol. 11, no. 1, pp. 95–103, 1983.

[267] D. Guinard, V. Trifa, E. Wilde, and U. C. Berkeley, “A Resource Oriented Architecture for the Web of Things,” in *IEEE International Conference on the Internet of Things (IOT)*, pp. 1–8, 2010.

[268] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach, Third edition*. Prentice Hall, 2014.

[269] L. R. Rabiner, “A tutorial on hidden Markov models and selected applications in speech recognition,” *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[270] D. A. Reynolds, “Gaussian Mixture Models,” *Encyclopedia of Biometric Recognition*, vol. 31, pp. 1047–64, 2008.

[271] A. Guillemin and N. Morel, “Innovative lighting controller integrated in a self-adaptive building control system,” *Energy and Buildings*, vol. 33, no. 5, pp. 477–487, 2001.

[272] S. Stumpf, V. Rajaram, L. Li, W.-K. Wong, M. Burnett, T. Dietterich, E. Sullivan, and J. Herlocker, “Interacting meaningfully with machine learning systems: Three experiments,” *International Journal of Human-Computer Studies*, vol. 67, pp. 639–662, Aug. 2009.

[273] T. S. Guzella and W. M. Caminhas, “A review of machine learning approaches to Spam filtering,” *Expert Systems with Applications*, vol. 36, pp. 10206–10222, Sept. 2009.

[274] S. Stumpf, M. Burnett, V. Pipek, and W.-K. Wong, “End-user interactions with intelligent and autonomous systems,” *Proceedings of the 2012 ACM annual conference extended abstracts on Human Factors in Computing Systems Extended Abstracts - CHI EA ’12*, p. 2755, 2012.

[275] Y. Ma and F. Borrelli, “Fast stochastic predictive control for building temperature regulation,” in *American Control Conference (ACC), 2012*, pp. 3075–3080, 2012.

[276] J. Široký, F. Oldewurtel, J. Cigler, and S. Prívara, “Experimental analysis of model predictive control for an energy efficient building heating system,” *Applied Energy*, vol. 88, pp. 3079–3087, 2011.

[277] F. Oldewurtel, A. Parisio, C. N. Jones, M. Morari, D. Gyalistras, M. Gwerder, V. Stauch, B. Lehmann, and K. Wirth, "Energy efficient building climate control using Stochastic Model Predictive Control and weather predictions," *American Control Conference ACC 2010*, vol. 45, pp. 15–27, 2010.

[278] W. Kastner, G. Neugschwandtner, and M. Kogler, "An open approach to EIB/KNX software development," *Fieldbus Systems and their Applications*, vol. 6, pp. 255–262, 2005.

[279] D. Vassis, G. Kormentzas, A. Rouskas, and I. Maglogiannis, "The IEEE 802.11g standard for high data rate WLANs," *IEEE Network*, vol. 19, pp. 21–26, 2005.

[280] W. Stevens, *TCP/IP Illustrated, vol. 1, 1994*. Addison-Wesley Professional, 1994.

[281] M. Gast, "802.11 Wireless Networks: The Definitive Guide, Second Edition," *OReilly Media*, pp. 1–656, 2005.

[282] K. Li, R. Sun, and G. Wei, "CoAP Option Extension: NodeId," tech. rep., IETF, 2014.

[283] C. Groba and S. Clarke, "Web services on embedded systems - a performance study," *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops PERCOM Workshops*, vol. 3, no. 6, pp. 726–731, 2010.

[284] H. Hamad, M. Saad, and R. Abed, "Performance Evaluation of RESTful Web Services," *Computer Engineering*, vol. 1, no. 3, pp. 72–78, 2010.

[285] W. Colitti, K. Steenhaut, N. De Caro, B. Buta, and V. Dobrota, "Evaluation of constrained application protocol for wireless sensor networks," in *18th IEEE Workshop on18th IEEE Workshop on Local & Metropolitan Area Networks (LANMAN), 2011*, pp. 1–6, 2011.

[286] X. Li and S. Qiu, "Research on Multicast Routing Protocol in Wireless Sensor Network," *2011 International Conference on Control, Automation and Systems Engineering (CASE)*, pp. 1–4, 2011.

[287] Y. Huang, W. Wang, X. Zhang, and Y. Wei, "An energy efficient multicast transmission scheme with patching stream exploiting user behavior in wireless networks," in *GLOBECOM - IEEE Global Telecommunications Conference*, pp. 3537–3541, 2012.

[288] G. Oikonomou, I. Phillips, and T. Tryfonas, "IPv6 multicast forwarding in RPL-based wireless sensor networks," *Wireless Personal Communications*, vol. 73, pp. 1089–1116, 2013.

Referenced Web Resources

[289] "Ipso alliance: Enabling the internet of things." http://ipso-alliance.org/.

[290] "Compression format for ipv6 datagrams over ieee 802.15.4-based networks." https://tools.ietf.org/html/rfc6282.

[291] "sens.se internet of things platform." https://sen.se/store/mother/.

[292] "Basics of bacnet." http://kargs.net/BACnet/BACnet_Basics.pdf.

[293] "Iso 16484-3:2005." http://www.iso.org/iso/catalogue_detail.htm?csnumber=37205.

[294] "Distributed component object model (dcom) remote protocol specification." http://msdn.microsoft.com/library/cc201989.aspx.

[295] "Java remote method invocation (rmi)." http://docs.oracle.com/javase/tutorial/rmi/.

[296] "Pyro distributed object middleware for python." https://pypi.python.org/pypi/Pyro4.

[297] "Common object request broker architecture (corba)." http://www.omg.org/spec/CORBA/3.3/.

[298] "Javaspaces." http://www.oracle.com/technetwork/articles/java/javaspaces-140665.html.

[299] "Simple object access protocol (soap)." http://www.w3.org/TR/soap/.

[300] "Oasis web services dynamic discovery." http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.html.

[301] "Web services eventing." http://www.w3.org/Submission/WS-Eventing/.

[302] "Oasis web services security." https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

[303] E. Alliance, "Enocean technology – energy harvesting wireless." http://www.enocean.com/fileadmin/redaktion/pdf/white_paper/WP_EnOcean_Technology_en_Jul11.pdf, July 2011.

[304] "Resource description framework (rdf)." http://www.w3.org/RDF/.

[305] "Clickscript mashup editor." http://clickscript.ch/site/home.php.

[306] "Message queuing telemetry transport." http://mqtt.org/.

[307] "Jersey - restful web services in java." https://jersey.java.net/.

[308] "Windows communication foundation." http://msdn.microsoft.com/en-us/library/ms731082%28v=vs.110%29.aspx.

[309] "Californium (cf) coap framework in java." http://people.inf.ethz.ch/mkovatsc/californium.php.

[310] "microcoap c library." https://github.com/1248/microcoap.

[311] "Wt http library." http://www.webtoolkit.eu/wt.

[312] "onion http library." https://github.com/davidmoreno/onion/.

[313] "libmicrohttpd http library." http://www.gnu.org/software/libmicrohttpd/.

[314] "libwebsockets websocket library." https://libwebsockets.org/trac/libwebsockets.

[315] "Web services description language (wsdl)." http://www.w3.org/TR/wsdl.

[316] "obix open building information xchange." http://www.obix.org/.

[317] "microformats." http://microformats.org/.

[318] "Goodrelations ontology." http://www.heppnetz.de/ontologies/goodrelations/v1.owl.

[319] "Goodrelations the web vocabulary for e-commerce." http://www.heppnetz.de/projects/goodrelations/.

[320] "Notation3 (n3): A readable rdf syntax." http://www.w3.org/TeamSubmission/n3/.

[321] "hcard 1.0." http://microformats.org/wiki/hcard-fr.

[322] "Ucum ontology." http://ontology-of-clinical-research.googlecode.com/svn/trunk/Archive/ucum.owl.

[323] "Ucum instances ontology." http://idi.fundacionctic.org/muo/ucum-instances.owl.

[324] "Semantic sensor network ontology." http://www.w3.org/2005/Incubator/ssn/ssnx/ssn.

[325] "Dolce ultralite ontology." http://www.ontologydesignpatterns.org/ont/dul/DUL.owl.

[326] "Restdec semantic descriptions for hypermedia apis." http://restdesc.org/.

[327] "Restful service description language (rsdl)." http://www.balisage.net/Proceedings/vol10/html/Robie01/BalisageVol10-Robie01.html.

[328] "Turtle - terse rdf triple language." http://www.w3.org/TeamSubmission/turtle/.

[329] "Rdf 1.1 n-quads." http://www.w3.org/TR/n-quads/.

[330] "Json-ld 1.0." http://www.w3.org/TR/json-ld/.

[331] "Rdf 1.1 xml syntax." http://www.w3.org/TR/rdf-syntax-grammar/.

[332] "Rasqal rdf query library." http://librdf.org/rasqal/.

[333] "Django rest framework." http://www.django-rest-framework.org/.

[334] "node.js javascript web server." http://nodejs.org/.

[335] "Erbium (er) rest engine and coap implementation for contiki." http://people.inf.ethz.ch/mkovatsc/erbium.php.

[336] "libcoap: C-implementation of coap." http://libcoap.sourceforge.net/.

[337] “Java library for coap.” https://code.google.com/p/jcoap/.

[338] “Java library for coap.” https://github.com/okleine/nCoAP.

[339] “Sensor model language (sensorml).” http://www.opengeospatial.org/standards/sensorml.

[340] “Hdfs architecture guide.” https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[341] “Json schema.” http://json-schema.org/.

[342] “Amazon rds with mysql.” http://aws.amazon.com/rds/mysql/?nc2=h_ls.

[343] “Mongodb as a service.” https://mongolab.com/.

[344] “Switchlan.” http://www.switch.ch/network/.

[345] “Predictive model markup language (pmml).” http://www.dmg.org/v4-2-1/GeneralStructure.html.

[346] “Google prediction api.” https://developers.google.com/prediction/?hl=EN.

[347] “Bigml.” https://bigml.com/.

[348] “Binary json.” http://bsonspec.org/.

[349] “Jetty web server.” http://eclipse.org/jetty/.

[350] “yahoo pipes mashup editor.” https://pipes.yahoo.com/pipes/.

[351] “Mashablelogic mashup editor.” http://mashablelogic.com/mashup_editor.htm.

[352] “Matlab urlread2 library.” http://www.mathworks.com/matlabcentral/fileexchange/35693-urlread2/content/urlread2.m.

[353] “Matlab jsonlab library.” http://www.mathworks.com/matlabcentral/fileexchange/33381-jsonlab--a-toolbox-to-encode-decode-json-files-in-matlab-octave.

[354] “Authentication and authorization for constrained environments (ace).” https://datatracker.ietf.org/wg/ace/documents/.

[355] “Serial data transmission and knx protocol.” http://www.knx.org/media/docs/KNX-Tutor-files/Session-2/Serial-Data-Transmission-and-KNX-Protocol.pdf.

[356] “Knx ip – using ip networks as knx medium.” http://www.knx.org/fileadmin/downloads/05%20-%20KNX%20Partners/03%20-%20Becoming%20a%20KNX%20Scientific%20Partner/2010-11%20Conference/Presentations/Session%202.pdf.

[357] “Calimero: Knx network access for java.” http://sourceforge.net/p/calimero/wiki/Home/.

[358] “Enocean serial protocol 3 (esp3).” https://www.enocean.com/esp.

[359] “openhab building automation platform.” http://www.openhab.org/.

[360] “Constrained restful environments (core).” https://datatracker.ietf.org/wg/core/documents/.

Résumé

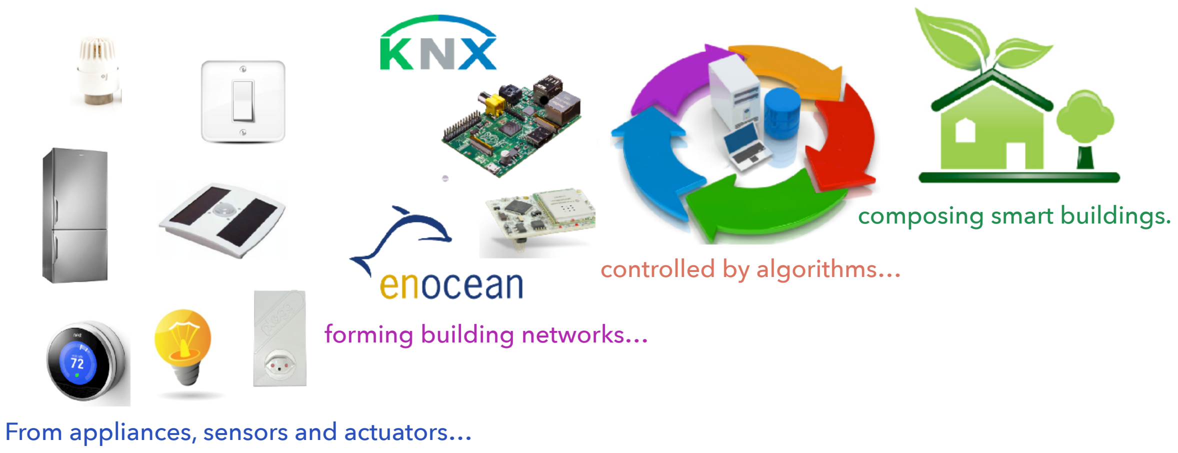

KNX
enocean
From appliances, sensors and actuators...
forming building networks...
controlled by algorithms...
composing smart buildings.

Figure 6.1: Une série d'étapes sont nécessaires avant que des économies d'énergie et une amélioration du confort puissent être obtenus dans les bâtiments intelligents. Les algorithmes de contrôle s'appuient sur les fonctionnalités offertes par les réseaux de bâtiments, eux-même composés d'objets intelligents. Ces objets peuvent être de l'électroménager, des composants multimédias tels que TV et radio, ainsi que des capteurs et des actionneurs.

Dans le domaine des bâtiments, l'efficacité énergétique peut être définie, de façon simplifiée, comme un compromis entre l'énergie économisée et le confort des usagers. Depuis quelques années, des systèmes automatiques de gestion des bâtiments de plus en plus complexes sont étudiés dans le but de l'optimiser. Historiquement, le contrôle thermique des bâtiments était simpliste, souvent basé sur des thermostats généraux ou pièce par pièce, visant une température de consigne. Motivées par les coûts croissants de l'énergie et par la prise de conscience de l'importance de la qualité du confort, des stratégies de plus en plus avancées ont été développées. Des systèmes ont vu le jour pour le contrôle joint du chauffage, de la ventilation et de l'air conditionné (systèmes HVAC). Aujourd'hui, la complexité de ces derniers n'a cessé de grandir incluant le contrôle de l'éclairage, de l'ouverture des fenêtres, des volets, des accès et des systèmes de production locale d'énergie, qu'ils soient de type solaire thermique, photovoltaïque ou géothermique. Les bâtiments sont devenus « intelligents » et incluent désormais de véritables systèmes d'information reposant sur un grand nombre de capteurs et d'actionneurs interconnectés par le biais de réseaux de communication, tels que KNX, EnOcean ou encore BACnet. D'un autre côté, nous observons l'apparition de capteurs se basant se le paradigme de l'Internet des Objets, pouvant nativement communiquer via des réseaux IP. La complexité de ces systèmes d'information croît également, par exemple, en permettant la détection du comportement des

utilisateurs ou en leur donnant un contrôle plus avancé à travers des applications mobiles. Cette évolution des systèmes d'automatisation est illustrée sur la Figure 6.1.

Il est aujourd'hui courant que plusieurs réseaux soient présents dans un seul bâtiment, comme illustré sur la Figure 6.2. Ceci est principalement dû à l'obsolescence technologique ou à des contraintes de câblage, imposant la transition vers un réseau sans fil. Cette situation conduit vers une hétérogénéité des réseaux, obligeant l'intégration de chaque technologie dans le système de gestion. Ces protocoles étant majoritairement incompatibles à cause de leur conception, leur intégration est coûteuse et demande de grands efforts de maintenance en cas de changement de version. Nous pouvons résumer la situation actuelle, en observant qu'il y a un manque de standardisation, spécialement au niveau applicatif.

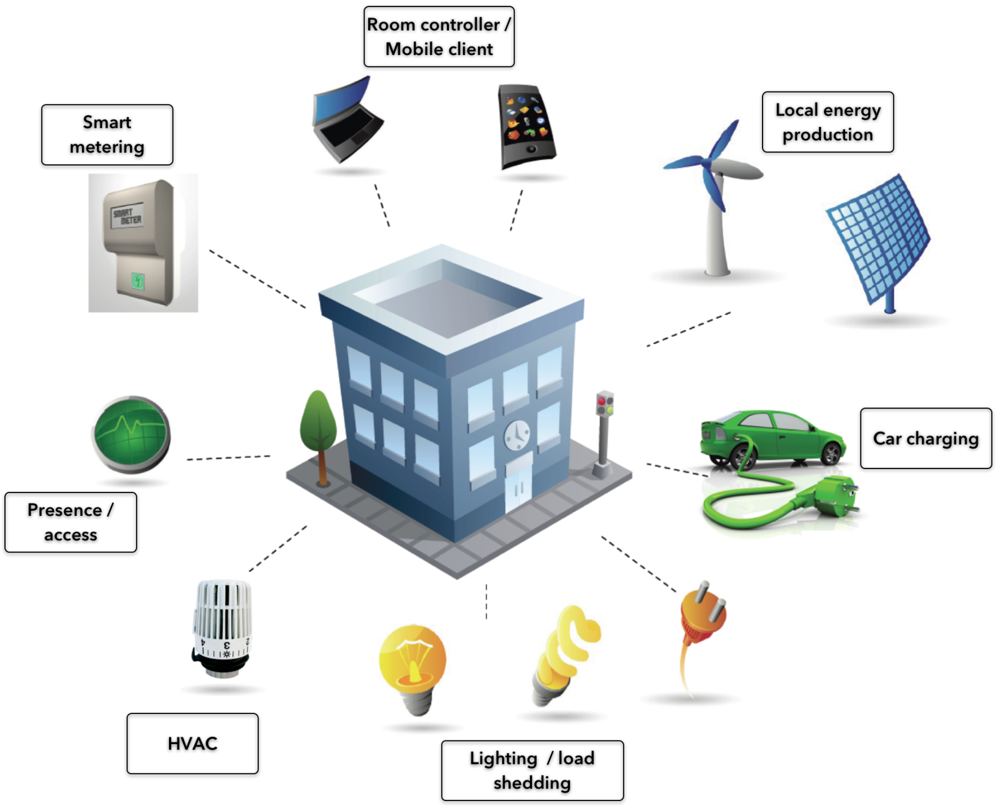


Figure 6.2: De nos jours, les bâtiments sont équipés avec de nombreux systèmes gérant différents aspects de l'automatisation. Ces technologies ne sont que peu compatibles entre elles, créant de ce fait des réseaux de capteurs et actionneurs isolés.

Le paradigme du Web des Objets propose l'utilisation de technologies Web, essentiellement HTTP en suivant le style architectural REST afin d'homogénéiser le niveau applicatif. Pour ceci, les capteurs, actionneurs et les objets de la vie courante embarquent un serveur Web exposant des services REST, menant au concept *d'objets intelligents*. Cette approche diminue le couplage entre les composants tout en réutilisant des technologies éprouvées et omniprésentes.

L'objectif de cette thèse est d'appliquer le Web des Objets aux bâtiments intelligents tout en l'adaptant à leurs spécificités et contraintes. Cela se représente par le développement d'un nouveau paradigme, que nous avons appelé le *Web des Bâtiments*. Les contributions visant à apporter une solution de bout en bout sont multiples:

Utilisation des technologies Web dans l'automatisation du bâtiment: Le Web des Objets fournit un ensemble de techniques pour pousser les technologies du Web sur les objets. Nous nous basons sur celles-ci afin de proposer des directives architecturales spécifiquement adaptées aux bâtiments. De plus, nous montrons par des exemples concrets comment les technologies Web peuvent être utilisées avec des objets de la vie courante.

Considérer les réseaux de capteurs comme des systèmes dans le nuage: Les objets intelligents sont aujourd'hui dotés de capacités de calcul et de mémoire généralement sous-exploitées. Dans notre approche, nous utilisons ces ressources afin de migrer des services exécutés sur des machines dédiées vers les objets intelligents, formant un nuage local dans le réseau d'automatisation. Concrètement, nous sommes en mesure de fournir un système de résolution de noms, de stockage de mesures, ainsi que d'exécuter des algorithmes avancés directement dans le réseau sans nécessiter de matériel particulier.

Intégration de l'apprentissage automatique dans le Web: L'automatisation du bâtiment basée sur l'apprentissage automatique est de plus en plus utilisée pour exploiter la grande quantité de données générée par les capteurs. Les systèmes actuels fonctionnant avec des logiciels principalement fermés, il est difficile de les intégrer avec des réseaux de capteurs. Nous proposons donc de considérer l'apprentissage automatique, et en particulier les algorithmes HMMs comme des ressources Web. En outre, nous démontrons qu'il est possible de déployer la partie exécution sur des objets intelligents.

Intégration des systèmes d'automatisation classiques dans le Web: Une grande majorité des bâtiments équipés utilisent des systèmes d'automatisation classiques tels que KNX ou EnOcean. Ces systèmes requièrent d'être intégrés comme partie prenante du Web afin d'assurer une compatibilité avec les nouveaux réseaux de capteurs. Pour cela, nous avons conçu et déployé avec succès des passerelles intelligentes, dont le rôle est d'offrir sous forme de APIs REST un accès aux fonctionnalités des protocoles classiques.

Optimisation de l'impact énergétique des notifications: Les notifications sont le principal mode de communication entre les capteurs et actionneurs. Elles sont donc le principal facteur influençant la consommation énergétique des objets intelligents. Nous avons analysé l'impact énergétique des différentes techniques de notification. En fonction de ces résultats, nous sommes en mesure d'optimiser leur utilisation, et par conséquent de réduire l'empreinte énergétique.

La Figure 6.3 reflète le plan du contenu. Le premier chapitre introduit cette thèse en fournissant une vue d'ensemble du contexte et énonce les questions scientifiques qui seront abordées. Les chapitres deux et trois présentent l'état de l'art lié aux systèmes d'automatisation du bâtiment ainsi qu'aux nouvelles tendances d'utiliser des technologies Web dans les réseaux de capteurs. Le chapitre quatre est la principale contribution de cette thèse, présentant notre architecture du Web des Bâtiments. Cette architecture est ensuite étendue dans le chapitre cinq afin d'intégrer les réseaux classiques d'automatisation et optimisée du point de vue énergétique.

Chapitre 2 - Contexte et Motivations

L'analyse globale de l'état de l'art nous a permis d'observer que les systèmes de contrôle du bâtiment KNX, EnOcean et BACnet fonctionnent avec des piles de protocoles différentes et majoritairement incompatibles. En comparant ces technologies, il ressort qu'elles ne visent pas les mêmes tâches dans un bâtiment, comme illustré sur la Figure 6.4. BACnet est plus orienté vers le management de bâtiments en proposant des services facilitant l'échange de données avec des applications d'entreprises, ainsi que la gestion des alarmes. Le fonctionnement du réseau de terrain qui comporte les capteurs et actionneurs n'est que peu précisé par BACnet. Ceci

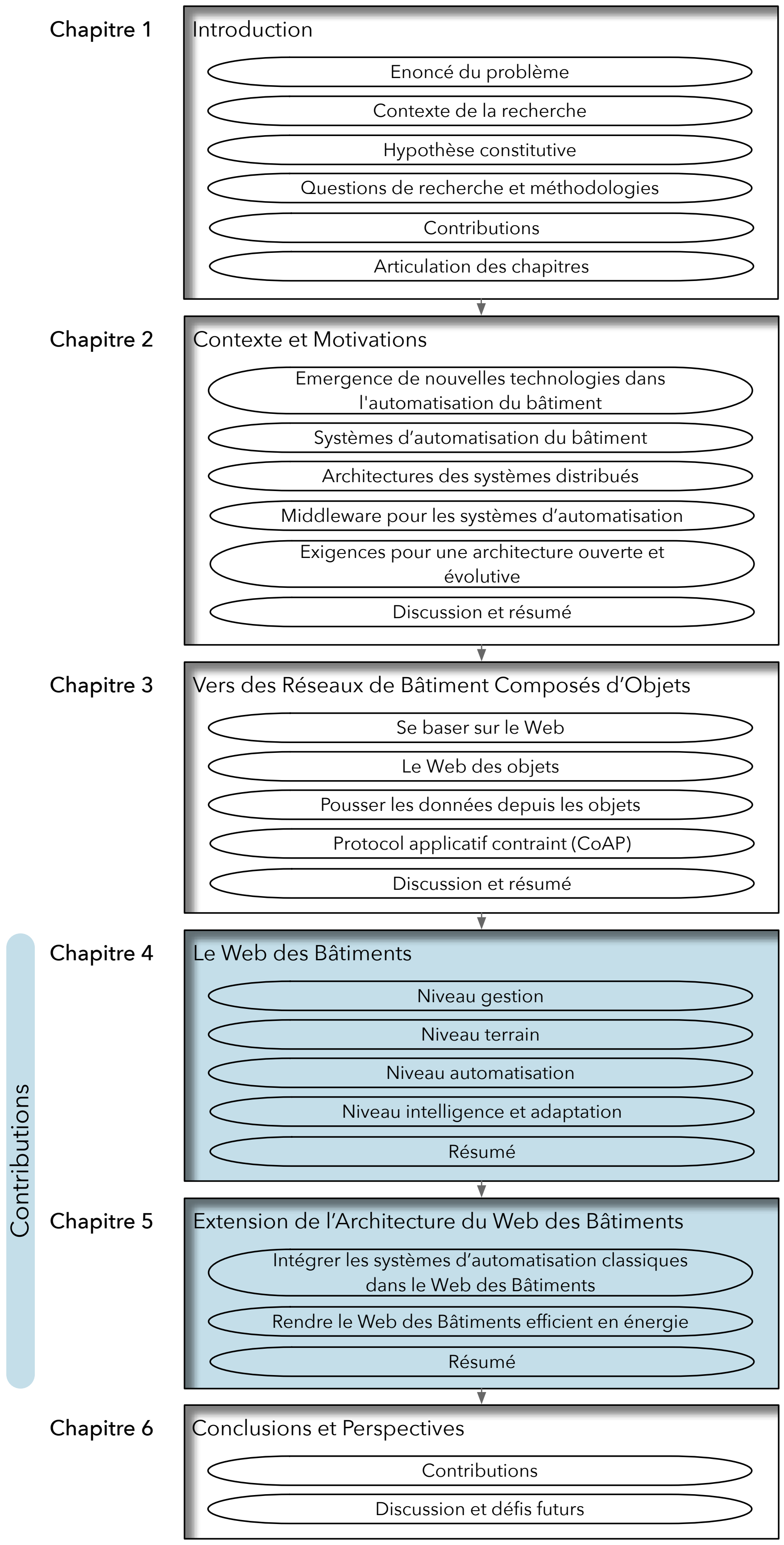

Chapitre 1
Introduction
Enoncé du problème
Contexte de la recherche
Hypothèse constitutive
Questions de recherche et méthodologies
Contributions
Articulation des chapitres
Chapitre 2
Contexte et Motivations
Emergence de nouvelles technologies dans l'automatisation du bâtiment
Systèmes d'automatisation du bâtiment
Architectures des systèmes distribués
Middleware pour les systèmes d'automatisation
Exigences pour une architecture ouverte et évolutive
Discussion et résumé
Chapitre 3
Vers des Réseaux de Bâtiment Composés d'Objets
Se baser sur le Web
Le Web des objets
Pousser les données depuis les objets
Protocol applicatif contraint (CoAP)
Discussion et résumé
Contributions
Chapitre 4
Le Web des Bâtiments
Niveau gestion
Niveau terrain
Niveau automatisation
Niveau intelligence et adaptation
Résumé
Chapitre 5
Extension de l'Architecture du Web des Bâtiments
Intégrer les systèmes d'automatisation classiques dans le Web des Bâtiments
Rendre le Web des Bâtiments efficient en énergie
Résumé
Chapitre 6
Conclusions et Perspectives
Contributions
Discussion et défis futurs

Figure 6.3: Plan de la thèse.

est tout autre pour KNX qui se focalise sur des technologies permettant de relier entre eux les composants, tout en offrant également des fonctionnalités liées à l'automatisation. EnOcean se contente principalement de fournir des capteurs et actionneurs auto-alimentés communicant à travers un réseau sans fil propriétaire. En outre, les technologies de l'Internet des Objets se contentent de fournir une connectivité de terrain, et ne proposent aucune solution touchant les niveaux supérieurs. Bien qu'étant fondamentalement différents, ces systèmes ont un point commun en représentant les valeurs des appareils par des point de données. Ces points de données (datapoints) décrivent le contenu d'une mesure en indiquant l'unité, les valeurs limites, ainsi que la granularité. Ce concept des systèmes d'automatisation sera repris dans notre approche afin d'homogénéiser les données, et de ce fait faciliter les échanges entre différentes technologies.

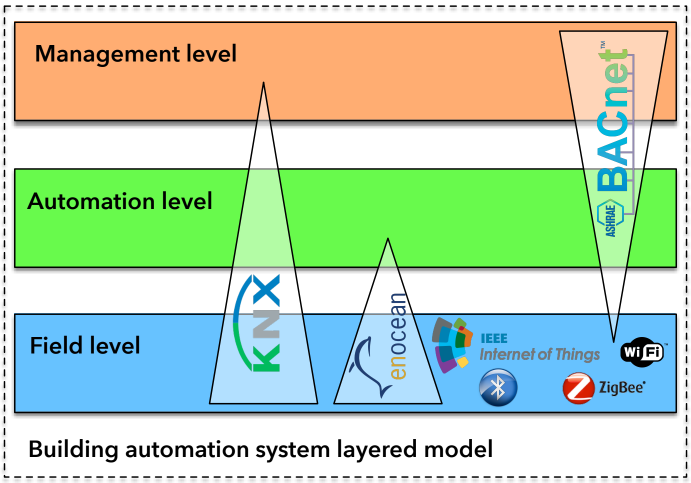


Figure 6.4: Les systèmes d'automatisation du bâtiment sont composés de trois niveaux ou couches. La couche terrain donne accès aux fonctionnalités des appareils. L'interaction entre les appareils est gérée au niveau automatisation, qui s'occupe des stratégies de contrôle (boucles de réglage et ensemble de règles). Finalement, la couche management offre des interfaces de configuration pour la mise en service, ainsi que le dialogue avec des systèmes d'entreprise.

L'étude des architectures de systèmes distribués que sont les réseaux de capteurs s'est focalisée sur les approches suivantes : orientée objet, orientée service et orientée ressource. Les principales différences entre ces architectures résident dans le couplage, l'exposition des points finaux d'accès, ainsi que dans la réutilisabilité des entités. Dans les architectures orientées objet (OOA), les entités sont des objets distribués sur les différentes machines, que ce soit de manière prédéfinie ou dynamiquement. Les appels de fonctions sur ces objets représentent quand à eux les points finaux, et induisent un fort couplage dû aux dépendances entre les objets. Les architectures orientées service (SOA) sont plus flexibles et permettent de découpler les technologies en exposant des services consommables par n'importe quel client à travers des messages. Un seul point final est exposé, permettant de consommer différents services en fonction des informations contenues dans les messages formalisés avec SOAP. WSDL est utilisé afin de décrire le point final ainsi que les différents services exposés. Le protocole HTTP sert généralement à encapsuler l'appel de service. Pour terminer, les architectures orientées ressource (ROA) se démarquent par leur faible couplage ainsi que par le fait de ne requérir aucun système de description. Chaque fonctionnalité ou donnée est exposée comme point final adressable individuellement par une URI. Les interactions s'effectuent à travers une interface de programmation (API) standardisée. Cette architecture est à la base du Web et permet de manipuler directement les données plutôt que de se focaliser sur des services. Dans la Table 6.6 nous comparons ces différentes architectures en fonction de différents critères liés aux bâtiments intelligents. Il en ressort que les architectures orientées ressource sont préférables dans le contexte des réseaux de capteurs, essentiellement grâce à leur faible couplage et leur compatibilité avec les objets intelligents.

Caractéristique	OOA	SOA	ROA
Granularité	objects	services	données (ressources)
Objectif principal	empaquetage	création de charge utile	adressage (URI)
Adressage	instance d'objet	point final unique	ressource individuelle
API	spécifique au langage	spécifique à l'application	générique
Légèreté	+	++	+++
Extensibilité	+++	++	+++
Faible couplage	+	++	+++
Aptitude au nuage	+	++	+++
Facilité d'utilisation	++	+	+++
Aptitude à l'automation	++	++	+
Efficience énergétique	++	+	++
Outils à disposition	+++	++	+
Sécurité	+++	++	++
Verbosité	+	+++	++

Table 6.6: Comparaison des caractéristiques des différentes architectures (+ signifie faible, ++ signifie moyen et +++ signifie haut).

Le principal défi dans les réseaux de capteurs est l'intégration de différentes technologies dans une seule application. Dans ce but, les middleware représentent une approche intéressante en réduisant l'écart entre les exigences applicatives de haut niveau et l'accès aux fonctionnalités de base des réseaux de capteurs. Au cours des dernières années, plusieurs travaux ont été entrepris dans le but de fournir un middleware pour les systèmes d'automatisation de bâtiments hétérogènes. Certains de ces middleware se focalisent sur une architecture orientée objets, qui dépend de certaines technologies. Plus récemment, des solutions se basant sur une approche orientée Web ont fait leur apparition. Malgré une plus grande ouverture et un plus faible couplage, ceux-ci utilisent SOAP et WSDL, ce qui limite leur déploiement sur du matériel embarqué. D'autres solutions ont pris une approche orientée resource, ce qui facilite l'intégration avec un faible couplage de différents capteurs et actionneurs. Cependant, ces solutions s'éloignent des principes fondamentaux ROA en ajoutant une couche service XML comparable à du SOAP afin de définir le type de points finaux.

Dans cette thèse nous avons cherché à fournir une nouvelle approche pour le développement de systèmes d'automatisation des bâtiments. Notre objectif est de réduire la complexité d'applications de contrôle en fournissant un framework faisant abstraction des spécificités de chaque technologie afin d'obtenir un langage unifié. L'approche retenue ici est de se positionner en tant que développeur ou utilisateur du système. En partant de ces points de vue, nous minimisons les connaissances nécessaires afin d'augmenter l'acceptation de tels systèmes non seulement dans les bâtiments, mais également dans les maisons et appartements. Pour cela, nous nous appuyons sur les nouvelles tendances de déployer des services à l'intérieur du réseau, sous forme de nuage local, supprimant le besoin d'une infrastructure dédiée de serveurs. De plus, en rendant ces services auto-organisés, il est possible d'obtenir un comportement dit plug-and-play, évitant ainsi aux développeurs d'applications et aux utilisateurs du système de se soucier de problématiques liées à l'installation et la configuration. En offrant le logiciel en tant que service, notre architecture offre une grande évolutivité tout en étant accessible à un grand nombre de développeurs. Pour cela, nous avons défini un certain nombre d'exigences qui ont pour rôle de guider la conception de notre framework.

Modèle requête-réponse : Les commandes de base sont envoyées pendant le fonctionnement aux dispositifs afin de lire l'état de capteurs, de changer l'état des actionneurs, ou pour récupérer des états d'"applications. Ce modèle est aussi le modèle de base utilisé par HTTP, où les utilisateurs envoient une demande de lecture pour récupérer une page Web, puis écrire le contenu d'un formulaire dans une base de données. Le modèle requête-réponse devrait être préféré lorsque les valeurs doivent être récupérées à un moment précis, par exemple dans une interface graphique statique.

Modèle basée sur les événements : Les événements sont générés occasionnellement par des capteurs lorsque certaines conditions prédéfinies sont remplies (par exemple, dès que la température varie de 1°C). Les algorithmes de contrôle et les applications utilisant l'apprentissage automatique exigent de recueillir des données historiques et de les stocker en vue d'un traitement ultérieur. Dans de tels cas, un modèle basé sur les événements devrait être préféré pour éviter des transferts inutiles de données redondantes sur le réseau.

Légèreté en étant assez léger pour fonctionner sur du matériel ayant peu de capacités mémoire et de calculs, comme des objets intelligents par exemple.

Extensibilité comme potentiellement des milliers de dispositifs vont former un système d'automatisation du bâtiment. La couche applicative avec ses implémentations ne devrait pas être une limitation. Le nombre de connexions parallèles réalisables doit être suffisant pour effectuer toutes les opérations de commande nécessaires.

Faible couplage en assurant une influence minimale sur l'ensemble du système si les composants ou périphériques du réseau doivent être ajoutés ou supprimés. Les systèmes sous-jacentes devraient être échangeables de manière transparente afin de garantir une évolutivité à long terme.

Aptitude au nuage par la construction du système en réutilisant les infrastructures existantes, et de ce fait les capacités de calcul et de stockage des périphériques disponibles tels que les capteurs et les passerelles. Les développeurs doivent être en mesure de déployer des applications de niveau supérieur en tant que service dans un nuage formé par le réseau, faisant abstraction du matériel et des préoccupations de distribution de services.

Facilité d'utilisation afin que les développeurs puissent concevoir rapidement des systèmes de contrôle unifiés sans avoir des connaissances particulières des technologies et protocoles sous-jacents. Ecrire des applications basées sur des systèmes distincts devrait être comparable au développement de mashup Web.

Facilité de mise en place en minimisant l'effort d'installation, suivant le concept plug-and-play et donc augmentant l'expérience utilisateur. Aucune compétence particulière ne devrait être nécessaire pour faire fonctionner les bâtiments intelligents mis à part les étapes de configuration de base abordables par tout utilisateur.

Automatisation adaptative en fournissant une architecture compatible avec de nouveaux algorithmes d'adaptation tel que l'apprentissage automatique. Ces algorithmes sont plus complexes que les règles basées sur des seuils prédéfinis et requièrent généralement une importante quantité de données historiques.

Efficience énergétique étant le but des systèmes d'automatisation du bâtiment, le protocole ne doit pas aller à l'encontre de cet objectif. Comme la consommation d'énergie du système est étroitement liée au nombre de messages transportés par les réseaux, la charge de trafic doit être maintenue aussi faible que possible.

Chapitre 3 - Vers des Réseaux de Bâtiment Composés d'Objets

Le Web est rapidement devenu incontournable afin de partager de l'information entre des utilisateurs situés de part le monde. Cela n'est pas une coïncidence si nous considérons les technologies qui en sont les fondations. Cela commence par un protocole simple d'utilisation étant suffisant pour transporter des documents liés entre eux, permettant une navigation. Un point clé du Web est son indépendance vis à vis des différentes technologies logicielles et du matériel. Il s'agit de ce faible couplage qui a rendu le Web autant intéressant, car les ressources n'ont pas d'influence dans l'intégrité du système dans son ensemble. De plus, le Web est une architecture extensible pouvant intégrer des milliards de ressources. Le protocole HTTP est au cœur du Web. A l'origine, il fut conçu comme un substrat dans le but de développer des système hypermédia distribués pour des documents liés. Historiquement, les pages Web ne présentaient qu'un contenu statique aux utilisateurs. Ce contenu était géré par des professionnels modifiant directement le code de la page. Les évolutions technologiques ont permis de rendre ce contenu plus dynamique en devenant participatif. Le contenu est désormais généré par les utilisateurs du Web eux-mêmes à travers des applications Web 2.0. Beaucoup de ces applications exposent des APIs Web, ce qui ouvre de nouvelles perspectives pour les développeurs dans la création d'applications composites Web. De nos jours, nous pouvons observer que les technologies Web ont une grande importance dans le développement d'applications professionnelles. Le Web s'est récemment transformé en la plus répandue et populaire des plate-formes d'échange d'informations, tout en s'étant étendu vers d'autres domaines : le Web social, le Web temps réel, le Web programmable, le Web sémantique et le Web physique. Le Web des Objets se positionne à l'intersection de ces cinq piliers du Web moderne.

L'Internet des Objets permet à n'importe quel type d'objet (capteur, téléviseur, frigo, etc.) de communiquer du point de vue du réseau (IP), mais ne spécifie pas le protocole applicatif. Comme conséquence, les objets se retrouvent incapables de se comprendre entre eux. Le Web des Objets tente de résoudre ce problème d'incompatibilité en proposant d'utiliser les protocoles du Web au niveau applicatif. Ceci est réalisé en considérant les objets comme composants du Web ayant les mêmes interfaces que les ressources traditionnelles du Web. Les objets exposent leur fonctionnalités sous forme de ressources Web. Le protocole HTTP joue un rôle clé en faisant office de protocole de référence entre tous les participants. En plus d'apporter une certaine consistance au niveau applicatif grâce au faible couplage des connections, ce protocole est omniprésent et peut fonctionner sur n'importe quelle infrastructure basée sur IP. Le Web des Objets envisage également les langages de script du Web et les frameworks comme étant l'avenir des applications. Les applications Web sont usuellement plus simple et plus rapide à développer que des applications traditionnelles. Les mashups (applications composites) se profilent comme une solution idéale afin d'intégrer des objets physiques par l'intermédiaire de HTTP et JavaScript. Plus globalement, le Web des Objets peut être considéré comme un écosystème composé de quatre couches : l'accessibilité, trouvabilité, partage et la composition.

Le Web des Objets se base sur le style architectural REST afin d'homogénéiser les interactions entre les différents objets connectés. REST vise à augmenter l'interopérabilité tout en réduisant le couplage entre les composants d'applications distribuées. Ce style doit être considéré comme une série de lignes directrices et de contraintes facilitant la composition architecturale d'applications. Les applications Web suivant le style REST sont appelées RESTful. REST se distingue des services Web traditionnels (SOAP et WSDL) de différentes manières. Premièrement, il s'agit d'une architecture orientée ressources (ROA) où les fonctionnalités sont représentées par des ressources et non des services. Cet aspect a une influence particulière sur la simplicité et la facilité d'intégration. Cette simplicité évite l'utilisation de langages de description supplémentaires comme WSDL. Ensuite, HTTP est utilisé comme véritable protocole applicatif et non seulement pour transporter du SOAP. Cette simplicité permet aux clients de se focaliser sur les données des ressources plutôt que de devoir interpréter le protocole les encapsulant. Ce concept représentant les fonctionnalités comme des ressources s'applique particulièrement bien dans le domaine des bâtiments intelligents. Les dispositifs d'automatisation offrent généralement des fonctionnalités simples qui peuvent soit être lues ou mises à jour.

REST définit une série de contraintes devant être respectées. La première impose que chaque fonctionnalité exposée en tant que ressource doit pouvoir être adressée explicitement (par exemple la température mesurée par un capteur, ou une valeur enregistrée par un service de calcul) par une adresse unique et globale. Les URI sont utilisées pour cela dans le contexte du Web afin d'offrir une manière simple de les identifier. Le fait d'utiliser le même adressage pour les objets que pour n'importe quel autre type de ressource accessible par le Web permet d'intégrer de manière transparente les capteurs et autres dans le monde du Web. Le Listing 6.1 montre comment des fonctionnalités peuvent être représentées par des URIs.

```
# La température en Celsius mesurée par un capteur
http://device.com/sensors/temperature/celsius

# La mesure numéro 8360 du 2014-10-29
http://storage.com/measures/2014/10/29/8360

# L'autorisation d'accès de l'employé numéro 45
http://access.com/employees/45

# Une liste des mesures disponibles d'un capteur
http://device.com/sensors

# Toutes les mesures du 2014-10-29
http://storage.com/measures/2014/10/29

# Une liste de tous les employés
http://access.com/employees
```

Listing 6.1: Les URIs pointent vers des fonctionnalités ou des ensembles représentés comme ressource Web.

La manipulation des ressources s'effectue à l'aide des "verbes" offerts par HTTP (également appelé méthodes). Cinq d'entre eux sont majoritairement utilisés sur le Web suivant les interactions CRUD (Create, Read, Update et Delete). Ceux-ci sont :

OPTION est probablement l'opération spécifiée par HTTP étant la moins connue. Celle-ci est offerte par la plupart des serveurs Web. Elle peut être utilisée afin de récupérer la liste des opérations étant supportées par une ressource. Cette fonctionnalité est très utile dans les applications ad-hoc dynamiques, car les clients peuvent découvrir à la demande comment interagir avec la ressource.

GET est une opération de lecture seule utilisée pour récupérer la représentation actuelle d'une ressource.

PUT peut être utilisé pour mettre à jour une ressource existante ou alors pour en créer une nouvelle. Le client connaît à l'avance l'URI de la ressource avec laquelle il souhaite interagir (mettre à jour ou créer). L'envoi de plusieurs messages `PUT` identiques n'aura pas d'effet non voulu sur la ressource (la ressource sera créée qu'une seule fois ou mise à jour avec la même valeur).

POST permet de créer une nouvelle ressource alors que l'URI de celle-ci ne peut être déterminée à l'avance. Le serveur se charge de définir une nouvelle URI pour la ressource et retourne cette information dans l'en-tête de la réponse.

DELETE permet de supprimer une ressource. L'envoi de plusieurs messages identiques n'aura pas de conséquence sur la ressource car celle-ci aura déjà été supprimée.

Le protocole HTTP est au cœur du Web, mais il est apparu que celui-ci n'est pas adapté aux objets connectés ni aux réseaux de capteurs de nouvelle génération basés sur le standard 6LoWPAN. Premièrement, HTTP se base sur TCP qui est orienté connexion et non recommandé pour des réseaux mesh ayant généralement une quantité non négligeable de pertes. Les retransmissions provoqueraient une augmentation du trafic ainsi que de la latence. L'autre raison est liée à l'encodage des en-têtes qui sont fournies en texte. Afin de pouvoir utiliser des interfaces REST dans les réseaux de capteurs, le protocole CoAP a été dérivé de HTTP tout en optimisant ses caractéristiques. Il est de ce fait très similaire à HTTP en utilisant des URIs pour identifier des ressources et offre également des opérations identiques sur celles-ci (`GET`, `PUT`, `POST` et `DELETE`). Il se distingue toutefois en se basant sur UDP (possibilité d'activer un contrôle d'échange dans le protocole applicatif) et en encodant les en-têtes en binaire. Le fait d'utiliser UDP ouvre la porte aux communications multicast. CoAP spécifie un mode d'interaction permettant de communiquer simultanément avec plusieurs ressources. Pour cela, les ressources sont attachées à un groupe étant composé d'une adresse IP et d'un port. Grâce à cette approche, un client peut à l'aide d'une requête `GET` récupérer la valeur de plusieurs ressources. La Figure 6.5 illustre comment contrôler plusieurs actionneurs avec CoAP et le multicast.

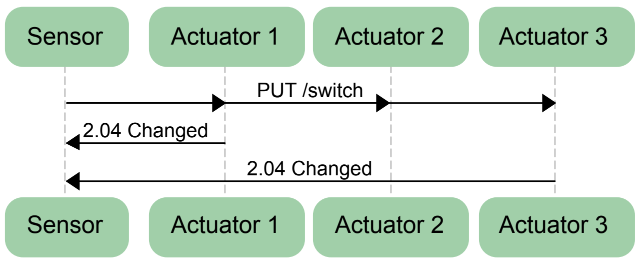


Figure 6.5: Avec la communication de groupe offerte par CoAP, un client peut interagir simultanément avec plusieurs ressources. Le capteur dans cet exemple est captable de contrôler les actionneurs 1 et 3 appartenant au même groupe. Les dispositifs ne faisant pas partie du groupe ne considèrent pas les messages (actionneur 2).

REST et particulièrement HTTP sont prévus pour des communications client-serveur, où les clients envoient une requête afin de récupérer la représentation actuelle d'une ressource. Ce mode de fonctionnement n'est cependant pas compatible avec les interactions faites entres les différents équipements de systèmes d'automatisation du bâtiment. Ceux-ci sont en effet basés sur un système asynchrone de notifications, évitant ainsi l'utilisation du polling. Dans le cadre de cette thèse, nous avons évalué les différentes possibilités permettant de communiquer de manière asynchrone tout en utilisant des protocoles orientés Web : HTTP long polling, HTTP streaming, HTTP callbacks, WebSockets, MQTT et CoAP observation. Le couplage, les timeouts, la latence ainsi que la taille des en-têtes sont des critères éliminant certaines de ces techniques de notification. Nous avons sélectionné HTTP callback, WebSockets ainsi que CoAP observation, illustrés sur la Figure 6.6, comme technologies de référence dans notre travail.

Chapitre 4 - Le Web des Bâtiments

Nous présentons ici notre principale contribution, qui est le Web des Bâtiments, illustrée sur la Figure 6.7. Notre proposition remet en question le modèle en couche classique des systèmes d'automatisation. La séparation des tâches fut définie selon les niveaux de mise en oeuvre de

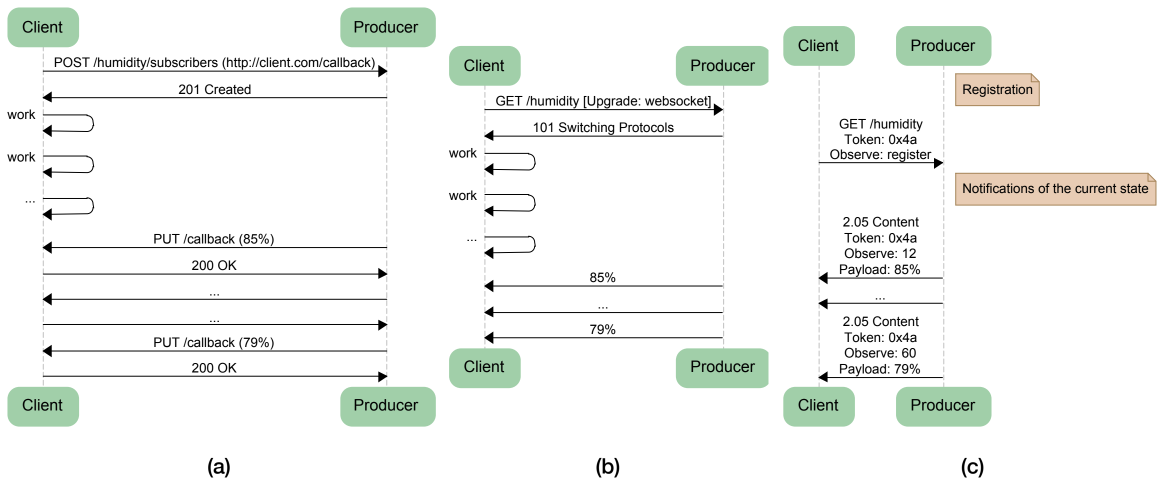


Figure 6.6: Diagrammes de séquences utilisant HTTP et CoAP. (a) le HTTP callback transforme le client en un serveur exposant une ressource de rappel. (b) HTTP est uniquement utilisé pour initier une WebSocket qui va se transformer en une connexion TCP bi-directionnelle. (c) les clients peuvent observer et être notifiés par une réponse CoAP contenant la nouvelle représentation.

chaque couche, ce qui devrait en principe découpler l'architecture. Dans l'approche classique, chaque couche cible des fonctionnalités spécifiques à l'intérieur de ses propres frontières. La complexité des fonctionnalités augmente à mesure que l'on monte dans les couches, ce qui signifie que le niveau de terrain est considéré comme simple, tandis que le niveau de gestion apporte la plus grande valeur au niveau organisationnel. Bien que partant d'une bonne intention, le modèle classique est cependant trop rigide et empêche de créer facilement des applications opportunistes car il est nécessaire d'avoir une importante connaissance du domaine.

Premièrement, nous introduisons une nouvelle couche d'adaptation intelligente qui se chargera d'entraîner des modèles d'apprentissage automatique dans le but d'adapter le bâtiment aux comportements des utilisateurs, par exemple à travers la reconnaissance d'activités. L'exécution des modèle pour sa part peut être placée dans la couche d'automatisation, car elle vise à réagir directement aux changements d'états des capteurs dans le but de réagir à certaines situations. Dans notre point de vue, la couche de gestion requiert une attention particulière et doit être repensée, car elle n'est en l'état actuel pas compatible avec des architectures Web modernes. Cette couche était initialement prévue pour deux rôles : la configuration verticale des couches sous-jacentes, ainsi que pour regrouper toutes les données liées au bâtiment (stockage et analyse). Ces tâches sont uniquement compatibles avec une approche centralisée où la gestion est effectuée par une seule entité. Nous reconsidérons donc l'utilité de cette couche en lui attribuant un autre rôle. Nous préconisons que la couche de gestion doit dans une architecture Web devenir un terrain d'entente pour des services d'intérêt général partagés par toutes les couches, d'où son positionnement vertical.

Nous avons conçu ces blocs pour une infrastructure permettant de développer des applications de gestion intelligente de bâtiments en utilisant exclusivement les standards du Web. Les services Web sont répartis sur les dispositifs en fonction de leurs rôles et forment ensemble un bloc réutilisable par les développeurs. D'un point de vue du développeur, notre architecture abaisse la difficulté et favorise le prototypage rapide d'applications opportunistes. En outre, cet accès direct aux fonctionnalités globales élimine les besoins en matière de logiciel dédié car tout est accessible à travers une API ouverte et suivant une norme.

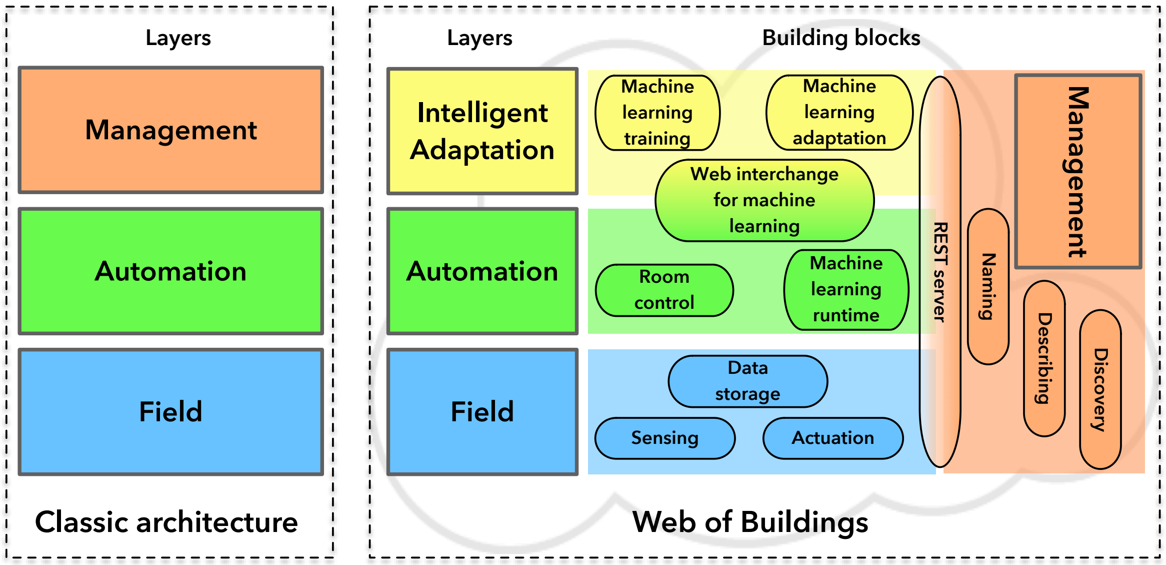


Figure 6.7: Comparaison entre l'architecture en couches classique de l'automatisation du bâtiment et notre contribution du Web des Bâtiments. Chaque couche peut fonctionner indépendamment et est composée d'une série de blocs facilitant le développement d'applications pour les bâtimenrs intelligents.

Couche Gestion

Dans la première couche de notre architecture du Web des bâtiments, nous abordons les fonctionnalités communes qui seront partagées et réutilisées par les autres couches. Cette couche obligatoire représente l'exigence minimale qui doit être mise en place sur les dispositifs. Nous avons opté pour une approche semblable à BACnet imposant l'objet "device" sur tous les appareils. Cependant, nous visons des fonctionnalités minimales plus étendues en incluant des services génériques, qui devront être spécialisés par les autres couches. Notre proposition consiste à offrir une couche générique qui facilitera l'exposition de différents types de fonctionnalités et de capacités compatibles avec le Web. Pour cela, nous proposons un serveur REST réutilisable et hybride (HTTP et CoAP) qui permettra à d'autres couches fonctionnant sur un dispositif d'exposer leurs services sur le Web. Ce serveur REST modulaire est optimisé pour les objets intelligents et inclue la gestion des notifications CoAP observation, HTTP callback et WebSockets. Les développeurs inscrivent leur services en précisant certaines informations (chemin, paramètres, méthode, etc.). Le serveur s'occupe de la gestion transparente des requêtes et réponses HTTP ou CoAP comme présenté sur le Listing 6.2. Les tests de performances ont démontré que CoAP est préférable pour les environnements limités ou le temps de réponse est primordial en étant cinq fois plus rapide que HTTP, tout en supportant plus de communications parallèles.

```
RestServer restServer;

void main(){
  ...
  RestServer_init(&restServer, 80, 5683);

  struct RestService svc_temp;
  svc_temp.method = GET;
  svc_temp.protocol = HTTP|CoAP;
  svc_temp.path = "/temperature";

  RestServer_register_service(&restServer, &svc_temp, temp);
  ...

  while(1){
    RestServer_do(&restServer);
  }
}

void temp (Dictionary* uri_params, Dictionary* params, char* data,
    Response* resp){...}
```

Listing 6.2: Exemple de code montrant comment configurer le serveur REST hybride et enregistrer un service.

En outre, le niveau de gestion propose également un nouveau système de nommage entièrement basé sur des APIs REST. Il a été démontré que le DNS classique n'est pas adapté aux réseaux de capteurs dynamiques et nécessite une connaissance particulière dans sa mise en œuvre. Nous avons donc opté pour une approche peer-to-peer auto-organisée ne demandant aucune installation spécifique. Les dispositifs du réseau de bâtiment se synchronisent entre eux afin de désigner des serveurs de noms. Notre approche est très réactive grâce à la formation automatique de zones de nommage gérées indépendamment. Nous nous basons pour cela sur la morphologie du bâtiment (étages, couloirs, pièces, mobilier, etc.) afin d'établir les noms des appareils. Pour citer un exemple simple, un capteur de température situé dans la salle numéro 23 au deuxième étage d'une organisation pourrait avoir le nom suivant : `temperature.23.2eme.batiment.com`. Les zones de nommages sont générées en fonction du nombre d'hôtes et peuvent dynamiquement fusionner avec d'autres ou être séparées selon les besoins. La Figure 6.8 illustre le découpage d'un bâtiment en zones ainsi que les relations de délégations entre elles. L'évaluation de notre système de zones dynamiques a l'avantage de réduire le trafic dû aux synchronisations et que pour un environnement similaire DNS génère 100% et mDNS 71% de trafic supplémentaire.

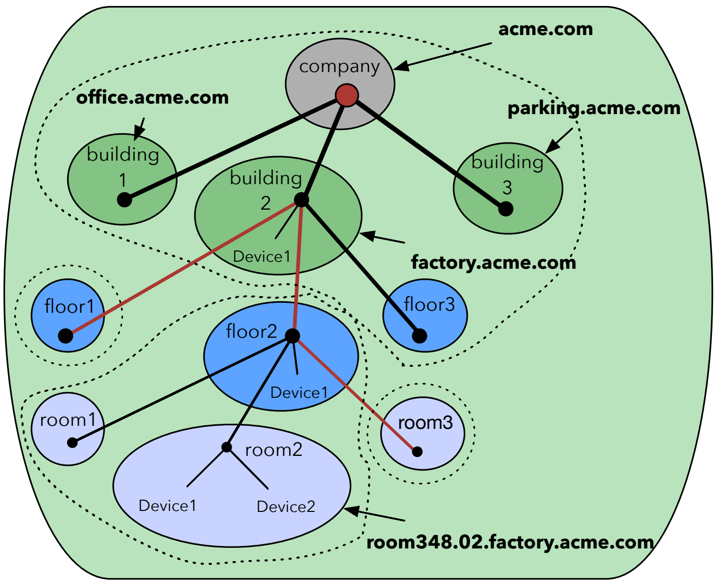


Figure 6.8: Les zones regroupant des lieux contigus sont créées à partir du modèle d'arbre représentant la morphologie d'un bâtiment. Chaque zone gère les entrées de noms pour les lieux dont elle est responsable et se synchronise avec sa zone parente. Les lignes rouges identifient les délégations entre elles.

Dans un réseau de capteurs, il est important de pouvoir identifier quels sont les types d'appareils installés ainsi que les fonctionnalités offertes. Pour cela, nous nous appuyons sur le système de description RDF (Resource Description Framework) provenant du Web sémantique. Celui-ci contient une série de formalismes et langages permettant de décrire des ressources Web. Nous avons choisi cette approche plutôt que de réutiliser les modèles existants (majoritairement basés sur XML) dans le but de proposer une ontologie extensible et se basant uniquement sur des standards du Web. Afin de garantir une extensibilité maximale, notre ontologie est composée à partir de références existantes. Cette-ci est décomposée en plusieurs parties :

Dispositif: Ces propriétés réfèrent à des informations liées au dispositif et qui ne dépendent pas des ressources exposées.

- **Type:** Contient la description de ce qu'est le dispositif en termes de plateforme et logiciel (par exemple le fabricant, modèle et version).
- **Localisation:** Spécifie où le dispositif est géographiquement placé.

Ressource: Ces propriétés décrivent les fonctions des services exposés, leur signification ainsi que comment les consommer.

- **Fonctionnalité:** Permet de comprendre ce que le service offre en termes de fonctionnalité et de données.
- **Porte:** Donne des informations sur comment un service doit être consommé (protocole, paramètres et méthode).
- **Relations:** Fournit les liens vers la ressource parente ainsi que vers les éventuels fils permettant de parcourir les APIs.

RDF propose également une solution élégante afin de rechercher des ressources à l'aide de SPARQL. Ce langage est comparable à du SQL et permet d'effectuer des requêtes sur des documents RDF représentant des descriptions. Nous avons donc décidé d'utiliser ce système afin de découvrir les fonctionnalités d'un réseau de capteurs. Cependant, nous optons pour une approche entièrement distribuée où chaque dispositif est capable d'interpréter une requête SPARQL. Ceci offre l'avantage d'éviter l'utilisation de serveurs centraux stockant les descriptions. Plus important, cela permet de rechercher des dispositifs ou ressources en fonction de propriétés dynamiques changeant fréquemment. Les solutions actuelles exigent des dispositifs de renvoyer leur description au système central à chaque fois qu'une propriété de la description change, générant ainsi une grande quantité de trafic. Nous parvenons à éviter ces échanges en émettant les requêtes en multicast, comme illustré sur la Figure 6.9. Tous les dispositifs du réseau recevront la requête, et l'appliqueront à leurs descriptions RDF. Uniquement les dispositifs ayant des correspondances répondent à la requête. Cette technique permet de diminuer le trafic d'environ 90% par rapport à des approches centralisées.

Couche Terrain

Dans la couche terrain nous adressons la connectivité des objets intelligents en nous basons sur les technologies Web et sur la notion de passerelles intelligentes afin de fournir un protocole transparent et réutilisable dédié aux bâtiments intelligents. Premièrement, nous proposons de compléter notre ontologie avec la notion de datapoint permettant un échange de données interprétables à la volée. Les autres approches constituent des datapoints prédéfinis avec comme propriétés le type de donnée, les valeurs d'intervalle, la sensitivité, la précision ainsi que la nature (pourcentage, valeur de pas, alarme, déclenchement, etc.). Bien que cela permet de standardiser les échanges, le fait de constituer une classe pour chaque datapoint possible ferme la porte à toute évolutivité. En effet, il pourra exister différentes versions de l'ontologie si des classes supplémentaires doivent être ajoutés. Ceci provoquera des incompatibilités entre appareils. Afin

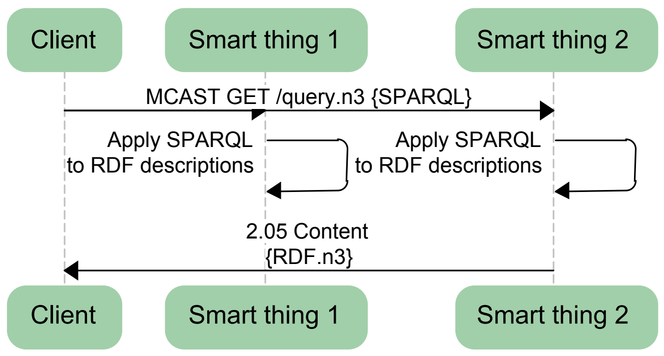

Client
Smart thing 1
Smart thing 2
MCAST GET /query.n3 {SPARQL}
Apply SPARQL to RDF descriptions
Apply SPARQL to RDF descriptions
2.05 Content {RDF.n3}
Client
Smart thing 1
Smart thing 2

Figure 6.9: Les clients émettent leurs requêtes SPARQL sur un groupe multicast prédéfini. Chaque objet intelligent dans le réseau vérifie s'il possède des ressources RDF correspondantes et répondra uniquement dans ce cas.

d'éviter cette limitation, nous avons opté pour une approche plus descriptive. Au lieu de définir ces datapoints, notre ontologie contient des classes et propriétés (visibles sur la Figure 6.10) permettant uniquement de les décrire. De cette façon, un dispositif va indiquer à l'aide de l'ontologie les propriétés de son datapoint. Les clients sont ainsi capable d'interpréter la description à la volée afin de comprendre la signification d'une donnée liée à une mesure ou à de l'actionnement.

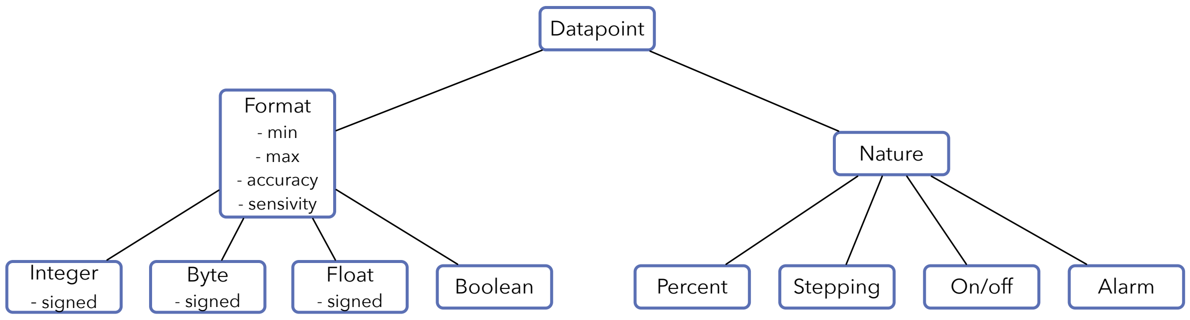


Figure 6.10: Les datapoints ne sont pas prédéfinis mais sont composés à l'aide de différentes propriétés et classes. Cette approche offre plus de flexibilité comparée à un ensemble fini de définitions.

Comme mentionné précédemment, il se peut que certains bâtiment soient équipés d'objets intelligents ainsi que de réseaux d'automatisations classiques. Afin d'offrir une communication transversale, il est nécessaire d'introduire des passerelles intelligentes exposant les fonctionnalités des dispositifs des réseaux classiques sous forme d'APIs REST. Dans notre proposition, pour assurer la transparence, un client n'a pas besoin de connaître l'adresse d'un dispositif sur réseau classique avec lequel il souhaite interagir. Chaque dispositif classique obtient, afin de se conformer à cette exigence, son propre nom et identifiant de manière identique à un objet intelligent nativement compatible IP. Cela signifie qu'il aura sa propre adresse IP pointant vers la passerelle. De toute évidence, l'adresse IP de la passerelle sera enregistrée dans le système de nommage plusieurs fois, mais avec des noms différents. La passerelle n'expose qu'un seul point de terminaison qui accepte tout type de requête avec n'importe quel chemin. Chaque requête est évaluée dynamiquement en utilisant une table de correspondance contenant les noms ainsi que les adresses des dispositifs classiques. De cette manière, chaque dispositif apparaîtra comme supportant nativement le Web des Bâtiments exposant ses fonctionnalités sous formes de services REST. Le fonctionnement générique d'une passerelle intelligente est illustré sur la Figure 6.11.

En appliquant l'architecture présentée précédemment, les clients sont en mesure d'interagir avec des capteurs et des actionneurs de manière simple. De nombreuses applications de contrôle du

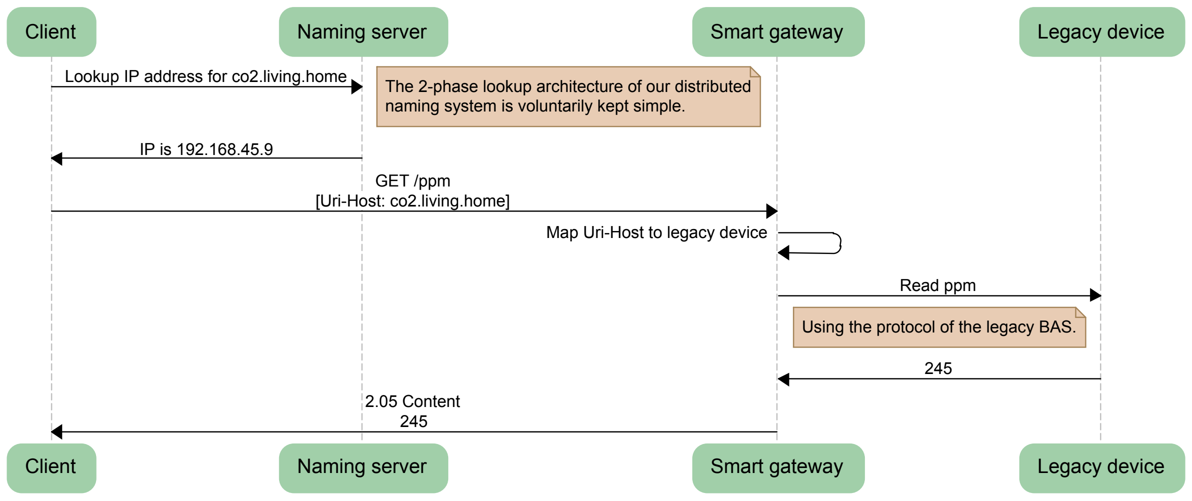

Client
Naming server
Smart gateway
Legacy device
Lookup IP address for co2.living.home
The 2-phase lookup architecture of our distributed naming system is voluntarily kept simple.
IP is 192.168.45.9
GET /ppm
[Uri-Host: co2.living.home]
Map Uri-Host to legacy device
Read ppm
Using the protocol of the legacy BAS.
245
2.05 Content
245
Client
Naming server
Smart gateway
Legacy device

Figure 6.11: Les clients utilisent les mêmes concepts d'interaction avec les dispositifs connectés à des réseaux d'automatisation classiques. La requête sera cependant traitée par une passerelle intelligente faisant la correspondance entre Uri_Host et une adresse. Après avoir communiqué avec le dispositif, la passerelle retournera le résultat dans la réponse HTTP ou CoAP.

bâtiment exigent plus que d'accéder à la valeur actuelle d'un capteur. Les boucles de régulation et les techniques basées sur les données visant une adaptation intelligente du bâtiment en fonction du comportement des utilisateurs font un usage intensif de données historiques. Afin d'éviter de devoir installer des système de stockage dédiés, ainsi que pour éviter de devoir pousser les données à l'extérieur du réseau de capteurs (problématique de sécurisation des données sensibles), nous proposons d'utiliser l'espace de stockage des objets et passerelles intelligents. Dans notre approche, nous considérons le réseau de capteur comme un nuage où chaque dispositif peut contribuer à l'effort global. Nous reprenons pour cela le même principe que pour le nommage, c'est à dire une approche de zones indépendantes, mais cette fois dans le but de gérer un espace de stockage distribué. La constitution des zones de stockage s'effectue en fonction de coûts liés au transfert des données à l'intérieur du réseau. Cette approche permet de diminuer l'impact sur le trafic et, de même manière, la consommation énergétique des objets intelligents.

Chaque zone à l'intérieur du bâtiment est composée d'un coordinateur et de dispositifs de stockage. Le rôle du coordinateur est de gérer un espace de stockage. Il décide quels dispositifs seront responsable de sauvegarder les données d'une ressource, tout en assurant la duplication. En outre, il a la responsabilité pour soit fusionner un espace enfant avec le sien ou alors de diviser son propre espace afin de toujours optimiser le transfert de données. Il est également responsable de servir les requêtes des clients souhaitant récupérer un historique, en agrégeant les données distribuées avant de les retourner au client. Pour leur part, les dispositifs mettront à disposition leur capacité de stockage au service de la zone. Pour cela, ils s'annoncent auprès du coordinateur qui décidera de les faire participer ou non, en fonction des besoins.

Afin d'assurer une redondance des données, au minimum deux dispositifs sont chargés de stocker l'historique d'une ressource. Afin de diminuer le trafic, nous avons opté pour une approche basée sur des notifications mutlicast. Le contrôleur se charge de lier le producteur (ressource à sauvegarder) avec les dispositifs de stockage. A chaque changement d'état de la ressource, le producteur émettra une notification CoAP sur un groupe multicast prédéfini. Seuls les dispositifs ayant été lié avec cette ressource interpréteront la notification et sauvegarderont la représentation. Nous avons pour cela fusionné l'observation et la communication de groupe offertes par CoAP.

Toutes les interactions à l'intérieur d'une zone de stockage, ainsi que pour l'organisation entre elles sont effectuées par des interfaces REST. Les clients désirant récupérer un historique interagissent également à l'aide de requêtes de type REST. Ils ont la possibilité de filtrer les données en précisant un intervalle de temps `...?start={start_date}&end={end_date}`. L'utilisation des API RESTful sur les dispositifs de stockage est une alternative intéressante au SQL. En effet les objets intelligents n'ont généralement pas les capacités pour exécuter des requêtes SQL, essentiellement à cause de leur mémoire limitée. REST permet de normaliser la façon dont les données historiques sont exposées dans les réseaux de capteurs, cela d'une manière très simple, et probablement suffisante pour la plupart des scénarios.

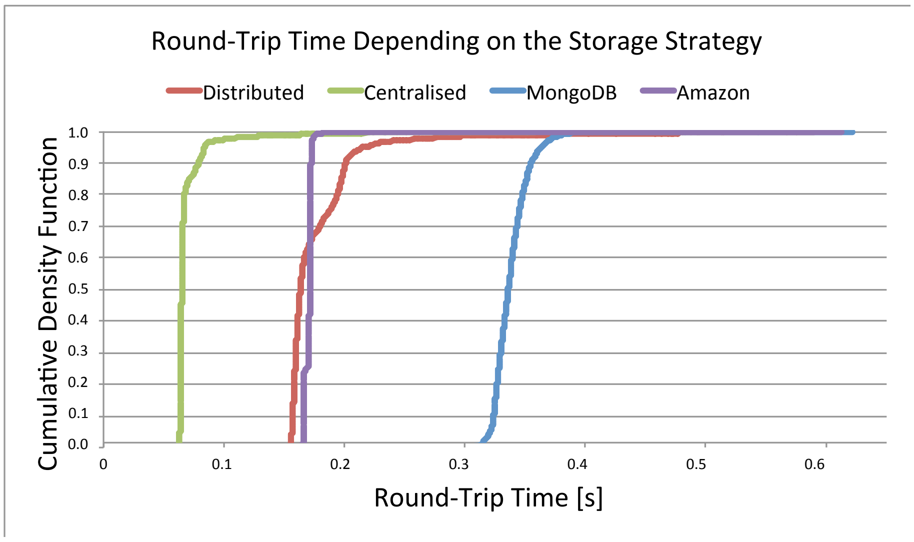


Figure 6.12: Distribution des temps de réponse pour des requêtes consécutives.

Les tests de performance de notre stockage distribué à l'intérieur du réseau de capteurs démontrent que notre approche ainsi que nos choix technologiques sont viables. Nous avons obtenu des temps de réponse similaires, voire même inférieurs à des systèmes de stockage dans le nuage (MongoDB et Amazon SQL). La distribution des temps de réponse pour des requêtes consécutives est illustrée sur la Figure 6.12. Seul un système centralisé local est plus rapide. Concernant, les requêtes parallèles, notre implémentation atteint rapidement ses limites avec environ 50 clients. Cela s'explique par le fait que les objets intelligents servant de stockage distribué dans notre infrastructure de test n'ont pas de système de threading, et traitent chaque requête l'une après l'autre.

Couche Automatisation

Dans la couche automatisation, nous visons à offrir des services permettant de réagir à certaines situations en fonction des états des capteurs. Nous ne nous focalisons par sur les algorithmes de contrôle classiques (boucles de réglage, ensembles de règles), mais visons des algorithmes d'apprentissage automatique, comme illustré dans la Figure 6.13. Nous nous basons sur des algorithmes génératifs dont le but est la classification de données provenant des capteurs. La sortie de cette classification peut ensuite être utilisée dans des mashups ou des applications d'automatisation. Ces mashups peuvent utiliser plusieurs informations pour sélectionner les actions qui doivent être prises pour atteindre l'état optimal dans le bâtiment. Par exemple, les services externes tels que les prévisions météorologiques permettent d'anticiper les changements de température et d'adapter les seuils de chauffage, ventilation et air conditionnée.

Comme pour le système de noms et le stockage, nous souhaitons utiliser les capacités des objets intelligents pour exécuter des modèles pré-entraînés d'apprentissage automatique. De plus,

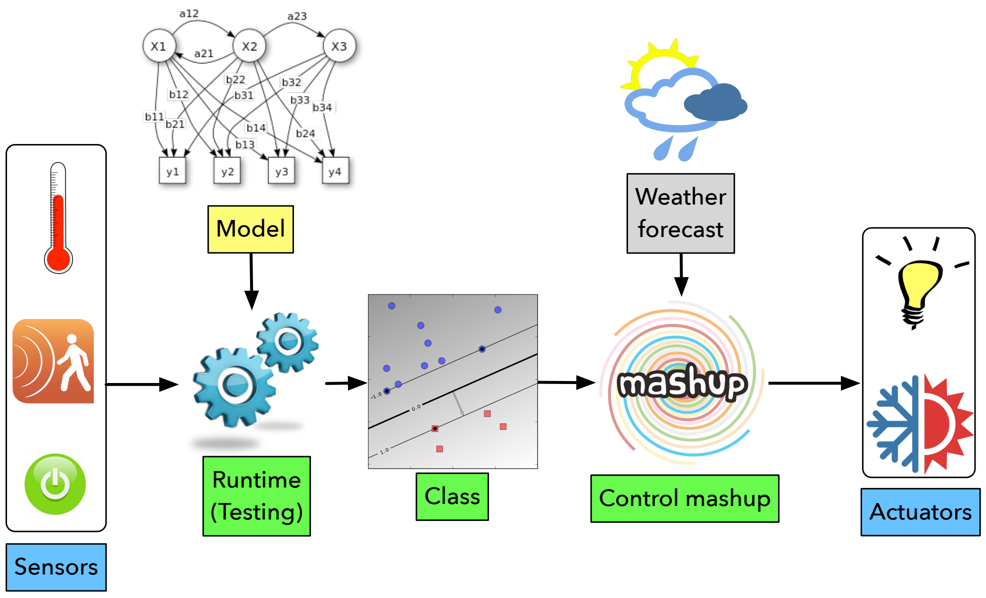


Figure 6.13: Les mashups et applications de contrôle utilisent les résultats de la classification afin de piloter des actionneurs dans le but d'atteindre un certain état dans le bâtiment.

nous visons à rendre l'apprentissage automatique accessible pour des personnes non-expertes du domaine, comme des développeurs d'applications. Afin de se conformer à un processus d'apprentissage automatique visant la classification, nous introduisons deux nouvelles entités, à savoir le *Capteur Virtuel* et la *Classe Virtuelle*. Elles sont destinées à masquer la complexité du processus de décision en exposant des APIs RESTful simples. Ces interfaces contribuent à l'abstraction du moteur d'exécution des algorithmes en proposant un ensemble de composants réutilisables et compréhensibles. L'architecture générale et les interactions entre ces entités sont présentés sur la Figure 6.14.

Capteur Virtuel : Une instance de capteur virtuel est capable de calculer une décision de classification en agrégeant des probabilités calculées par les classes virtuelles. En d'autres termes, une instance de capteur virtuel fusionne les données fournies par d'autres capteurs (physiques ou virtuels) afin d'en extraire de la connaissance, comme par exemple l'appartenance à une classe. Dans notre architecture, une instance de capteur virtuel représente un capteur non physique exploitant des modèles génératifs. Un capteur virtuel peut alors être considéré comme un capteur "de niveau supérieur" émettant une valeur liée à une décision de classe au sens de la loi de Bayes. Dans le réseau de capteurs, ils apparaissent comme des capteurs effectuant des mesures régulières. Cacher la complexité de la fusion de données en agissant comme un capteur classique augmente la ré-utilisabilité et facilite l'intégration au sein de réseaux de capteurs déjà existants. Nous proposons également de dissocier l'API de configuration de l'API d'exécution afin de cacher la partie d'apprentissage automatique.

L'API de configuration est basée sur HTTP pour être accessibles par les applications de haut niveau comme les systèmes d'entraînement résidant sur le réseau de l'entreprise. De nouveaux capteurs virtuels peuvent être créés en émettant une requête de type `PUT` à une adresse du type `http://capteur-virtuel.cuisine.maison/capteurvirtuels/{id}`, qui serait une instance d'agent disponible. Le client doit fournir dans la charge utile les modèles paramétriques en suivant notre format d'échange MaLeX. Par la suite, le rôle du capteur virtuel est alors de distribuer ces modèles sur des objets limités. Il va pour cela interroger les agents potentiellement capables d'exécuter les modèles fournis, devenant des classes virtuelles.

L'API d'exécution représente la sortie de la tâche de classification. La règle de Bayes sera appliquée aux probabilités fournies par les classes virtuelles. Chaque instance de capteur virtuel possède sa propre ressource d'exécution accessible sur CoAP en émettant une requête `GET` à une URI comme `coap://capteur-virtuel.cuisine.maison/{nom_capteur_virtuel}`. Par exemple, un capteur virtuel dédié à la détection d'activités exposerait une ressource `/activite` retournant la classe gagnante qui représente l'activité en cours. Nous avons opté pour CoAP comme protocole applicatif au lieu de HTTP, car CoAP est conçu pour les réseaux de capteurs en étant plus adapté aux dispositifs limités.

Classe Virtuelle : Comme mentionné précédemment, une tâche de classification d'apprentissage automatique repose sur le calcul de plusieurs probabilités de classe. Ces classes possèdent un modèle mathématique comme par exemple un GMM contenant les paramètres de moyenne et les matrices de covariance. Pour simplifier l'interaction avec ces modèles mathématiques complexes, nous appliquons le même principe de haut niveau d'abstraction en les encapsulant dans les classes virtuelles. Suivant le même principe que pour les capteurs virtuels, nous les faisons apparaître comme des capteurs réguliers en décomposant leur interface en deux API distinctes accessibles avec CoAP.

Le déploiement de classes virtuelles est toujours réalisé par un capteur virtuel en interagissant avec différentes ressources exposées sur l'API de configuration des agents de classe virtuelle. La structure de l'API dépend du type d'algorithme que l'agent est capable d'exécuter. L'API contient une ressource pour chaque type de paramètre du modèle, ce qui permet le remplacement de paramètres spécifiques au lieu de l'ensemble du modèle. La première étape est toujours la création de la classe virtuelle en envoyant une requête de type `PUT` vers une URL du type `http://classe-virtuelle.cuisine.maison/classesvirtuelles/{id}` avec les propriétés générales du modèle contenues dans la charge utile. La sous-ressource `/run` doit ensuite être appelée afin de lancer l'exécution de l'algorithme. Pendant cette phase, la classe virtuelle va s'enregistrer comme observateur sur les capteurs utilisés en tant que dimensions d'entrées, et ceci afin d'éviter de devoir continuellement interroger les capteurs. En outre, l'instance de classe virtuelle planifiera l'exécution de l'algorithme en fonction de la propriété d'intervalle du modèle. Les algorithmes HMM et GMM exigent une exécution récurrente à intervalles réguliers même si aucune valeur d'entrée n'a changée.

L'API de classe apparaît de manière similaire à un capteur usuel et expose une ressource pour récupérer la probabilité actuelle. Après chaque exécution de l'algorithme, la nouvelle probabilité est copiée par le moteur d'exécution en tant que nouvelle valeur dans la ressource représentant la classe. Les capteurs virtuels enverront des requêtes `GET` sur une URL du type `http://classe-virtuelle.cuisine.maison/{nom_classe}` pour récupérer la valeur actuelle. Nous avons décidé de rendre cette ressource observable pour avertir les capteurs virtuels en temps réel de la disponibilité d'une nouvelle valeur.

Afin d'évaluer notre proposition, nous nous basons sur un scénario concret de reconnaissance d'appareils en utilisant des mesures de consommation d'électricité. Des capteurs de consommation sont placés entre la prise de l'appareil et la prise électrique murale. Le but du processus de reconnaissance est d'analyser la signature de consommation électrique afin de reconnaître la catégorie de l'appareil, par exemple, une machine à café ainsi que son état d'utilisation, par exemple en veille. Cinq modules Wi-Fi OpenPicus Flyport-PRO embarquent une instance d'agent de classe virtuelle capable d'exécuter des modèles HMMs. Ceux-ci calculent par un algorithme standard de Viterbi les probabilités d'appartenance à une classe ainsi qu'à un état. Une instance de capteur virtuel est exécutée sur un Raspberry Pi. Les tests de performances ont montré qu'une

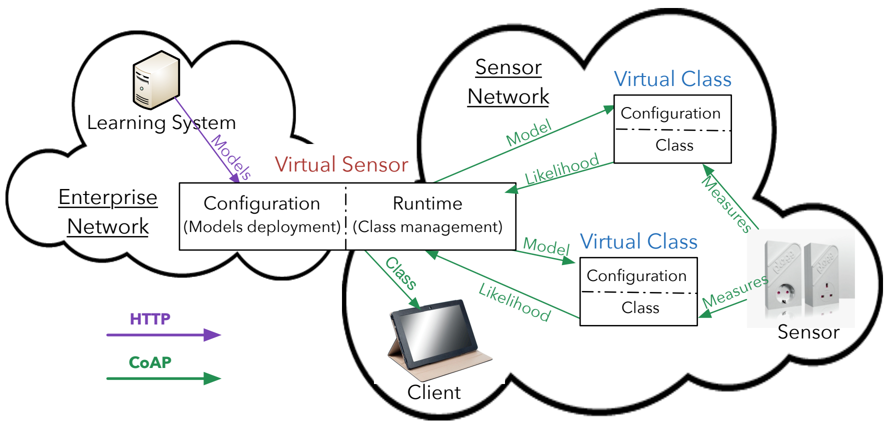


Figure 6.14: Concept d'architecture du *Capteur Virtuel* et de ses *Classes Virtuelles* avec les informations échangées durant l'exécution.

approche événementielle entre les Classes Virtuelles et le Capteur Virtuel est préférable dans un environnement temps réel, avec un temps de réponse moyen de 32 millisecondes. Avec l'approche événementielle, l'architecture est capable de supporter un nombre de clients simultanés supérieur à l'approche à la demande. Avec 100 clients en parallèle, le temps de réponse ne dépasse pas la seconde.

Couche Adaptation Intelligente

La couche d'adaptation intelligente est quant à elle plus conceptuelle et a deux objectifs bien distincts : permettre aux logiciels d'analyse numérique effectuant l'apprentissage d'interagir avec notre architecture du Web des Bâtiments, ainsi que de pouvoir affiner les modèles en fonction des retours utilisateurs. Les algorithmes de type HMM et GMM exigent que les données d'entraînement soient traitées en une seule fois, ce qui empêche un déploiement sur des objets intelligents dû à la grande quantité de mémoire nécessaire.

Dans notre vision, l'apprentissage automatique sera capable de créer des capteurs virtuels ainsi que d'inférer automatiquement des règles d'automatisation qui pourront être exécutées sous forme de mashups. Pour citer un exemple, l'apprentissage automatique pourrait trouver des relations entre les valeurs de capteurs de mouvements et d'illumination ainsi que l'état des interrupteurs de lumière. Cela pourrait aboutir à la gestion d'un ou plusieurs scénarios gérant automatiquement l'éclairage dans une pièce. Cette approche peut être considérée comme une automatisation de l'automatisation du bâtiment, car les scénarios de contrôle sont appris en cours d'opération plutôt que d'être définis manuellement. Le principal avantage réside dans la continuité de l'apprentissage et de l'adaptation automatique à des changements d'habitudes des utilisateurs ou des conditions saisonnières. Les utilisateurs ne sont pas tenus de définir des scénarios de contrôle par eux-mêmes, mais comptent sur l'apprentissage automatique. Sur la Figure 6.15, nous illustrons les interactions nécessaires avec les différents blocs de notre framework.

Les logiciels d'analyse numérique effectuant l'apprentissage des modèles ne peuvent pas interagir directement avec les réseaux de capteurs utilisant des technologies spécifiques comme CoAP. Ces logiciels peuvent cependant intégrer des librairies HTTP permettant de communiquer avec des interfaces REST. Dans notre solution, nous visons à fournir un ensemble de passerelles directement compatibles avec HTTP, ce qui les rend intégrable avec presque tous les logiciels ou langages. Le rôle des passerelles est d'offrir des services Web RESTful accessibles via HTTP, cachant les particularités liées à CoAP et l'utilisation du multicast. A cet effet, nous introduisons

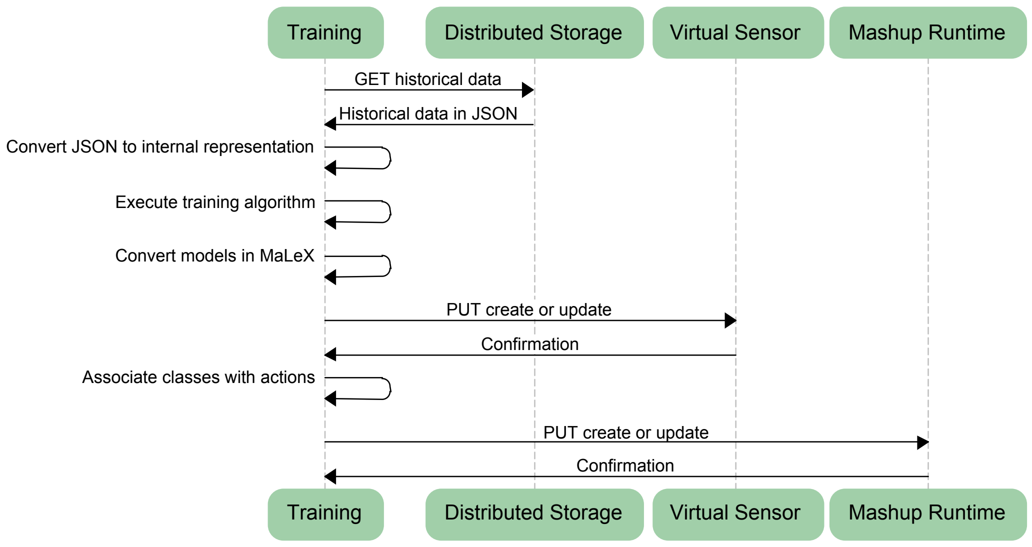


Figure 6.15: La phase d'entraînement de l'apprentissage automatique est positionné au dessus des autres niveaux. Les données historiques des capteurs et actionneurs vont être récupérées à partir du stockage distribué à l'intérieur du réseau de capteurs. Après avoir entraîné les modèles, ceux-ci peuvent être déployés comme Capteur Virtuels et Classes Virtuelles. De plus, la phase d'apprentissage peut également automatiquement définir des scénarios de contrôle sous forme de mashups.

deux passerelles dédiées au stockage et aux mashups:

Passerelle données fournit une API RESTful pour utiliser le stockage distribué à l'intérieur du réseau de capteurs. La passerelle transforme les requêtes HTTT vers CoAP et l'inverse pour les réponses. De plus, elle se charge des interactions multicast n'étant pas possibles avec HTTP, comme par exemple la découverte de services. Cette passerelle expose uniquement deux services : activer le stockage d'une ressource, et récupérer l'historique.

Passerelle mashups permet de gérer des scénarios de contrôle à travers une interface RESTful HTTP cachant également les particularités du protocole CoAP et du multicast. Les scénarios de contrôle sont déployés sur des objets intelligents compatibles.

Considérant une application d'apprentissage automatique fonctionnant avec Matlab, il devient très facile de communiquer avec des capteurs virtuels et avec HTTP. La bibliothèque urlread2 est livrée avec un ensemble de fonctions Matlab pour construire des requêtes HTTP et pour analyser les réponses. La réponse est fournie comme un objet contenant tous les en-têtes et la charge utile. La bibliothèque JSONlab permet quant à elle de manipuler des entités JSON. Le Listing 6.3 montre sous forme d'un exemple comment les deux bibliothèques peuvent être utilisées pour effectuer des requêtes HTTP.

```
[output,extras] = urlread2('http://data-proxy/storage/home/kitchen/temp', '
    GET', '', [], 'from', date1, 'to', date2);
if strfind(extras.firstHeaders.Response, '200 OK')
   json2data = loadjson(output);
   % process the historical data
end
```

Listing 6.3: Extrait de code Matlab utilisant les librairies urlread2 et JSONlab afin de récupérer un historique de données à travers la passerelle de données.

Afin de détecter les actions inopportunes faites par le système, tels que les scénarios de contrôle générés automatiquement, il est nécessaire de donner ce qu'on appelle la "ground truth" comme entrée pour le processus d'entraînement. Dans le cadre de l'automatisation du bâtiment, avec des évaluations correctives, la ground truth sera fournie par les utilisateurs sous forme de réactions contre les choix (classification) faits par la couche d'automatisation. Cela comprend généralement la gestion des systèmes de chauffage/ventilation/air conditionnée et d'éclairage. Nous pouvons illustrer ceci par un exemple concret lié à l'éclairage. Après la phase de collecte de données, les modèles seront entraînés et ensuite exécutés par des objets intelligents. Il pourrait toutefois arriver que les modèles soient inexactes et que de mauvaises décisions soient prises sur l'éclairage. L'utilisateur va réagir à cette situation en agissant sur la commande de l'éclairage. Cette réaction est une information clé indiquant que le modèle conduit à une décision erronée qui doit être relayée au niveau de l'entraînement. Les interactions avec les autres niveaux sont illustrées sur la Figure 6.16.

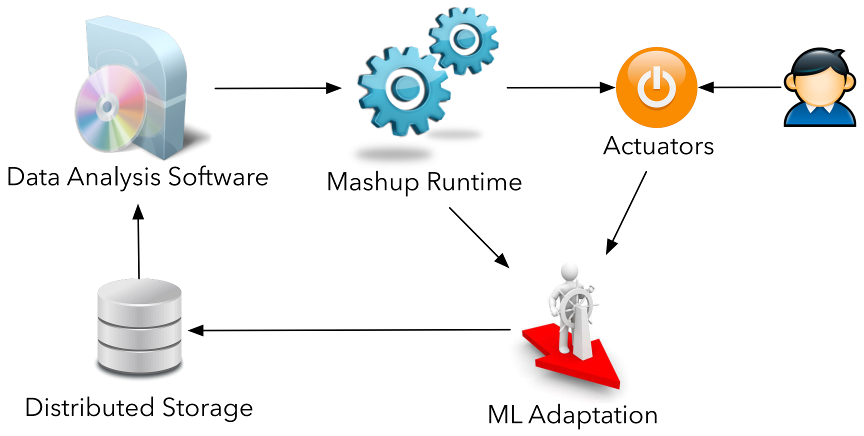


Figure 6.16: La couche d'adaptation va utiliser les états des actionneurs ainsi que les actions effectuées par les scénarios de contrôle pour déterminer si une action a été effectuée par un utilisateur. Les conflits seront notifiés afin que le processus d'entraînement puisse affiner les modèles.

Chapitre 5 - Extension de l'Architecture du Web des Bâtiments

Dans le chapitre précédent, nous avons exposé certains blocs de notre framework du Web des Bâtiments. Alors que les blocs et directives architecturales fournissent un écosystème puissant autour des objets intelligents, cette architecture peut être étendue sous plusieurs angles. Nous pouvons citer par exemple la sécurité et le déploiements à grande échelle entre autres. Dans ce chapitre, nous nous concentrons sur deux aspects liés au Web des Bâtiments, qui nous apparaissent comme les plus importants par rapport à nos exigences.

Les Systèmes d'Automatisation Classiques Dans le Web des Bâtiments

Nous rencontrons de plus en plus le cas où des systèmes d'automatisation classiques (KNX et EnOcean) sont étendus à l'aide de nouvelles technologies, essentiellement des objets intelligents. Afin de pouvoir bénéficier d'une gestion globale du bâtiment, il est nécessaire de rendre ces systèmes compatibles. Comme présenté précédemment, les passerelles intelligentes permettent d'exposer sous forme d'interfaces RESTful les fonctionnalités des systèmes classiques. Nous avons pour cela conçu et déployé dans des environnements existants (bâtiment LESO-PB à l'EPFL) des passerelles intelligentes pour KNX et EnOcean.

Dans notre approche, nous souhaitons faciliter la mise en œuvre de la passerelle intelligente KNX à l'aide du fichier d'export du logiciel ETS, principal outil pour la configuration de réseaux KNX.

Cette archive contient toutes les informations nécessaires afin de pouvoir construire automatiquement des interfaces RESTful liés à des dispositifs KNX. La Figure 6.17 illustre le processus de transformation de cette archive en un fichier XML permettant de faire la correspondance entre une ressource et une fonctionnalité sur un dispositif. Les URLs sont composées de façon dynamique en suivant cette règle : `coap://<group_name>.<location>.<organisation_domain>/<datapoint>`.

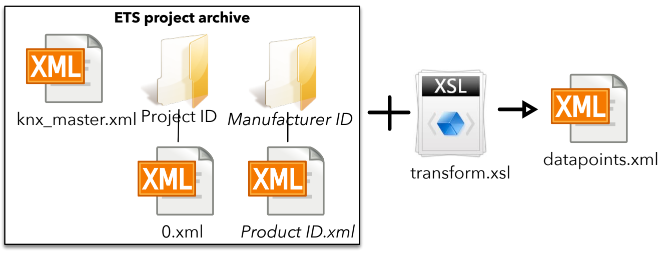


Figure 6.17: Structure de l'archive de projet provenant de ETS et la transformation XSL.

Comme les dispositifs EnOcean ne sont pas gérés par une application centrale mais de manière physique en effectuant un pairage, il n'est pas possible de récupérer cette connaissance afin d'automatiser la configuration de la passerelle intelligente. Nous avons donc créé une application Web offrant aux utilisateurs la possibilité d'ajouter et de configurer des dispositifs (nom, localisation et type de dispositif). Ces informations suffisent à créer sur la passerelle intelligente les ressources donnant accès aux fonctionnalités. Ces fonctionnalités sont accessibles en suivant cette URL générique : `coap://<sensor_name>.<location>.<organisation_domain>/<shortcut>` où `<shortcut>` correspond au nom du datapoint (EEP) EnOcean. En raison du principe de pairage de EnOcean, où les capteurs sont appris sur des actionneurs, notre passerelle doit fournir deux modes de travail: actif et transparent. Ces comportements sont présentés dans la Figure 6.18. Dans le mode actif, la passerelle pilote les actionneurs en fonction des groupes virtuels créés grâce à l'application Web de configuration. Cela simplifie le pairage physique des appareils en nécessitant seulement de jumeler la passerelle avec les actionneurs. Dans ce mode, la passerelle écoute les télégrammes envoyés par les capteurs, et si l'expéditeur d'un télégramme est configuré en mode actif et fait partie d'un groupe, la passerelle va alors retransmettre le télégramme d'origine en remplaçant l'ID de l'expéditeur avec le sien. L'inconvénient de ce mode est d'ajouter de la latence et d'affaiblir le système dans le cas où la passerelle serait défaillante (par exemple problème de matériel, panne de courant, etc.). Dans de tels cas, les utilisateurs ne seront plus en mesure d'actionner (par exemple commuter la lumière ou les stores, etc.). Afin d'éviter ces problèmes, nous avons mis en place un deuxième mode de fonctionnement, le mode transparent. Cette fois, non seulement la passerelle est pairée avec les actionneurs, mais aussi les capteurs. La passerelle et les capteurs sont en mesure de piloter les actionneurs. En ayant un lien direct entre les capteurs et actionneurs, nous pouvons fournir une meilleure expérience utilisateur. Comme pour le mode actif, la passerelle va stocker chaque télégramme capturé pour fournir les valeurs de l'état des capteurs aux clients.

Nous avons testé les deux passerelles intelligentes dans des conditions réelles. Pour cela, nous avons développé des prototypes en Java exécutés sur un RaspberryPi. Les performances (Table 6.7) sont suffisantes pour agir dans des environnement temps réel avec des notifications en dessous de 35 millisecondes. Le nombre de requêtes consécutives pour KNX est limité par le support physique KNX fonctionnant à 9600b/s. De manière générale, les performances des passerelles intelligentes sont proches de celles des objets intelligents. De plus, le Raspberry Pi est une plateforme peu onéreuse est suffisante pour ce genre d'applications. Il est nécessaire de rendre attentif au fait que notre passerelle n'offre pas la possibilité d'actionner des dispositifs EnOcean. Cela

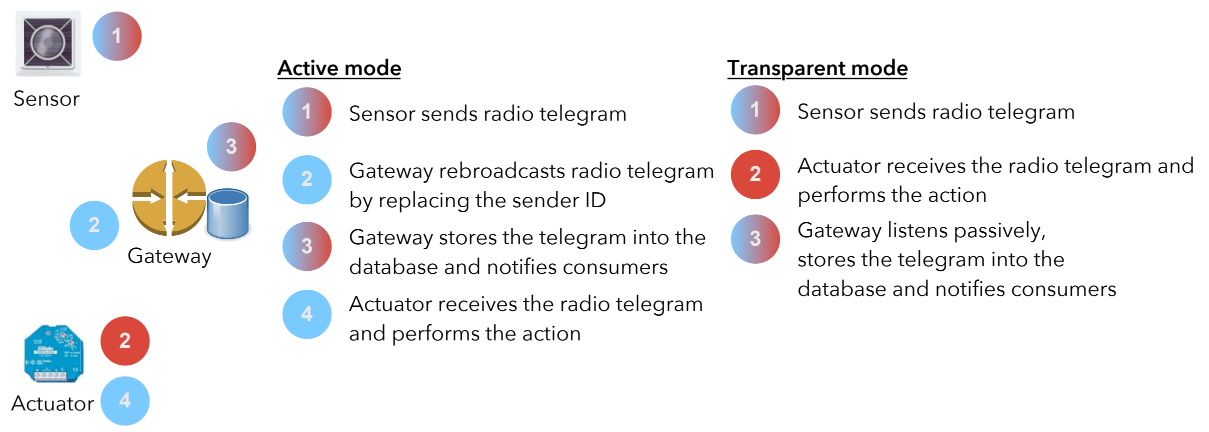

Sensor
Gateway
Actuator
Active mode
1 Sensor sends radio telegram
2 Gateway rebroadcasts radio telegram by replacing the sender ID
3 Gateway stores the telegram into the database and notifies consumers
4 Actuator receives the radio telegram and performs the action
Transparent mode
1 Sensor sends radio telegram
2 Actuator receives the radio telegram and performs the action
3 Gateway listens passively, stores the telegram into the database and notifies consumers

Figure 6.18: La passerelle intelligente EnOcean peut fonctionner dans deux modes d'opération. Dans le mode actif, la actionneurs seront uniquement pairés avec la passerelle. Dans le mode transparent, la passerelle ainsi que les capteurs sont pairés avec les actionneurs.

est dû au protocole EnOcean ne permettant pas de définir quelle donnée d'un télégramme est utilisée par l'actionneur.

Measure type	KNX	EnOcean
Traitement du fichier archive ETS	30 [min]	-
Nombre maximum de requêtes consécutives CoAP par seconde	45	82
Temps de réponse moyen pour 500 requêtes consécutives	22 [ms]	12 [ms]
Nombre maximum de requêtes simultanées	620	631
Temps moyen de réaction pour 500 notifications	33 [ms]	29 [ms]

Table 6.7: Performances des passerelles intelligentes pour une installation KNX avec 265 dispositifs (bâtiment LESO-PB à l'EPFL) et EnOcean avec 14 dispositifs (bureau à la HEIA-FR).

Le Web des Bâtiments et l'Efficience Énergétique

Dans un premier temps, nous avons déterminé l'impact énergétique des différents protocoles utilisés pour les notifications (WebSocket, HTTP callback et CoAP observation). Ceci a été réalisé en émettant depuis un module Wi-Fi OpenPicus Flyport des notifications en direction d'un client. La quantité de charge utile ainsi que la fréquence ont été variées pour simuler un capteur de flux d'air. De la Figure 6.19, nous pouvons observer que HTTP est le plus énergivore. WebSocket consomme en moyenne 3.98% de moins que HTTP, mais seulement 0.69% de moins que CoAP. Le gain maximal de WebSocket par rapport à HTTP est de 9.52%. CoAP est autant performant que WebSocket, en étant en moyenne 3.76% plus efficace que HTTP.

Pour déterminer quelle méthode est la plus efficace dans des conditions particulières, nous proposons pour ce faire de construire des modèles mathématiques permettant de calculer la consommation d'énergie. Pour WebSocket et CoAP, nous pouvons exprimer des formules à partir de la composition d'une trame Wi-Fi. Concernant HTTP, cela est plus compliqué à cause des procédures de négociation de la fenêtre TCP, et dépend de plusieurs paramètres non maîtrisables. Nous sommes donc passé par un modèle paramétrique où les paramètres sont définis en fonction des mesures.

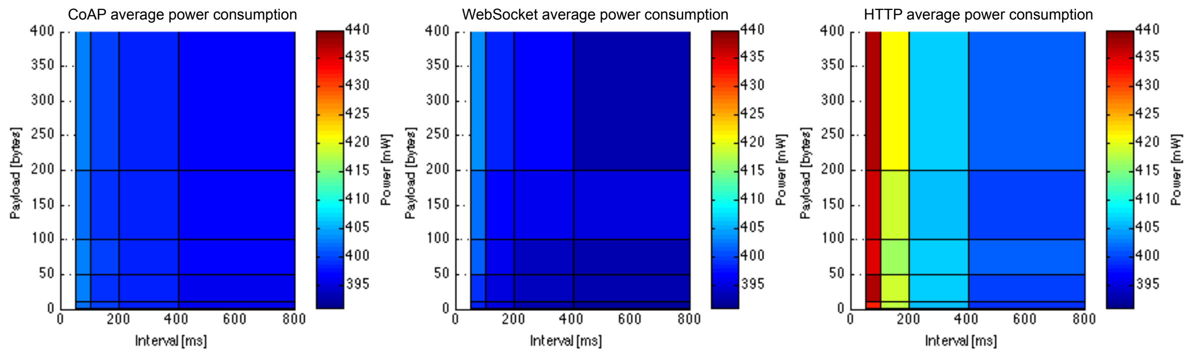


Figure 6.19: Mesures de la consommation d'énergie moyenne pour CoAP observation, WebSocket et HTTP callback sur un module Wi-Fi Flyport.

WebSocket *E(payload) = (PLCP_preamble + (MAC_header + IP_header + TCP_header + WebSocket_header + payload) * ByteRate) * TransmitPower*

En calculant les valeurs de consommation théoriques, l'erreur moyenne est de 0.86% par rapport aux mesures.

HTTP *P(interval) = a * exp(b * interval) + c * exp(d * interval)*

Les paramètres a, b, c et d ont été trouvés par un algorithme d'ajustement numérique pour chaque possibilité de charge utile (1, 10, 50, 100, 200 et 400), donnant six fonction distinctes. En calculant les valeurs de consommation théoriques, l'erreur moyenne est de 0.05% par rapport aux mesures.

CoAP *E (payload) = (PLCP_preamble + (MAC_header + IP_header + UDP_header + CoAP_header + payload) * ByteRate) * TransmitPower*

En calculant les valeurs de consommation théoriques, l'erreur moyenne est de 1.44% par rapport aux mesures.

Nous avons par la suite utilisé ces formules afin de déterminer durant l'opération d'un objet intelligent quel protocole est le plus adapté en fonction de certains paramètres. Pour cela, notre serveur hybride REST a été étendu avec un module intelligent de sélection dont le but est de calculer l'impact énergétique de chaque protocole, et de basculer sur celui consommant le moins d'énergie. Le module intelligent de sélection a prouvé son utilité en permettant d'économiser jusqu'à 6,17% d'énergie dans le meilleur des cas et de 2,10% en moyenne avec WebSocket et HTTP. Lors de l'utilisation de CoAP au lieu de HTTP, les résultats sont beaucoup moins impressionnant, car WebSocket et CoAP sont très proches en terme de consommation, laissant très peu de place pour l'optimisation. Le module intelligent de sélection choisit également le meilleur protocole pour des intervalles supérieurs à 10 secondes, évitant ainsi WebSocket et ses coûts liés aux contrôle de connexion. Nos économies d'énergie étant assez faibles pour un seul client, elles deviennent beaucoup plus intéressantes avec plusieurs client. En effet, dans le cas de plusieurs clients enregistrés (limité à trois dans notre cas en raison des limitations de mémoire du module Flyport), les gains ont été presque multipliés par le nombre de clients, jusqu'à 15% dans le meilleur des cas.

Une autre approche permettant d'économiser de l'énergie consiste à émettre des notifications en mutlicast. Ce principe s'applique principalement dans le contexte où plusieurs clients sont intéressés à être notifiés d'un changement d'état pour une même ressource. Le fait de n'émettre qu'un seul télégramme au lieu de plusieurs peut faire diminuer la consommation énergétique du producteur. Actuellement, les standardisations CoAP pour la communication de groupe et l'observation sont mutuellement exclusives, dû à la problématique de fiabilité (acquittements). En multicast, il est en effet pas possible de savoir en avance quels seront les clients qui potentiellement font partie d'un groupe. Le producteur se retrouve donc dans l'incapacité de déterminer quels clients n'ont pas reçu le message ou n'ont pas répondu avec un accusé de réception. Afin de pallier à cette problématique, nous introduisons de nouveaux en-têtes CoAP permettant au producteur ou à un nœud organisationnel d'organiser les groupes multicast. Chaque client doit s'enregistrer auprès du producteur et fournir un identifiant unique. Grâce à cet identifiant unique ainsi qu'à la liste des clients, le producteur est capable d'identifier lesquels n'ont pas transmis d'accusé de réception lors d'une notification, et peut ainsi gérer les retransmissions.

Pour évaluer l'impact des notifications multicast dans un contexte d'automatisation du bâtiment, nous avons mis en place un environnement de test composé de six modules OpenPicus Flyport. Deux d'entre eux étaient reliés par Ethernet tandis que les autres ont été connectés à un réseau Wi-Fi. Un des modules Wi-Fi a simulé un capteur de flux d'air. Le programme interne du capteur a déclenché un événement à un intervalle moyen de 100 millisecondes. Les cinq autres modules ont agi en qualité d'observateurs. Nous avons mené deux expériences de 24 heures. Dans le premier scénario, nous avons utilisé les notifications multicast suivant notre proposition. Dans la second cas, nous avons utilisé le modèle d'observation unicast classique de CoAP. La fiabilité a été activée dans les deux cas. La Table 6.8 présente les différences entre les deux approches. Nous pouvons observer que le nombre de messages est cinq fois plus élevé lors de l'utilisation de unicast. Ce résultat est logique car une notification séparée est envoyée pour chaque client. Le nombre de notifications reçues par les clients est sensiblement similaire, avec seulement 0.01% de différence. A noter qu'avec les deux méthodes de notification, 100% des messages ont été correctement reçus après les retransmissions. En ce qui concerne la consommation d'énergie, l'expérience a montré une différence de 2,25% entre les deux approches. L'utilisation du multicast permet d'économiser de l'énergie car moins de messages doivent être envoyés. L'antenne de transmission est en effet moins souvent utilisée, ce qui réduit donc la consommation d'énergie.

	Multicast	Unicast
Consommation énergétique [kJ]	34.90	35.68
Nombre de messages	17280	86400
Taux de succès [%]	99.95	99.96
Nombre de retransmissions	8	33
Taux de succès après retransmissions [%]	100	100

Table 6.8: Cette table résume les résultats d'une simulation de 24 heures utilisant l'observation CoAP ainsi que notre proposition de notifications multicast.

Conclusion

Le cœur de cette thèse est la proposition d'une architecture mettant l'accent sur l'homogénéisation de la couche applicative des systèmes d'automatisation du bâtiment. Notre architecture est décomposée en quatre couches que nous avons analysées et conceptualisées en nous appuyant sur les technologies Web. Le design de chaque couche a également fait l'objet d'implémentations prototypes. Ces implémentations ont par la suite été évaluées afin de comparer leurs performances aux références actuelles. Nous présentons dans ce résumé les principaux résultats obtenus qui démontrent que les technologies Web et le style architectural REST sont des technologies facilitant l'intégration de dispositifs ainsi que le développement d'applications composites dans un contexte hétérogène. Nous avons également rappelé les avantages de notre framework mettant l'accent sur la simplicité d'utilisation tout en incluant de nouvelles fonctionnalités provenant de l'apprentissage automatique. Notre proposition d'architecture devra dans l'avenir faire face à d'autres défis. Le domaine de la sécurité est particulièrement important pour garantir que le système d'automatisation ne soit pas sous le contrôle de personnes malintentionnées. En outre, nous manquons de recul par rapport au comportement de nos blocs dans des bâtiments de grande échelle. Pour terminer, l'apprentissage automatique devrait prendre une place plus importante dans le Web, particulièrement en distribuant sur des objets intelligents la phase d'entraînement, ce qui nécessite de revoir certains concepts d'analyse numérique.

Gérôme Bovet

R&D software architect

Rte de Corminboeuf 29
CH-1782 Belfaux
+41 78 788 57 35
gerome.bovet@gmail.com
Born 23.06.1985 (29 years)

Degrees and Education

2012 – March 2015 **PhD in telecommunications**, *Télécom ParisTech, France*, jointly supervised with, University of applied sciences of western Switzerland HES-SO//Fribourg.
Thesis: *A Scalable and Sustainable Web of Buildings Architecture*
Topics: sensor networks, Internet/Web of Things, building automation, autonomous networks, in-network cloud computing, energy efficient networking, machine learning

2012 **MSc in engineering**, *University of applied sciences of western Switzerland HES-SO*.
Major in information and communication technologies

2008 **BSc in engineering**, *College of Engineering and Architecture of Fribourg, Switzerland*.
Specialization in embedded systems

Experiences

2010 – 2012 **Solution architect for mobile systems**, *Logifleet SA*, Le Mont-sur-Lausanne, Switzerland.
Responsible for the architecture of a mobile computing solution. Management of customers projects.

2009 – 2010 **Developer of mobile and location-based systems**, *Novasys SA*, Liebefeld, Switzerland.
Development and solution architecture for a vehicle tracking solution (hardware and software). Consulting for swiss telecommunication companies.

Skills

Computer Science

Concepts	RESTful APIs, IoT, WoT, Semantic Web	Databases	Oracle, SQL Server, MySQL, MongoDB
Programming	Java, C/C++, C#	Web	XHTML/CSS, PHP, Javascript
Protocols	HTTP, CoAP, mDNS, DNS-SD, IPv6, 6LoWPAN	Misc	Git, Maven, knowledge of system and network architecture, as well as Unix-based operating systems.
Building Automation	KNX, EnOcean, BACnet, IP500	Project Management	UML, Agile, Scrum, extreme programming

Languages

English	**Full professional capacity**	*Writing language of the PhD thesis*
French	**Mother tongue**	
German	**Full professional capacity**	*Diploma Deutsch für den Beruf (Level B2)*

Awards

2014 **Best paper award for *Virtual Things for Machine Learning Applications***, *Fifth Workshop on the Web of Things (WoT 2014)*, MIT, Boston.

2009 **3rd place of the European Satellite Navigation Competition with project Assistis**.

2008 **Swiss Engineering prize for the Sphéniscoptère diploma work**, `http://www.muav.ch`.