Diss. ETH No. 19890

# Building Blocks for a Participatory Web of Things: Devices, Infrastructures, and Programming Frameworks

A dissertation submitted to the
ETH Zurich

for the degree of
Doctor of Science

Presented by
**Mihai Vlad Trifa**
Dipl. Ing. EPFL
Born on 12 March 1982, in Oradea, Romania
citizen of Bex (VD), Switzerland

accepted on the recommendation of
Prof. Dr. Friedemann Mattern, examiner
Prof. Dr. Cesare Pautasso, co-examiner
Prof. Dr. Gero Mühl, co-examiner

2011

# Abstract

In less than three decades, computers have evolved from bulky desktop units to tiny, embedded chips that can communicate wirelessly with each other. Battery-powered and stuffed with the latest sensing technologies, these *networked embedded devices* (NEDs) have been pervading our world and are increasingly implanted in our cars, buildings, and cities. From mobile phones, to surveillance systems, to contact-less payment systems – a legion of sentient machines continuously record our behavior. Analyzing the massive amounts of data collected by NEDs would be very valuable in many disciplines.

Unfortunately, the usage of NEDs remains cumbersome because a multitude of hardware and software platforms have been developed – both in academia and industry – and the communication protocols supported by these platforms are often proprietary and incompatible with each other. Without a unified protocol to interconnect the various devices, building and maintaining large-scale applications that integrate NED data and services is an overly complicated process that requires extensive time, expert knowledge, and resources. A common infrastructure that can scale and evolve as new devices are added will be necessary to build a global network of NEDs installed and maintained by different actors. This in turn would lower the access barrier to develop *distributed sensing applications* (DSA) as devices could be shared and used easily. In this thesis, we address the integration problem in large-scale DSAs and explore how to simplify access to data and services of heterogeneous embedded devices.

Nowadays, access to Internet has become cheap and ubiquitous and it is very likely that all sorts of electronic devices will increasingly possess Internet connectivity. Since the mobile Web is rapidly becoming a commodity, this thesis suggests that the existing Web infrastructure and its protocols should be leveraged to exchange data with NEDs. The tremendous success of the World Wide Web as a global platform for easily sharing information and connecting people to

computing services attests that simple and loosely coupled approaches offer a high degree of scalability, robustness, and evolvability. This success is due to a large extent to an open and uniform interface which enabled non-experts to develop interactive hybrid applications rapidly. Unlike most protocols used in embedded computing, Web standards in the Internet are widely used and highly flexible, and in this thesis we suggest that this model would also be beneficial to future embedded computing applications.

Our research bridges the fields of Web technologies and embedded sensing into a unified vision called the *Web of Things* – where the Web's well-known standards and tools are leveraged to seamlessly blend NEDs with the existing Web infrastructure. By drawing upon tools and techniques from both domains, we define the fundamental building blocks of the Web of Things as an extension of the current Web paradigms. After evaluating the limitations of current Web technologies with respect to the requirements of NED applications, we propose practical solutions to alleviate these difficulties to enable the development of efficient, event-driven, and scalable DSAs. Finally, we propose an end-to-end, fully Web-based framework that fosters fast prototyping of distributed sensing applications that run on top of heterogeneous NEDs.

In contrast to existing research in sensor networks, the central question explored in this thesis is how much of the existing Web infrastructure can be reused to accommodate embedded devices. We further examine the common belief that Web standards are inappropriate for building efficient DSAs. Experimental results and prototypes are provided to support the hypothesis that using Web standards for NEDs is possible. Our results further show that the Web is not only a suitable, but actually a desirable medium to build distributed sensing applications that match the requirements for future large-scale sensing systems.

We provide a comprehensive – conceptual and empirical – investigation of the usage of Web standards to exchange information with embedded devices, and the contributions of our work are multiple. First, our results are relevant to the sensor network and pervasive computing communities, as they support the hypothesis that the existing Web ecosystem is sufficient *as is* to build a new generation of scalable and flexible participatory applications on top of heterogeneous NEDs. Second, the Web community at large can build upon our set of guidelines to push the Web into the physical world by integrating devices in the Web fabric, thus making the idea of a Web API for the real world realistic. Third, we explore the practical usage of Web technologies in various contexts, from smart spaces to smart cities, and show that a fully Web-based infrastructure is an excellent basis to build an ecosystem of reconfigurable cyber-physical systems. Finally, we hope the work presented here will serve as inspiration for future Web developers and sensor network researchers. Bridging the gap between these two worlds will very likely shed light upon an unexplored design space to create more potent solutions for important societal problems, from energy-efficient buildings, to catastrophe detection and response systems, to more livable and enjoyable cities.

# Résumé

En moins de trois décennies, les *systèmes embarqués connectés (SEC)* sont devenus omniprésents dans notre monde, car de plus en plus implantés dans nos maisons, voitures, ou dans nos villes. Ces ordinateurs miniature peuvent communiquer entre eux en utilisant différents protocoles radio et possèdent toutes sortes de capteurs qui mesurent une myriade de paramètres (température, pression, mouvements, etc.). Des téléphones mobiles, aux systèmes de surveillance (CCTV), en passant par les systèmes de paiement sans contact – une légion de machines enregistrent en continu nos comportements. L'analyse des quantités massives de données recueillies par ces systèmes serait très utile dans de nombreuses disciplines.

Malheureusement, l'utilisation de ces machines n'est de loin pas à la portée de tout le monde en raison de la variété et de la complexité de plateformes matérielles et logicielles disponibles – tant dans le milieu académique qu'industriel. De plus les protocoles de communication utilisés par ces plates-formes sont souvent fermés ou sous licence, et surtout incompatibles entre eux. Sans un protocole unique pour interconnecter ces dispositifs hétérogènes, le développement et le maintien d'applications à grande échelle qui intègrent les données recueillies par les SEC, resteront des processus complexes qui nécessitent beaucoup de temps, de connaissances, et de ressources. Ces solutions compliquées sont généralement basées sur un couplage étroit entre composants, ce qui entrave fortement la flexibilité et la scalabilité de ces systèmes. De plus, dans la plupart des cas, ces données collectées finissent isolées dans des applications fermées.

Une infrastructure commune qui peut s'adapter et évoluer en ajoutant de nouveaux SEC sera un élément essentiel pour la réalisation d'un réseau mondial de SEC développé et entretenu par différents acteurs. Cette infrastructure faciliterait l'usage des SEC et permettrait à un grand nombre d'utilisateurs de créer leurs propres applications d'analyse de données du monde réel. Dans cette thèse, nous abordons le problème de l'intégration à grande échelle de ces

systèmes afin de comprendre comment simplifier l'accès aux données et aux services offerts par des dispositifs embarqués hétérogènes.

Aujourd'hui, l'accès à Internet est devenu omniprésent et bon marché, de plus il est fortement probable que toutes sortes d'appareils électroniques pourront se connecter à Internet. Étant donné que le Web mobile est sur le point de devenir une commodité, nous suggérons que l'infrastructure existante ainsi que les protocoles du Web peuvent être mis à profit afin d'échanger des données avec les SEC. Le formidable succès du World Wide Web comme plateforme mondiale pour le partage facile d'informations et pour se connecter aisément aux services informatiques atteste qu'une approche simple offre une robustesse et une scalabilité hors pair. Ce succès est dû dans une large mesure à une interface ouverte et homogène qui a permis à des programmateurs novices de développer facilement des applications interactives hybrides. Contrairement à la plupart des protocoles utilisés dans l'informatique embarquée, les standards du Web sont largement utilisés et très flexibles, et dans cette thèse nous suggérons que ce modèle serait également bénéfique pour les futures applications d'informatique embarquée.

La recherche présentée dans ce document de la combinaison des technologies Web et des systèmes embarqués ont donné naissance à une vision appelée le *Web des Objets* où les standards et outils du Web sont mis à profit pour intégrer ces SEC au Web de façon transparente. En s'appuyant sur les outils de ces deux domaines, nous définissons ici les éléments fondamentaux ainsi que des composants concret qui forment le Web des Objets. Après avoir évalué les limites des technologies Web pour des applications en temps réel, nous proposons des solutions pratiques pour compenser ces limitations afin de permettre la mise en place d'applications de captage distribuées, qui sont entièrement basées sur les technologies du Web. Enfin, nous proposons un système complet entièrement basé sur le Web qui favorise le prototypage rapide d'applications qui se basent sur ces capteurs sans fil. Nous discutons l'idée reçue selon laquelle les technologies Web ne sont pas adaptées aux contraintes des réseaux sans fil et qu'il faut choisir des protocoles spécifiquement adaptés et optimisés. Des résultats expérimentaux ainsi que des évaluations de prototypes sont fournis afin de corroborer notre hypothèse : développer des applications de captage distribuées en se basant uniquement sur des technologies Web est non seulement possible, mais serait même souhaitable grâce aux nombreux avantages offerts par les standards Web.

Les implications de notre travail sont multiples. En premier lieu, nos résultats sont utiles aux chercheurs en systèmes embarqués car ils bénéficient de nouveaux outils pour développer et tester de nouvelles applications distribuées rapidement. Grâce à la facilité d'usage des standards Web ainsi que des évaluations et applications concrètes, nous offrons des bases solides qui confirment l'hypothèse que l'écosystème du Web permet de construire une nouvelle génération d'applications participatives qui utilisent des capteurs hétérogènes. En second lieu, la communauté Web au sens large peut aussi se baser sur nos directives en permettant une meilleure intégration du monde physique dans le Web, rendant ainsi l'idée d'une API Web pour le monde réel plausible. En troisième lieu, nous explorons l'utilisation pratique des technologies Web dans

différents contextes, en passant de la domotique aux villes intelligentes, et montrons qu'une infrastructure entièrement basée sur le Web serait une excellente base pour un écosystème reconfigurable de systèmes cyber-physiques. Finalement, nous espérons que les travaux présentés ici pourront servir d'inspiration pour les développeurs Web et chercheurs en réseaux de capteurs. Combler la séparation entre ces deux mondes pourra très probablement donner lieu à la conception de nouvelles solutions plus efficaces pour faire face aux problèmes majeurs de notre sociétés tels que des bâtiments économes en énergie, les systèmes de détection et de réponse lors de catastrophes naturelles et accidents, ou simplement créer des villes plus efficaces, moins gourmandes en énergie, et plus agréables à vivre.

# Acknowledgments

Four years ago, doing a Ph.D. didn't sound *that* hard. After all, many of my friends have done it. What a fool! Looking back at these years, I realize it has been a fantastic journey that has changed me in so many ways. I have learned so many things, met so many people, been to so many places. Even though this thesis is based on the academic convention that prescribes the use of "we" instead of "I", the "we" used throughout this thesis is justified in many occasions, as the research described in these pages would never have existed without the support, help, and contributions of many people, and I will attempt to thank them here.

First and foremost, I would like to thank my advisor Prof. Dr. Friedemann Mattern for being my mentor for all these years. His open mind and trust in me have been instrumental both for my research, but also for my own maturation. He always supported me (and not only financially!) every time I wanted to attend or organize a workshop or a conference, which gave me the chance to circle the globe a few times. His constants "*yes, I think this is a great idea, go ahead*" (when many others would have said "*hmmm, well, not so sure*") have been extremely motivating. Also, I would like to thank my co-advisers, Prof. Dr. Cesare Pautasso and Prof. Dr. Gero Mühl, for their constrictive remarks and useful feedback while writing this thesis.

I would also like to thank my awesome colleague and friend Dominique Guinard for all these years of collaboration and fun. He has always been there stuck in mud next to me – often deeper – yet his everlasting positive attitude and good mood, smile, kindness, and extraterrestrial motivational abilities have been detrimental to help me overcome all the side effects of a PhD, and I cannot thank him enough for all the times he's been there for me. Thanks to all my colleagues and friends from SAP, ETH Zurich, and MIT (there are just way too many to list them all here) for many insightful and constructive discussions: it was great fun working with you.

I was blessed to have had the right amount of liberty to create my own research agenda and to interact with many talented and inspiring people throughout the world that have contributed in their own way to shape my research and ideas. I could never thank everyone that inspired me on these few lines (and I am deeply sorry for those I forgot), but I would like to thank in particular Matt Cottam, Adam Dunkels, Hannes Gassert, Fabien Girardin, Nicolas Nova, J-P. Martin-Flatin, Frédéric Montagut, and Erik Wilde for inspiring me and helping me structure my thesis and thoughts. Thanks also the whole wonderful team of the SENSEable City Lab at MIT, particularly Prof. Carlo Ratti, where I was fortunate to spend half a year.

Then I would also love to thank all the amazing students I had the chance to supervise: Vlatko Davidovski, Michael Haenni, Andreas Kamilaris, Simon Mayer, Oliver Senn, Thomas Pham, Samuel Wieland. They have contributed to much of the implementation of the various components described in this thesis. I can only hope they learned at least a fraction of all I learned by working with these brilliant folks.

I was fortunate enough to have worked as an undergrad with amazing researchers that have taught me the craft of research and hard work (Professors Alcherio Martinoli, Auke Ijspeert, Deborah Estrin, Charles Taylor, and Gordon Cheng), you inspired me so much and I can only hope I have lived up to your expectations. Also thanks to a large bunch of amazing assistants of the DISAL (ex-SWIS) and BIOROB (ex-BIRG) – you know who you are.

Thanks also to all the supporters of the Web of Things for their kind words of encouragement and suggestions. A special thanks goes to our detractors for their critiques (especially the non-constructive ones that gave us even more motivation to work harder!!) and of course to all the readers of our research blog, the WebofThings.com.

Finally, thanks to all my friends for providing me moral support in rough times and for being there to cheer and beer me up. A special thanks goes Claudia for being the greatest teaching assistant for my human-to-human interaction graduate class – I hope I passed the exam.

Above all, I would like to thank my parents and grand-parents for their love and determination to make me a better man, but also for all the sacrifices they made for me. Without their devotion for trying to offer me a better life than they had, I'd never have been arrived where I am today.

Two thumbs up for all of you!!!

# Table of contents

# CHAPTER 1

# Introduction

*"Inventions have long since reached their limit, and I see no hope for further development. "*

Julius Sextus Frontinus (Highly regarded engineer in Rome, 1st century A.D.)

Two decades ago, a world where everyday objects can sense, analyze, store and exchange information existed only in science-fiction novels. Since then, the colossal progress in embedded systems brought into the world a new class of computing devices – *networked embedded devices (NEDs)* – which are tiny computers endowed with wireless communication and various sensors and actuators. NEDs make it possible to build large-scale, *distributed sensing applications (DSAs)* (e.g., facilities and environmental monitoring) with a much finer spatial and temporal resolution than previously possible. DSAs would be an invaluable help for biology, civil engineering and many other disciplines – if only they were simpler to program and to use. Despite the fact that companies building such products are blooming, the majority of academic

and industrial research projects have mainly targeted isolated, small-scale, and homogeneous networks customized for rather specific applications. Because most projects rely on customized and often incompatible hardware or software platforms, integration of different systems is very difficult and costly.

The vision where people, objects, and environments could be augmented with computational capabilities that provide information and services to assist us in everyday tasks was introduced by Mark Weiser [239] and is commonly referred to as *ubiquitous computing*. The pervasive availability and access to sensing and computing services embedded in the world around us changes fundamentally the relationship and interaction between humans and the physical world. From traditional desktop computers, digital services become decentralized and can follow us anywhere by allowing continuous interaction with the computing infrastructure via various interfaces. New computing and interaction paradigms such as context-aware computing [106] or natural interaction [231] are therefore required to enable efficient interaction with digitally augmented environments.

## 1.1 Problem Statement

So far, little attention has been paid to the development of simple and functional open systems for embedded devices that do not require experts that re-implement similar functionalities over and over again. A large amount of work is still devoted to low-level programming and to the creation of application-specific user interfaces, which is a waste of resources that could be devoted to the development of the application logic.

Despite the increasing popularity of the *Maker movement*[1] and the flourishing of open source communities, the materialization of Mark Weiser's vision [239] is still impeded by the lack of clear, simple, and interoperable standards that allow embedded systems to interact smoothly at a global scale. Simple tasks such as discovering dynamically devices present in a particular location and interacting with them in an ad-hoc manner, are unnecessarily complex problems that require customized applications. While various protocols, as for example Universal Plug and Play (UPnP) [58], offer powerful mechanisms for discovering and interacting with electronic appliances on a local network, a common ground on which heterogeneous devices using different protocols could interact globally and transparently is still missing.

A high-level, uniform, and simple protocol to connect to and interact with heterogeneous devices is necessary to enable applications that can turn raw sensor data into social and economic value. Such a protocol should ideally support both automated data acquisition and direct, ad-hoc interaction with embedded devices, as shown in Figure 1.1. Finally, scalable, distributed applications for sensor data collection, processing, and storage should be easy to implement on top of this framework.

[1] See: http://radar.oreilly.com/2008/01/maker-movement-gaining-recogni.html

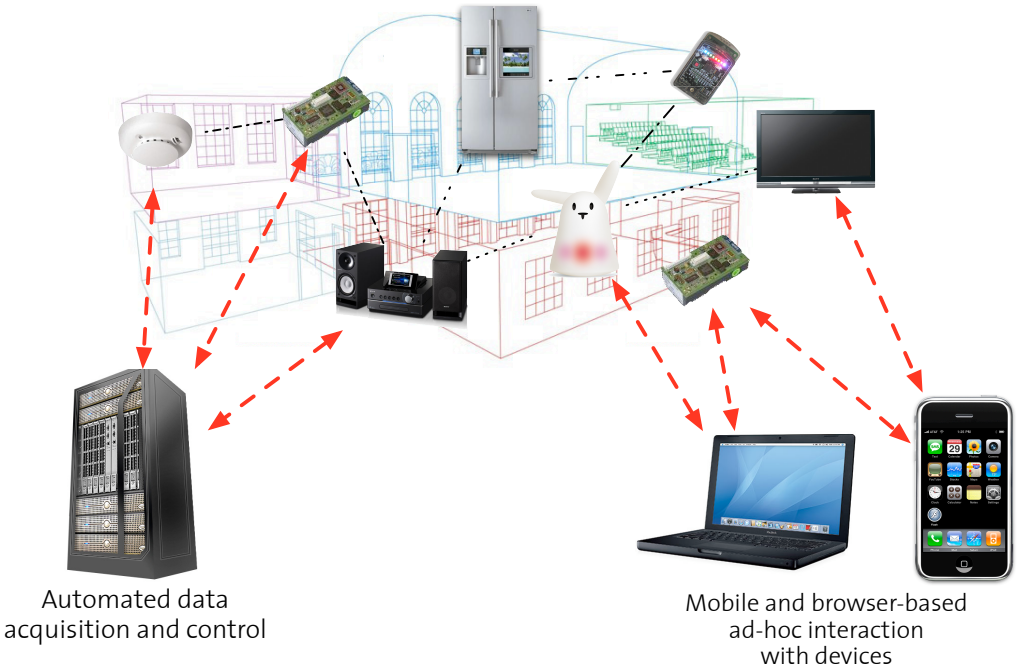



**Figure 1.1:** Emerging heterogeneous sensor network applications require uniform interfaces for both automatic data acquisition and control, as well as for manual interaction.

## 1.2 Methods

Considering the recent progress in mobile communications, Internet access will very likely soon become a commodity accessible from most mobile devices. According to the IP for Smart Objects Alliance (IPSO) [111], an increasing number of embedded devices will be supporting the IP protocol, so that many physical objects will be connected to the Internet. This convergence of physical computing devices and the Internet provides new design opportunities, as digital communication networks will soon not only contain virtual data (images, text, etc.), but also real-time information from the physical world.

Early approaches for developing distributed applications were based on tightly coupled solutions, where each element had full knowledge about the other peers and the functions they offered (so-called RPC architectures). More recently, Web Services standards promoted by organizations such as the World Wide Web Consortium (W3C) or OASIS have become a standard solution for building distributed solutions, and an increasing number of companies are using them for exchanging data across IT applications. Such tightly coupled approaches are suited for interconnecting a few devices in controlled situations with mostly fixed network topologies, but are not flexible enough to handle many resource-constrained and mobile devices with unknown capabilities in dynamic environments.

In the last decade, a newer generation of Web applications – commonly referred to as Web 2.0 [193] – has become highly popular. In contrast to the early days of the Web, the focus has

shifted on the user and user-generated content on the one hand, and on the other hand on a set of technologies (e.g., AJAX, RSS) that support the development of highly interactive interfaces that offer a rich user experience, comparable to common desktop applications. As shown by the high scalability of the Web, simple standards (HTTP/HTML) can support efficient and flexible systems where a large variety of hardware and software platforms coexist and interact smoothly.

Open access to data through Web services has enabled information to be reused across independent systems, therefore has lowered the access barrier that allows people to develop their own composite applications. The tremendous success of several Web 2.0 applications (e.g., wikis, open source communities, blogs, and more recently Web mashups) illustrates the benefits of such an open collaboration and information sharing for building large participatory applications.

Considering the evolution of the Web and the need for a common technical infrastructure to connect sensing devices and other smart physical objects, we propose in this thesis the creation of the *Web of Things* as an evolution of the more abstract Internet of Things [182], which extends the Web to encompass embedded computers and sensors using the same principles and standards. Reusing the open and simple standards such as XML, HTTP, or RSS that made the Internet so successful in order to allow physical devices (e.g., sensor and actuator networks, mobile phones, etc.) to offer their services to the wide public through the Web will allow any device to finally "speak" the same basic language as other resources on the Web. This makes it much easier to integrate physical devices with any other content – resource or service – on the Web which will enable the Internet to reach out into the real world and sense real-time information about the everything around us. In this thesis, we propose the development of tools which facilitate the publishing and processing of data collected by networked embedded devices, and which promote the open sharing of data and hardware infrastructures by maximizing reuse and scalability, while minimizing coupling between participating devices. People, objects, and environments will then become first class citizens of the World Wide Web, which will make them linkable, discoverable, searchable, just like any other Web resource.

To realize the Web of Things vision, we start by dissecting the core architectural principles the modern Web is built upon, and propose a set of patterns for implementing common interaction types and functions commonly used in distributed sensing applications. First, we describe the various methods for connecting NEDs to the Web by reusing the Web model to support the different interaction types required for embedded devices. Afterwards, we design a software framework (gateways) that facilitates the integration of any device into the Web and propose how gateways can be connected together to form a scalable and flexible infrastructure (the backbone) for the Web of Things. We then explore the limitations of the request-response model of HTTP and propose a Web-based messaging mechanism. On top of the infrastructure we implemented WISSPR, a Web-based infrastructure for sensor data stream collection, processing,

sharing, and storage that allows to build end-to-end distributed applications for NEDs solely using Web standards.

## 1.3 Contributions

The goal of this thesis is the development of tools that facilitate the publishing and processing of data collected by networked embedded devices, and promoting the sharing of data and hardware infrastructures by maximizing reuse and scalability, while minimizing coupling between participating devices.

The contributions of this thesis are multiple. First, we provide a comprehensive conceptual and empirical investigation of the usage of Web standards as an interaction protocol for embedded devices to date. Our results are relevant to the sensor network and pervasive computing communities, as we show that the existing Web standards and tools can be used *as is* to build a new generation of open scalable and flexible applications on top of heterogeneous networked embedded devices. Second, the World Wide Web community at large can build upon the set of best practices and guidelines to extend the Web as we know it to the real world, by seamlessly integrating devices in the Web fabric, thus making the idea of a Web API for the real-world realistic. By leveraging recent developments in Web technologies such as real-time Web or the semantic Web, we demonstrate that the Web is not only an appropriate – but actually a highly desirable – medium to build distributed sensing applications that match the requirements for future, large-scale, ad-hoc and heterogeneous sensing systems. In the long run, we expect that extending the existing Internet to naturally integrate physical devices will reshape the Internet into a versatile collection of physical and virtual resources that can be automatically (re)combined at run-time.

## 1.4 Thesis Outline

This thesis is organized as follows: in Chapter 2 we introduce a high-level survey of related work in device integration and data management. Chapter 3 introduces the core architecture of the Web along with the founding principles of the Web of Things. Chapter 4 describes how to Web-enable devices, and in Chapter 5 we explore how to use smart gateways for building scalable infrastructures. In Chapter 6 we present an application framework for sensor data streams management. Finally, we discuss our results and review our contributions in Chapter 7.

# CHAPTER 2

# Background and Motivation

*"The trouble with programmers is that you can never tell what a programmer is doing until it's too late."*

Seymour Cray (1925-1996)

In less than a decade, embedded devices equipped with sensors have become one of the most promising technology for the 21st century [149, 52]. Yet, the true potentials of this technology lies in the interconnection of isolated devices into a single unified network – the *Internet of Things*. Because searching and using particular devices increases in complexity with the size of the network, new approaches to design, deploy, and maintain applications that run on top of a globally distributed collection of devices is needed. In this chapter, we survey and discuss the related work in the field of integration of data and services from embedded devices with applications.

## 2.1 Motivation

Thanks to the recent progress in embedded systems, tiny embedded computers with various sensing and communication capabilities are being increasingly used in many disciplines. From gaming [188], to logistics [142], to urban planning [203], to environmental monitoring [223], embedded sensors and computers have become indispensable tools for collecting and processing real-time data from the physical world. Because tremendous benefits are to be gained from integrating the data generated by a distributed collection of sensors with higher-level applications (business information systems, scientific analysis frameworks, etc.), devices increasingly include wireless communication interfaces such as Bluetooth, ZigBee, or even Wi-Fi to connect with the external world. Most mobile phones nowadays can directly connect to the Web using 3G/EDGE, and probably soon WiMax as well.

### 2.1.1 Networked Embedded Devices (NEDs)

The central elements of this thesis are *networked embedded devices (NEDs)*, which we define as any electronic device with constrained computational power and/or energy resources, equipped with a wireless or wired communication interface, and various sensors or actuators. According to this definition, the various devices shown in Figure 2.1, digital photo frames, mobile phones, remote controls, or networked media players are all NEDs.


**Figure 2.1:** Various *networked embedded devices (NEDs)* that have been used in this thesis.

A *NED application* (also called *pervasive application*) refers then to any software application that uses and combines in some way services offered by NEDs, for example real-time and

historic data from one (or many) energy meter(s) in a house displayed on a digital photo frame. Even though the gap between the physical and virtual world is constantly shrinking [156], interoperability between NEDs from different manufacturers remains limited as no unique standard for connecting physical devices and applications prevails. A wide variety of hardware platforms with diverse capabilities and functionalities have been developed, and often because of commercial reasons most of these device use incompatible communication protocols. As a consequence, applications that integrate data and services from different NEDs are often complex and their development requires much time, money, and expertise. One reason is that for each new project, sensor integration is only the first step and much has to be developed from scratch. In particular, most services and tools used both in the back-end and front-end of such composite applications (for example the event detection system, data processing engine, database, or Web interface) have to be customized for the devices at hand for each project. This results in a waste of resources that could be used to design the application itself, rather than re-implementing components that already exist and could simply be reused.

As a result of recent initiatives such as the Make magazine[1], hackerspaces[2], or FabLabs [129], amateurs have been increasingly empowered to build physical computing applications. Tools such as Processing[3] and Arduino[4] (device No. 1. in Figure 2.1) have lowered the barrier to use and program electronic devices, which enabled a large community of designers and developers to create interactive applications with little technical knowledge. However, large-scale distributed sensing applications will require to reconsider how one designs, builds, and deploys applications that take advantage of many networked resources. Easy to use and versatile standards for interacting with embedded devices will be essential to achieve an uniform, scalable, and robust common ground where diverse devices and services can exchange data efficiently.

After dressing a list of requirements for an ecosystem of distributed devices derived from general use cases and scenarios, in the remainder of this section we present existing solutions for building such applications and discuss their advantages and shortcomings with respect to our requirements.

### 2.1.2 Wireless Sensor Networks (WSNs)

A particular class of NEDs called *Wireless sensor networks* (WSNs), consist of autonomous sensors (also called *nodes*) that monitor certain physical conditions (such as temperature, vibration, pressure or humidity). These sensors are geographically distributed and communicate via a wireless (ad-hoc) network where data is forwarded (often via multiple hops) to a base-station,

[1] Major magazine on practical do-it-yourself (DIY) projects. See: http://makezine.com/

[2] Community-operated spaces enabling amateurs to work on their projects collectively. See http://hackerspaces.org/.

[3] An electronic sketchbook for prototyping multimedia applications. See: http://processing.org.

[4] An open-source hardware platform for physical computing. See: http://arduino.cc.

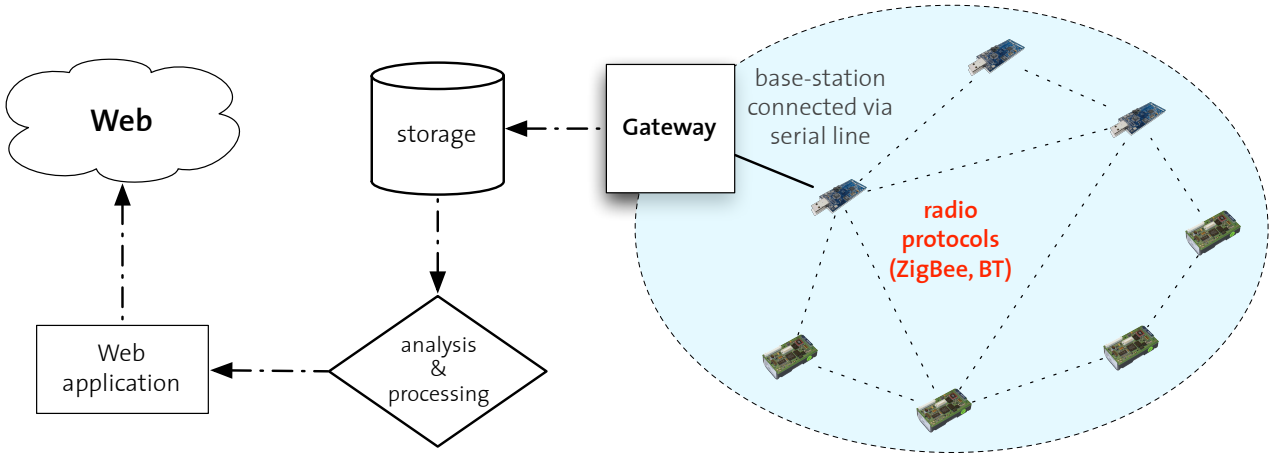



**Figure 2.2:** A typical WSN application. Distributed sensors nodes monitor an environment, communicate with each other using low-power radio communication, and transmit critical information to a base-station connected via a wired connection (USB, I2C, etc), which acts as a gateway that connects the WSN with other applications or the Internet.

as shown in Figure 2.2. WSNs were initially used almost exclusively for scientific or research purposes (see [70, 237, 253] for general surveys about WSNs), but over the last decade they have gradually become invaluable tools in many disciplines from structural health monitoring, to building automation, to industrial automation, or even the smart grid. As this field further develops, there will be a need for simple and standardized automated data acquisition and control mechanisms, but also ad-hoc manual or mobile interaction mechanisms. For most of these applications, interoperability with existing network infrastructures will be vital.

Typical WSN applications were initially based on a *sense, store, and analyze* pattern. Sensor nodes transmit the data they gather to a base-station (also called *sink*) where the data is stored and subsequently analyzed by domain experts, as illustrated in Figure 2.2. A typical example of this pattern is the "Macroscope in the Redwoods" project [221] where sensor nodes were deployed on several redwood trees in California and monitored various environmental factors such as temperature, humidity, and solar radiation over long periods of time.

With the growing usage of WSNs in different areas (e.g., military, environmental, health and home applications), continuous monitoring and immediate event detection became an important requirement as well [70]. As embedded systems keep improving, more processing power and smaller energy use will make possible to monitor environments with higher sampling rates. As a consequence, it is very likely that sensor networks will soon become key instruments to observe large-scale phenomena with high temporal and spatial resolutions.

Much of the existing research in the WSN domain addressed specific problems in small-scale deployments such as minimizing energy consumption [252, 232], routing algorithms and MAC protocols [105], or programming paradigms [76, 179, 247]. Because little attention has been paid so far to higher level issues such as interoperability and simpler programming models,

most projects still operate in isolation are incompatible with each other. However, as concrete use cases for these technologies appear and become financially viable (see Section 2.1.3), new research objectives for the WSN community have appeared, for example support tools for developing, deploying, and debugging large-scale applications. Besides, the need for sharing device data and services with more actors and users also increases, therefore scalability, robustness, and openness are becoming critical requirements [214].

According to the IP for Smart Objects Alliance (IPSO) [111], an increasing number of embedded devices will be supporting the IP protocol. Many physical objects will therefore be able to connect to the Internet, especially as IPv6 and its embedded variant 6LowPAN reach mainstream acceptance. The convergence of NEDs and the Internet provides new design opportunities, as digital communication networks will soon not only contain virtual data (images, text, etc.), but also data and services from physical objects. IP-based sensor networks have become widely adopted as a way to provide interoperability both with existing networking infrastructure and with existing equipment [109, 110, 116, 154, 201]. Unfortunately, even though IP provides a common networking layer which allows interoperability and evolvability of the system, application-layer interoperability is by no means guaranteed although it is the crucial element for building scalable distributed applications on top of heterogeneous devices.

### 2.1.3 Use Cases and Applications

In this section, we present a few typical use cases for NED applications that are considered in this thesis. Subsequently, we discuss the particular requirements for each scenario in order to derive a set of common features that need to be supported by the application protocol.

#### Real-time Enterprise

In today's business landscape, manufacturing companies are facing a huge market pressure for reduced production time and costs. Current industrial automation systems are often based on proprietary communication protocols; therefore integration with enterprise information systems (EIS) is tedious and costly. To remain competitive, companies need to improve end-to-end visibility in the supply, production, and delivery chains in a timely manner, which is essential to measure and optimize industrial processes. Although classic Web services help overcome the rigid and monolithic organization of traditional enterprise applications and facilitate data exchanges across departments and strategic units [71], there is still a huge gap between the real-time status of the physical assets of a company (equipment, inventory, machines, products, etc.) and higher level monitoring systems such as enterprise resource planning (ERP) applications [102]. NEDs could increase the efficiency of the business processes and the integration of device-level services with enterprise systems is essential for real-time visibility with fine granularity in the whole supply-chain. This in turn would allow critical information to be delivered

to the interested parties in a timely manner, so that appropriate measures can be taken with minimal impact on the production process. In particular, a judicious processing and filtering of large amounts of real-time information from the shop floor is required for efficient business activity monitoring to help with the optimization and maintenance of the equipment.

#### Digitally Augmented Environments

Home automation and building management systems increasingly rely on NEDs to improve the efficiency and comfort of houses or buildings. Even though home automation and security systems have been in use for decades, real-time monitoring via distributed sensors is becoming more popular thanks to the drastic price reduction of these devices. Besides, distributed monitoring applications can significantly improve security and safety thanks to a increasingly accurate and rapid detection of incidents. For example, digitally augmented environments are useful in various scenarios such as habitat monitoring [218], wildfire detection [238], security and surveillance [248], healthcare [26, 59], or energy efficiency [240]. Numerous projects in the pervasive community have looked directly at mobile interaction and digitally augmented environments, for example museum guides [96], location-aware applications and games (for example "Can you see me now?" [81] or "Urban Tapestries" [169] projects), or smart spaces [180, 200, 233].

#### Distributed Participatory Mobile Applications

Beyond small-scale applications in the Internet of Things such as smart spaces, larger scenarios are increasingly considered. As mobile phones and various sensors keep invading urban spaces, creating scalable and participatory sensing applications becomes an increasingly promising – but also challenging – goal. As shown by recent research [94, 117, 100, 212], new models that leverage these distributed "eyes" could significantly improve our lives and communities, as for example Seeclickfix [47], Citysense [9], Cabspotting [8], or Sproxil [53]. More recently, projects that explore city-scale platform to collect, process, and share real-time have been explored [203, 227].

### 2.1.4 Requirements for an Open and Scalable NED Ecosystem

In a world where billions of devices can sense, process, and exchange information with each other, a common ground understood by all devices is crucial. WSN frameworks have only marginally addressed the design of large-scale deployments of many heterogenous devices. As deployments grow in size, so do the incentive to share them across various applications and users. For this purpose, an open and flexible middleware solution that emphasizes usability and integrability is necessary.

An infrastructure for connecting sensor networks and other existing networks will also need to scale and evolve with minimal coupling between all its components (devices, users, applications, etc.). These observations shape the central research question addressed in this thesis: *"how to combine heterogeneous, mobile devices to form interactive, ad-hoc NED applications?"*.

As will be shown in this chapter, various solutions to integrate devices and applications have been proposed in literature. Simplicity and interoperability have been only marginally addressed, therefore most existing deployments have turned into isolated and incompatible islands of functionality. Because of the nature of the use cases presented in Section 2.1.3, the focus of this thesis is on lighter and simpler applications, where concerns such as integrability, scalability, and programmability prime over raw performance aspects (such as latency and/or throughput). Vinoski uses the notion of **serendipity** to describe such applications, a term defined as *"the occurrence and development of events by chance in a happy or beneficial way"* [235]. In essence, this is very similar to the notion of **opportunistic programming**, which emphasizes *"speed and ease of development over code robustness and maintainability"* according to Brandt et al. [92]. These principles lie at the heart of the approach defined in this thesis, which seeks to simplify the usage and integration of embedded sensors in order to enable fast prototyping of interactive and ad-hoc applications on top of many heterogeneous devices. In our vision, we assume that future sensor network deployments will be much larger in terms of nodes, which will require a greater degree of flexibility in order to be manageable. In particular, future deployments shall be **participatory** which means open and accessible for simultaneous users who want to easily retrieve data from devices and use it right away in their applications. Likewise, users could add new devices and services to the network with minimal effort thanks to the application-level interoperability made possible by using open standards.

Based on various sensor networks projects and deployments in the real-world [221, 170, 77, 76], one realizes that all NED applications can be classified in the following general categories:

- **Ad-hoc interaction (get/set).** Simple commands are sent to devices during operation time to read and write sensors, actuators, or application state (usually using a request-response model). This is the central pattern used in many home automation scenarios, where users can send commands to devices around them (for example *"turn on LEDs"* or *"read temperature sensor"*).

- **Continuous data collection (streaming).** Devices stream data and sensor readings at regular intervals and then all the data is analyzed on-the-fly, or stored in a database to be processed later. This pattern is commonly used for environmental monitoring scenarios (for example: *"send the light sensor reading every 5 seconds"* or *"get the temperature and humidity in this field every minute"*).

- **Event-based monitoring (eventing).** Events are generated sporadically by devices when predefined conditions are met (for example *"send an SMS notification as soon as*

*the temperature raises above 30 degrees Celsius"* or *"alert me as soon as a camera detects movement in this building"*). Collected data must be analyzed in real-time to allow timely reaction to environment state.

Such a coarse-grained abstraction certainly ignores many subtle aspects critical for each particular application at hand, but helps in generalizing application types so that they cover most scenarios for NED applications.

One can also observe that streaming and eventing are both based on a push model, the only difference being that notifications are sent periodically in the former case, and sporadically in the latter. This means that most NED applications can be entirely implemented using only two fundamental interaction models:

- **Request-Response Model.** The most common interaction pattern in pervasive applications is the *get/set* model, where the user can read and write directly sensors, actuators, and applications parameters by sending commands to devices. It usually follows a client-initiated cycle, where a server executes a command (request), and eventually returns a a value or acknowledgement (response). This is also the core model used by HTTP, where users can send a *read* request to see a Web form, and then *write* the content submitted in the form to a database. This is the central pattern for a user that wants to interact with a particular device in a given location or with a higher-level aggregation of device data that represent the environment state.
- **Push-based Model.** In some NED applications, devices must monitor the environment for a particular phenomenon to occur (for example detecting a fire or intrusion) and then react to it via notifications or actions. Devices must send data only if the phenomenon occurs and this as quickly as possible. In this case, polling is ineffective because latency is limited by the polling interval and the communication overhead must be minimized for resource-constrained embedded devices. Other NED applications have for purpose the periodic collection of data from the different nodes, where devices must send their data to a sink to be processed and stored at specified time intervals. This is exactly the same process as for events described above, the only difference is that data transmission is triggered by data a periodic timer instead of a rule or pre-condition. For this scenario, the pull-based request-response model of HTTP is not suited and a solution for pushing data over HTTP is necessary to provide timely information.

In addition to supporting both interaction types, an interoperable universal protocol suited for Internet-scale pervasive applications must also have the following properties:

- **Lightweight-ness** as the protocol will be running on embedded devices, therefore small memory footprint and parsing complexity shall be preferred.
- **Flexibility** as the protocol must be easily extended and adapted by users to cope with the various requirements and functions of a large spectrum of applications.

- **Scalability** as potentially billions of devices might be connected to each other, the protocol and its implementations must be able to scale and gracefully handle hundreds or thousands of concurrent users accessing these devices.

- **Loose coupling** as many different actors will have control over the devices and applications in the ecosystems, it should be possible to add, remove, or upgrade devices with minimal influence over the whole network in order to maximize the *evolvability* of the system.

- **Simplicity** as a large community should be able to access and use these devices, a low access barrier to develop applications must be ensured. WSN functionality should be abstracted easily behind high-level services that hide the peculiarities of various devices (devices are not always reachable, variable QOS, failure masking, etc.)

- **Standard** as all devices and applications shall speak the same language, a widely adopted protocol and minimal effort to access and use is needed.

As such a network shall be – at least partially – public, one needs to ensure open and royalty-free usage of the protocol and its implementations. As shown by the success of open-source projects such as Firefox or Apache (or the Web itself), openness, code sharing, and reuse are properties that play a central role in the wide adoption of any technology, therefore should be enforced. Besides, performance shall be comparable with other proprietary approaches, or at least sufficiently scalable and efficient to support thousands of concurrent mobile devices.

In many cases, applications might benefit from being connected to the Web, for example to share the data they collect or to reconfigure a running system via Web APIs. Previous projects attempted to integrate only some data produced into Web applications in a hard-wired manner, without enabling full access to devices' data and functions. By naturally blending the device API into the existing Web, data can be easily exported into Web applications using a standard format. Additionally, the entire functionality of the device becomes an extension of the Web, therefore all the operations, sensors, actuators, and properties of devices can be individually accessed and used in an uniform manner. Simplifying the integration by using a unified API instead of glue code and convertors for various protocols greatly improves the robustness and scalability of the resulting system.

In addition to these systemic properties, various higher level functions also need to be supported to facilitate the prototyping of scalable sensing applications: devices and services description for search and discovery, service composition and application prototyping techniques suited for mobile and embedded devices, and efficient eventing and stream processing techniques that are simple to use and to reprogram.

### Fast Prototyping Opportunistic Applications

Opportunistic programming stems from the hacking and making communities which emphasize fast prototyping of simple, lightweight strategic applications. In other words, mashups and Web applications had a great success because they allowed simple integration of various data sources with a low barrier to access and design such applications. We believe a similar model for NEDs can encourage fast prototyping as a design method, which would be beneficial for the pervasive computing community at large:

> *"Opportunistic practices in interactive system design include copying and pasting source code from public online forums into your own scripts, taking apart consumer electronics and appropriating their components for design prototypes, and "Frankensteining" hardware and software artifacts by joining them with duct tape and glue code."* in Hartmann et al. [148].

### Ad-hoc Interaction with Hybrid Environments

As more devices are being embedded into the environment, many use cases require the ability to interact with pervasive services with minimal prior knowledge about them without requiring to install drivers or specific applications. On the Web, interacting with any Web page is based on an uniform mechanism: any browser that talks HTTP can seamlessly interact with any server without any prior knowledge about the services or functions offered by the Web site. A similar mechanism that allows any mobile client to search and use services available in a physical location with minimal effort would be necessary.

### Real-time Sensor Data Management

Among the use cases addressed previously, many of them would require a highly scalable infrastructure to collect, process, share and store colossal amounts of real-time data from a wide variety of sensors. This will require the combination of many research areas that have only marginally been integrated together.

## 2.2 Distributed Systems Architectures

Building distributed applications has been an early challenge in computer science, and many approaches have been proposed for different application types [183]. Most of these approaches have primarily addressed application-layer interoperability between software components, but did not address the particular requirements of NED applications presented in the previous

section. Therefore, even if a classic middleware solutions could be used for some NED applications (e.g., in the back-end system to transport data between a database and a processing application), they cannot be used natively on NEDs because they do not meet most of our requirements.

In this section, we provide an overview of existing solutions for building NED applications from various perspectives. First, we introduce the notion of *architectural style* as a set of (generic) patterns and constraints for building distributed applications, describe the major styles existing, and discuss their properties in general. Then, we evaluate how they match the requirements of NED applications and discuss the advantages and drawbacks of each style when it comes to integrations with other devices and applications. Finally, we describe existing solutions that attempt to use the Web as an application protocol for inter-device communication.

### 2.2.1 On Architectural Styles

An *architecture* is a reusable system structure composed of components and interconnections and interactions between them, defined by standards and policies, and provides a template for subsystem structure and communication between subsystems which fosters reuse. An *architectural style* is a set of architectural constraints, that is a *"family of systems in terms of a pattern of structural organization, a vocabulary of components and connectors, with constraints on how they can be combined"* [210]. A *software architecture* determines how system elements are identified and allocated, how the elements interact to form a system, the amount of granularity of communication needed for interaction, and the interface protocols used for communication. In other words, it is the equivalent of design patterns for structures and interconnections within, and between, software systems.

In this section, we consider only three major architectural styles (object-oriented, service-oriented, and resource-oriented) and contrast two types of interactions (request-response and message passing). In the request-response model, a client invokes an operation on a remote object (usually synchronous), and operations have well-defined input parameters and return values. The emphasis is mainly on operation input/output and the correlation between them. In the message passing model, the explicit notion of client-server becomes less rigid, and the emphasis is on how to build a message (payload marshaling), how to send it (transport), and where to send it (endpoint). This offers a much looser coupling between consumers, which improves the scalability and evolvability of applications.

### 2.2.2 Object-oriented Architectures (OOA)

In OOA, applications are built upon interaction with distributed object instances. Communications are stateful and interactions are primarily remote procedure calls (RPC). The emphasis

is on creating a standard definition of the interfaces between objects, and on marshaling various data structures that are transferred between objects. RPC architectures are tightly coupled and consistency throughout the system must be ensured after any change in the interface, which makes these systems hard to scale and to maintain. This style was used predominantly in early distributed systems, and many standards for RPC-based communication have been proposed among which DCOM [11], RMI [17], or JINI [18]. Unfortunately, RPC architectures often used closed or proprietary protocols that are usually not Internet-friendly, thus are difficult to integrate in large-scale cross-boundary applications (for example cross-company supply chain management applications). Although this style still prevails today for most closed scenarios and critical applications (banking, healthcare, industry, etc.), they are being progressively replaced by more flexible architectures such as SOA/ROA which can improve system integration significantly and are also easier to work with.

### 2.2.3 Services-oriented Architectures (SOA)

SOA have been a significant improvement over RPC architectures by making distributed applications more flexible, robust, and interoperable. Similarly to RPC, interaction with services happen by sending messages (usually encoded using SOAP [50]) to service endpoints. Communications are usually stateless, which helps in scaling the system. Services usually offer a machine-readable description of their interface (usually using WSDL [236]) that specifies completely the messages and payloads used to interact with the service. This allows a looser coupling between clients and servers thanks to late binding and machine-readable interface contracts offered. Thanks to their versatility, SOAP-based Web services have been widely adopted by various industries in the last decade, come with an extensive array of additional standards that extend the basic SOA model with useful features for various types of applications. For example, WS-Discovery is a standard that defines how to find new services, WS-Eventing to support eventing, WS-Security for secure interactions. All these extensions are standardized by OASIS, and we refer to SOAP-based services and their extensions as `WS-*` throughout this thesis.

Although SOA usually rely on standard Internet and Web protocols (HTTP/XML over TCP/IP), they remain overly complex for most non-experts. Besides, `WS-*` extensions further augments the coupling between client/server as each party involved in interaction needs to support all the extensions used, which makes the development of such applications a hassle. In particular, SOAP-based systems were not designed with serendipitous interactions in mind, where various components come and go and interact in an ad-hoc manner with each other. As such, SOA are best suited for large-scale applications that cross organizational boundaries, but where only a limited amount of clients and services interact with each other in a static way, as all possible interactions should be known in advance. Obviously, every component must have a fully con-

sistent interface description document for each component they interact with, which is hard to ensure – and especially to maintain – in an open, participatory network.

### 2.2.4 Resource-oriented Architecture (ROA)

Unlike SOA where the fundamental units are services with well-defined individual interfaces (RPC method signatures) all available at a unique end-point, in a resource-oriented architecture (ROA), every basic component of an application is individually addressable (has its own endpoint) and they all use the same uniform interface (API). As more and more Web sites need to open up their data silos and allow direct access to their services and data via simple Web APIs, ROA has emerged as a much lighter and simpler alternative to SOAP-based Web services. Interactions happen directly with data instead of ready-made services wrapped around the data. This offers more flexibility for developers to access data in a neutral format and with a finer granularity. Interfaces are usually fixed, and each resource supports only a set of fixed operations that rarely change (therefore no need to regenerate all the clients/stubs after each change, as would be the case with SOA). Thanks to these properties, resource-orientation allows more scalable and robust architectures, and for this reason Web sites increasingly adopt ROA to offer their data and services (in particular using RESTful HTTP interfaces as discussed later in Section 3.2).

### 2.2.5 Discussion

Various architectural styles have been developed, used successfully, and cohabited for decades. In the early days, where computer networks were comprised of a few machines installed and maintained by the same authority, RPC-based systems were sufficient as the tight binding they impose only marginally affected the performance and scalability of the system. Besides, the high-predictability and low-evolvability was critical given the context these early systems operated in (for example terminals in a banking system). OOA are excellent for closed applications, where all the components interacting within the application are controlled by the same entity, and where new devices appear rarely, therefore ad-hoc interaction is unnecessary.

As Internet became increasingly used by various companies had to exchange data with each other, more loosely-coupled and open middleware solutions (as for example CORBA or JINI) have been preferred as they improved interoperability between applications by making the API programming language-agnostic. In the last decade, as the Web has become the most accessible medium to exchange information (both in B2B and B2C scenarios), SOAP-based Web services in combination with WSDL have become widely popular as they significantly improved interoperability between systems. Additionally, SOA allows to do more complex operations and its SOAP-based implementation comes with an exhaustive (and complex) stack of protocols and

extensions that add many features from secure communication to discovery. Most applications and programming languages support XML and HTTP, and one could easily build stubs to interact with specific services that do not need to be upgraded as long as the WSDL file does not change. Unfortunately, even though tools that help to automate the development of such services became widely spread and used, SOAP-based Web services remain quite difficult to understand and use, as in practice different implementations of the same `WS-*` standards are rarely entirely compatible with each other.

Another problem persists with SOAP-based Web services: they are a legacy of RPC style protocols, therefore are not flexible enough for highly dynamic environments where new, unknown devices continuously appear and disappear. Besides, `WS-*` consider HTTP only as a *transport protocol* to perform remote procedure calls instead of being directly integrated into the Web [244], in which case there would be no need for any additional API or descriptions of resource/function. This situation gave birth to numerous debates about ROA versus SOA[5].

| Attribute | OOA | SOA | ROA |
|---|---|---|---|
| Granularity | objects | services | data (resources) |
| Main focus | marshaling | creation of payload | addressing (URI) |
| Addressing | object instance | unique endpoint | individual resources |
| Replies cacheable | No | No | Yes |
| API | language-specific | application-specific | generic |
| Lightweightness | +++ | + | ++ |
| Flexibility | + | ++ | +++ |
| Scalability | +++ | ++ | +++ |
| Loose-coupling | + | ++ | +++ |
| Simplicity | +(+) | + | +++ |
| Standard | + | ++ | +++ |
| Tools available | +++ | +++ | + |
| Security | +++ | ++ | ++ |
| Verbosity | + | +++ | ++ |
| Ambiguity | + | ++ | ++ |

**Table 2.1:** Overall feature comparison of different architectural styles (+ means low, ++ means medium, +++ means high).

In Table 2.1, we summarize the differences and properties of the various styles we presented. Based on the requirements we defined in Section 2.1.4, we suggest the ROA style is the most suitable architectural style to implement a scalable participatory infrastructure for NED applications. In particular, as long as applications do not require custom protocols to meet particular "hard" requirements (such as reliability, security or throughput), the loose coupling of ROA architectures offers many advantages and features useful for NED applications. Also,

[5] An insightful discussion about the roots of this debate can be found here:
http://www.prescod.net/rest/rest_vs_soap_overview/

because ROA architectures tend to be simpler, they are more appropriate to be used directly on resource-constrained devices than SOA.

## 2.3 Middleware Solutions

The early vision of ubiquitous computing defined by Weiser [239] already mentioned heterogenous devices that could interact with each other to offer more complex services. Nevertheless, the integration of various devices into a common application has been a major challenge in networked embedded systems. As mentioned in [132], the high cost of development and deployment motivates platforms that support a broad class of applications that can host multiple independent users via resource sharing. According to [237], the use of a middleware can help reduce the gap between the high level requirements of pervasive computing applications and the underlying operation of WSNs. A *middleware* is defined by Bernstein [84] as *"a distributed system service that includes standard programming interfaces and protocols"*. These services are called middleware because they act as a layer above the OS and networking software and below industry-specific applications. A middleware is therefore a common ground to improve interoperability when integrating various components in distributed applications. The EM-* project at UCLA, is such an attempt to provide an environment to develop and deploy complex applications on top of heterogeneous sensor networks [131]. The Speakeasy [118] project at PARC has developed a new approach to interoperability, called recombinant computing. It allows devices and services that were not explicitly designed to interact with each other to interoperate fluidly. JXTA is an open network computing platform designed for peer-to-peer computing that can be implemented on all kinds of devices [222], but is not based on the Web architecture. Agimone [135] is a higher-level middleware that supports integration of WSN and IP networks. It focuses on the integration and coordination of WSN applications beyond WSN boundaries. The MiLAN middleware [150] focuses on sensor network management to enable proactive WSN applications which support QoS requirements and energy constraints. Sgroi et al. [209] suggest basic abstractions, a standard set of services, and an API to free application developers from the details of the underlying sensor networks. However, the focus is on systematic definition and classification of abstractions and services, while issues such as scalability and integrability are only marginally considered. Several approaches that attempt to define an unified API for sensor networks have been proposed [160, 99, 217]. Other bridging methods for sensor networks with the Internet are proposed in [166, 172], however the bridging is only at the network level (IP connectivity) and not at the application level.

## 2.4 Internet as Transport Protocol

Because of the versatility and omnipresence of the TCP/IP protocol, various projects have attempted to leverage the Internet to build more scalable, distributed applications. For example, Hourglass [213] provides an Internet-based infrastructure for connecting sensor networks to applications and offers topic-based discovery and data-processing services. HiFi [124] relies on a hierarchical tree model where sensors (leaves) are connected through relatively powerful intermediary nodes that can perform various operations such as filtering, data cleaning, aggregation and join, which is similar to the Tenet architecture for tiered sensor networks proposed in [133].

To improve interoperability across systems, recent projects have focused on using Service Oriented Architecture (SOA), in particular Web Services standards (SOAP, WSDL, etc.) directly on devices [157, 102, 201]. *Device Profiles for Web Services (DPWS)* [192] is a subset of SOAP-based Web services created to integrate devices into the networking world and make their functionality available in an interoperable way. It offers various useful features such as dynamic discovery, interaction, or search, and is a more open refinement of its predecessors Jini [18] and UPnP [58]. Initially, home automation was the main scope for these standards, but efforts within the SOCRADES project [102] have shown its applicability to the industrial automation world as well.

## 2.5 Internet as Application Protocol

The term Sensor Web refers to a global network of Web-connected sensors, and several projects have been proposed to build such a worldwide Sensor Web. With advances in computing technology, tiny Web servers could be embedded in most devices [88]. The idea of each thing having its own Web page is appealing because Web pages could be indexed by search engines, then searched and accessed directly from a Web browser. Early approaches to linking objects with the Web were based on physical tokens (such as barcodes or RFID tags) [174]. In the Cooltown project [167], each thing, place, and person has an associated Web page with information about them.

IrisNet [130] proposes a two-tier architecture consisting of sensing agents which collect and process sensor data, before storing it into a hierarchical and distributed XML database. The model used is similar to the DNS system used in Internet stack and can be queried using XPath. A middleware for smartphones, with event delivery, acquisition, query resolution, aggregation, access control, storage, registration and availability has been proposed in [230]. The proliferation of smartphones throughout the past few years has provided an increasingly ubiquitous platform for intelligent services at the edge of the network. Sensorbase [98] is a centralized sensor data logging system. WSN send data in plain format (e.g., as CSV file) to a central component that acts as data logger. The data is stored into a database from where it can be searched and retrieved via a Web-based graphical user interface or API. Sensor

updateThrough RSS feeds, changes in sensor data can be distributed asynchronously in a publish/subscribe paradigm. Additionally, the middleware can be accessed through SOAP as a traditional Web service. The SenseWeb [164] project proposes a middleware where a central coordinator keeps track of the various devices registered and handles external queries from various Web clients by caching sensor data to avoid polling the device frequently. The Global Sensor Network (GSN) [66] is a middleware for connecting sensors to the Internet based on the concept of virtual sensors that can be actual raw sensor data or higher-level component such as an aggregation from different sensors. The middleware collects sensor data streams and all data tuples are stored into a centralized database which allows users to access and query for data directly from the Web. Prehofer et al. [200] recently proposed a Web-based middleware, however, Internet is used only for transport as there is no mention about using HTTP as an application protocol.

## 2.6 Towards the Web of Things

Projects that specifically focus on re-using the core architectural principles of the Web as an application protocol are still lacking. Creation of devices that are Web-enabled *by design* would facilitate the integration of physical devices with other content on the Web. According to [244], physical "items" (sensors) should be integrated directly into the Web as resources, following the design principles of the Web (e.g., REST principles, see Section 3.2). Sensors should be uniquely addressable via URI (comparable to hypertext documents), and users that want to interact with any sensor can use any HTTP client library, which lowers the access barrier significantly. As pointed out in [244], there would then be no need for any additional API or descriptions of resource/function.

Early mentions of the Web of Things came from Dave Raggett [202] and Erik Wilde [244], which inspired our early work on the Web of Things, where Web technologies were used to control robots [225] and other embedded devices [224]. Much of the Web of Things research has been done in collaboration with Dominique Guinard, which led to key publications on the basis of the Web of Things [139, 144, 141]. While this thesis emphasizes distributed infrastructures, gateway-based enabling, and stream processing frameworks for the Web of Things, Guinard's research emphasizes devices sharing with social networks [137], resources description and search [143, 140], and various prototypes from Web-enabled energy monitoring [145, 240] to RFID-based supply chain applications [138]. Much of the work presented here is complementary in content – yet compatible in concept – with his own dissertation [136]. Details about differences and references to Guinard's work are given throughout this thesis.

Architectures that allowed devices services to be accessible directly over the Web have been already proposed and investigated in literature. A generic architecture for accessing devices using SOAP-based Web services was proposed in [103], where a gateway is used to enable

access to device services, however no implementation details or evaluation results are given. Bagnasco et al. [75] describes a two-layer architecture for accessing device functionality using REST. The lower layer provides basic services to access heterogeneous hardware resources using a RESTful Web service architecture. Devices run a Web server on top of TCP/IP and the services offered are used by a more powerful gateway to expose these services to external clients which use SOAP-based services. Although the gateway offer a WSDL file to describe the SOAP-based services offered by the device, there is no end-to-end REST support, which limits the Web integration of their solution. In the tiny Web services project [201] a Web service implementation is presented that is directly installed on devices. The implementation has a very small footprint (48kB of ROM, 10kB of RAM) and uses WSDL to describe services, but SOAP should avoided as binding because of its overhead.

pREST [108] describes how to use REST to interlink resources in NED applications. Everything in pREST is modeled as Web resource. Producers and consumers of sensor data can then be connected using the pREST protocol (e.g., a camera sending an image to the Web server to be published). TinyREST [176] also proposes a gateway to connect directly devices as resources into the Web, thus allows clients to send Web requests to the URI of various device service via the gateway. Unfortunately, they introduce an additional verb (`SUBSCRIBE`) in addition to `POST` and `GET` to support publish/subscribe interactions. This is contrary to the REST principles as new operators are introduce which increases the coupling between components therefore reduces compatibility with other RESTful applications. Besides, the paper mainly focuses on isolated experiments and does not address the scalability of the system. Web feeds have been used to access data provided by sensor nodes [107]. In particular, they describe an extension to RSS suited for high-rate data streams with a Web-oriented querying interface to retrieve sensor data. A direct consequence of the stream abstraction is that sensors are considered solely as data publishers, not as service providers. The approach found in [216] has a very similar approach as well, but focuses mainly on the discovery of devices. Unfortunately, a more systematic approach, implementation details, and a performance evaluation are all lacking.

Various community Web sites that allow people to share their sensors using more Web-friendly data formats (XML, JSON, CSV) have appeared in the last two years such as Pachube [36], Sensorpedia [49], Sensor.network [146], Evrythng [13], ThingWorx [56], or Sen.se [34]. These solutions offer various tools and services, unfortunately they are all based a centralized application, which is contrary to the distributed nature of the Web.

## 2.7 Discussion

In this chapter, we have described the requirements for future distributed sensing applications and open ecosystems that foster the integration of data and services from embedded devices. We have then surveyed the related work in integrating heterogenous devices and discussed their

shortcomings with respect to our requirements. In this section, we summarize our key findings and discuss the limitations that will be addressed in this thesis.

Early distributed programming abstractions based on Remote Procedure Calls (RPCs) have simplified the development of complex distributed systems [204]. As seen in Section 2.2.5, RPC-based systems are not flexible enough to build evolvable and scalable networks of embedded devices, as they heavily rely on tight interface contracts that increase coupling and complexity (binding devices strongly to a specific application scenario). Connecting incompatible middleware requires expensive protocol translation and data transformation, which makes it difficult to integrate sensors across various applications.

In the last decade, various projects proposed to leverage the Internet infrastructure already in place for building distributed applications. Although these projects proposed original and efficient solutions for the design of such applications, they only used the Internet protocols (TCP/IP and/or UDP) as a data *transport* mechanism, by enforcing various (non-standard) application-level protocols designed for each deployment at hand. Using Internet protocols does not guarantee application-level interoperability, as various deployments could only exchange data with each other, but not necessarily *understand* the content transmitted.

More recently, many projects have proposed to use the Web for providing access to sensor data, in particular Web services and/or Web applications. However, most of these projects suffer from two main drawbacks. First, these projects proposed only to integrate the *data* generated by devices into the Web, not the devices and their *services* themselves. The priority was often to link physical objects with their virtual representation on the Web, rather than actually *integrating* devices as actual Web resources along with an API that allows programmatic access. Such an API would allow devices data and services to be easily integrated, searched, browsed, and used just like any other content on the Web. Second, HTTP was often used only as transport protocol on top of which custom application protocols were required, whereas HTTP is in fact an application protocol. Besides, most of these approaches required a centralized repository where all the devices had to be registered before they can publish data. This might not be sufficient for Internet-scale distributed applications that require large amounts of data to be processed in real-time. Besides, using custom protocols on top of HTTP introduces a tighter coupling that prevents ad-hoc interaction with devices, and most solutions do not support direct interaction with devices without server mediation. Such a central point of failure is against the distributed nature of the Web, which reduces the scalability and evolvability of NED applications.

The various projects presented showed that Web services can provide a more elegant and flexible approach to develop NED applications. In particular, Web technologies support more scalable applications that integrate many devices and clients simultaneously. However, mechanisms for accessing sensor data, such as push notifications and streaming, are not supported. Besides,

an application-level infrastructure that supports semantic discovery and search based on real-time sensor values have been only marginally explored. Finally, frameworks that facilitate the integration of heterogenous devices that do not support Web protocols and end-to-end applications on top of them have are not available. In this thesis we address these limitations, and propose the fundamental building blocks for a scale and entirely Web-based infrastructure for building interactive NED applications.

# CHAPTER 3

# Web-oriented Architectures

*"It is unworthy of excellent men to lose hours like slaves in the labor of calculation which could be relegated to anyone else if machines were used."*

Gottfried Wilhelm von Leibnitz (1646-1716)

In the previous chapter, we have highlighted how NED applications benefit from the integration of various data sources and applications by cross-organizational actors. When devices and their services can be easily shared and reused at a global scale, tremendous network effects will emerge, which will bring this technology to its full potentials. The Web has been successful at building a large participatory network that enabled over two billions users to share and retrieve information. In this chapter we explore the reasons why it has worked and present how to leverage these benefits for NED applications as well.

## 3.1 Web-based Architectures

Because the number of embedded devices with built-in Internet connectivity is growing exponentially, it is very likely that in the near future there will be more NEDs on the Web than humans. The combination of increased processing power on NEDs and the flourishing of robust and easy to use frameworks for developing Web applications, will make it simple to fast prototype Web applications that break out from browsers and servers into the real world. However, to realize the vision of a global network of NEDs, devices will need to be fully integrated with the existing Web infrastructure. This means NEDs shall *speak the same language* and behave just like any other Web resources. Using Web standards to interact with NEDs will make it much easier to merge the real world with existing Web content, and physical things could then be bookmarked, browsed, queried, or shared just like any content on the Web.

This chapter describes the principles of the modern Web architecture and how they can be ported on embedded devices to support efficient Internet-scale NED applications. A common assumption is that Web technologies are inherently inappropriate for embedded devices, which raises the central question addressed in this thesis: *"is the HTTP protocol as-is, ready to support NED applications, or are new and custom protocols on top of HTTP unavoidable?"*. Even though the current specification of the HTTP protocol has not been updated since 1999 [121], Web applications have tremendously evolved over the last decade. This observation motivates the core hypothesis of this thesis: Web technologies have not been fully exploited – especially in the context of embedded devices – and further exploration could bring the Web to its full potentials, even without upgrading the current Web infrastructure and its standards.

### 3.1.1 Modern Web Infrastructure

At the core of the *World Wide Web (WWW)* lies the HTTP protocol, which was designed as a substrate for building a distributed hypermedia system for linked multimedia documents [82]. As demonstrated by the tremendous success of the Web, loosely coupled approaches offer the greatest scalability, robustness, and evolvability. In recent Web 2.0 applications [193], the focus has been on the user and user-generated content, and many applications (and especially the data and services within) were made accessible via open Web APIs. This has lowered the barrier for programmers to build *Web mashups*, which are hybrid applications that combine content from various sources, as for example the chicago crime maps.

In the last two decades, various tools and techniques have transformed the Web in the most successful and accessible information sharing platform, and we present five fundamental *pillars* (or *trends*) that form the core foundation of the modern Web we know and use today. As shown in Figure 3.1, the combination of these ingredients opens up a new, unexplored design space where new solutions to old problems can be conceived. In this thesis, we define the **Web of**

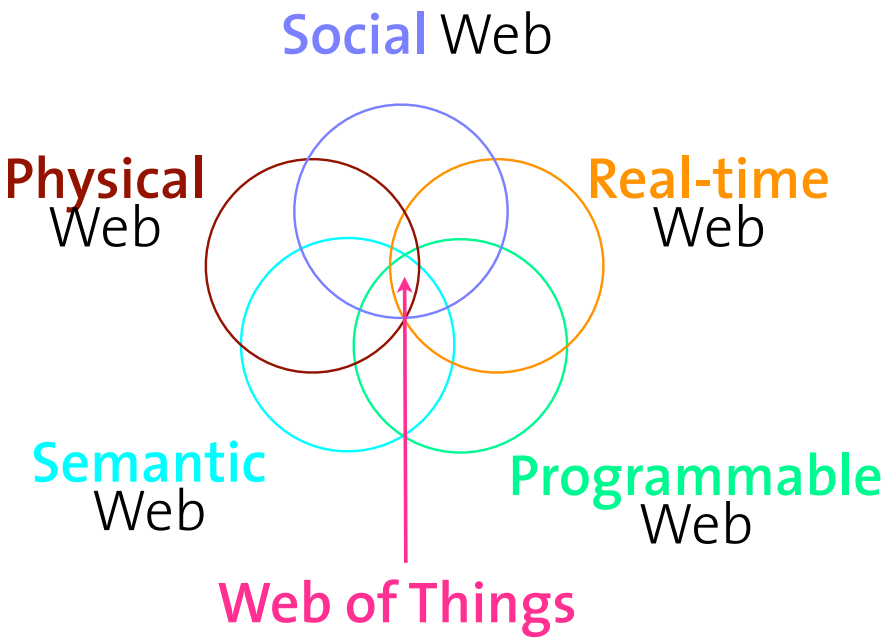



**Figure 3.1:** The Web of Things (WoT) builds upon five pillars of the modern Web architecture. We define the WoT as an unexplored design space to build more efficient and flexible distributed sensing applications built upon the combination of the pillars of the modern Web.

**Things (WoT)** as being the intersection of these five trends, and propose the Web of Things as an evolution of the Web that reaches into the physical world.

**Trend 1: The Social Web.** The emergence of open and participatory services on the Web such as forums, blogs, or wikis have played a critical role in the shift from a Web of pages towards a Web of people. Countless online services are available today to build and support virtual communities, from social networks such as Facebook or MySpace, to more generic social platforms (Ning, Facebook/Google connect), to authentication and identification tools (OAuth, OpenID), to social data portability[1].

**Trend 2: The Semantic Web.** Another component of the modern Web is the rise of machine-readable content being embedded along or within Web sites. Various semantic annotations are increasingly used today such as metadata header information parsed and indexed by search engines, machine tags used on Flickr[2], microformats [165], or even elaborate semantic languages such as OWL or RDF/RDFa [69]. More generally referred to as Linked Data [85], many tools and techniques can be used to publish, connect, and find structure information on the Web.

[1] Online: http://dataportability.org/
[2] See: http://www.flickr.com/groups/api/discuss/72157594497877875

**Trend 3: The Real-Time Web.** From finances, to news, to social networks, many domains rely on real-time information delivered in a timely manner, and online services increasingly require large amounts of information to be robustly delivered as fast as possible. Various tools and techniques have been appearing to transmit timely information over the Web. As described in Section 6.1.2, the traditional poll-based model of HTTP is being rapidly supplemented by techniques such as Comet or Web sockets to push data to clients over the Web asynchronously.

**Trend 4: The Programmable Web.** A key feature of the Web 2.0 is the ability to access raw data from Web applications via a Web API. Many companies have realized the benefits of allowing their data and services to be accessed in a programmatic way, not only deliver the finished product. Today, thousands of Web sites offer their data through open APIs[3]. This has enabled developers to easily build new *mashups* – services built on top of existing API by combining content and functionality from various sources (see Section 3.4).

**Trend 5: The Physical Web.** The Web has been increasingly stepping out of servers and browsers as various *things* [4]have been connected to the Web or Twitter, such as such as bridges, plants, houses, or even bakers. This extends the reach of the information on the Web to real-world status of physical objects. Thanks to the proliferation of smart phones with GPS sensors and Internet connectivity, location-aware applications have been flourishing and services from Google Places[5] to Foursquare[6] make it possible to create a physical Web, where the content delivered depends on the physical location where it is accessed from.

### 3.1.2 Motivation for a Web of Things

As the size of typical NED applications will continue to increase, so will the incentives for these applications to be more open and shareable. In this thesis, we propose the Web of Things as a viable solution to build more scalable, open, and flexible NED applications, and investigate the hypothesis that the Web is an excellent medium to build such applications. There are various reasons why integrating NEDs to the Web is desirable.

First, native integration of devices (as opposed to only integrate their data or using a Web page to control them) allows to treat devices and their services just like any other Web resource. This makes it easy to tap directly the expertise accumulated over the last two decades in building massively scalable Web sites, but also benefit from other tools and techniques widely used on the Internet such as caching, linking, search, authentication, or scripting among others. In

[3]As of 1 June 2011, the ProgrammableWeb directory lists of 3301 APIs and 5842 mashups [40].
[4]http://www.extremetech.com/internet/84236-the-five-best-twitter-feeds-by-nonhumans
[5]Online: http://www.google.com/places/
[6]Online: https://foursquare.com/

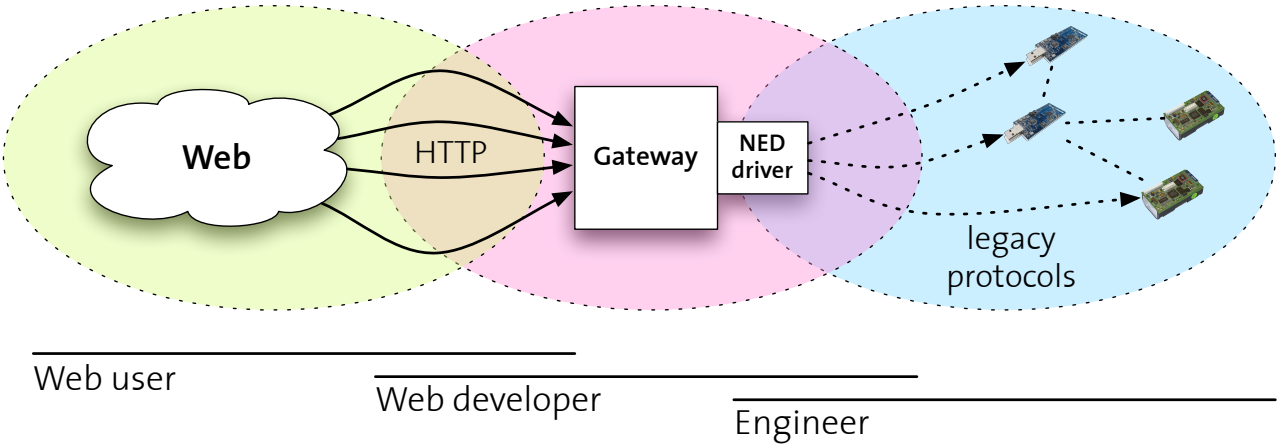



**Figure 3.2:** Enabling Web developers and users to access, use, and program NEDs democratizes innovation and facilitates prototyping of distributed sensing applications, which in turn unlocks new design opportunities.

other words, native Web integration allows devices and NED applications to directly leverage the five pillars of the modern Web with minimal effort.

Second, native integration diminishes the costs to network heterogeneous devices as the Web infrastructure is already in place: TCP/IP is omnipresent, Wi-Fi routers are ubiquitous in occidental households, and Web standards are efficient, well known, and used by millions of developers and billions of users[7]. HTTP is a highly versatile and omnipresent protocol thanks to its simplicity, powerful and scalable Web servers are freely available as open-source projects, HTTP clients and libraries exist for virtually any programming language and platform. Furthermore, using Web standards to interact with devices makes it possible for the applications built upon them to benefit from the system properties introduced by the modern Web architecture such as scalability, evolvability or simplicity.

Third, Web applications are often simpler and faster to develop than classic desktop software applications. Current software for real-world integration and business applications are tailored for specific use cases, thus are often too rigid and closed to be customized by end-users easily. In a large ecosystem of networked devices, one could use a higher-level, declarative approach to rewire and recombine data, in particular use tools commonly used to develop Web mashups.

Tightly coupled back-end systems using proprietary or optimized protocols will remain the most desirable choice for high-performance systems with specific requirements (as for example in the industrial automation or banking domain). Nothing prevents the functionality of these systems to be exposed on the Web using a high-level API to hide the complexity and underpinnings of the closed back-end. However, much simpler loosely coupled approaches are to be preferred for tasks where latency and throughput are less of a concern (which represent most use cases in building automation or environmental monitoring), because of their inherent flexibility and intuitive use.

[7] As of 1 June 2011, over 2.1 Billion users according to: http://www.internetworldstats.com/

Allowing people to easily reuse and combine data sensed by different devices anywhere around the globe will lower the entry barrier for developing monitoring applications. The loss in raw performance induced by choosing HTTP over optimized protocols, is largely compensated by the seamless Web integration, a simpler programming model, and by the availability of tools, techniques, and expertise in building robust, secure, and massively scalable Web applications.

In the remainder of this thesis, we describe the fundamental building blocks of the Web of Things. We explain how interoperability at the application layer can be obtained by leveraging the mechanisms and protocols that are successfully used by the World Wide Web. We describe how Web patterns cab be used to support both automatic data acquisition from, as well as manual interaction with NEDs. Automatic data acquisition is important for integration into enterprise IT systems, such as databases, while manual interaction enables users to interact with and to configure on-the-fly devices – or groups of them – simply using a Web browser. This allows to turn the Web into an ecosystem of heterogenous devices and services that can be recombined at runtime to build applications.

## 3.2 Representational State-Transfer (REST)

REST (Representational State Transfer) is the architectural principle that lies at the heart of the Web and has been described in Roy Fielding's PhD thesis [120]. Rather than being a technology or standard, REST is an architectural style (see Section 2.2) which attempts to increase interoperability for a looser coupling between the various components of distributed applications. REST is the underlying principle of HTTP and consists of a set of constraints that defines how distributed applications should be built.

In other words, REST defines the core architecture of the modern Web and emphasizes the use of the Web (and of its central protocol HTTP) as an *application* protocol, and not only as a *transport* protocol in the way that `WS-*` Web services do[8]. This way, Web applications that comply to the REST architectural style are said to be *RESTful* and benefit directly from all the features of HTTP such as authentication, authorization, encryption, compression, or caching. REST follows the principles of resource-oriented architectures (ROA) in order to achieve simpler and more lightweight integration by focusing on resources instead of functions. This allows to interact with services without requiring additional service description languages (e.g., WSDL, IDL, etc.) for generating complex source code and stubs.

The Web was designed to create a large-scale distributed hypermedia system, and according to Roy Fielding [120]:

[8]Many integration technologies on the Web (among which `WS-*` Web Services) reduce the role of HTTP to only a transport protocol by using only a minimum set of it features (only GET/POST verbs of HTTP, only status code 200 or 404, etc.) and redefine a complete custom application protocol on top of HTTP (SOAP for `WS-*` ). This is redundant as HTTP is already an application protocol that doesn't require an additional layer of complexity on top.

> *"REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems."*

REST focuses on the various components in a distributed application and their role, and constrains the interactions between them, but ignores the implementation of the components or protocol syntax. Components interact by exchanging representations of resources in a format commonly agreed upon. For any distributed hypermedia system to benefit from the emergent properties of REST (performance, scalability, simplicity, modifiability, visibility, portability and reliability), it must comply to the REST architectural style by applying the following five[9] constraints to the architecture:

- **Client-server.** Clients and servers communicate via a uniform interface, which reduces the coupling between both roles. Clients do not need to know the internals of the server, that is how data is stored or processed. Likewise, servers are not concerned by user interfaces and client state. This separation of concerns between data and control logic improves scalability and portability of the client code, and such a loose coupling enables the development of components independently, which is desirable in a large scale Web of Things.

- **Uniform interface.** Loosely coupled architectures that allow the evolvability of a system usually result from using a uniform interface between components. Unambiguous, simple, and standard interfaces that can be easily extended for various content and contexts have largely contributed to the success of the Web as a participatory system. This is a central feature for the Web of Things to which new, unknown devices can be added and removed at any time with little effort.

- **Stateless.** The client context and state should not be kept on the server, but on the client. As each request to the server must contain the client state, visibility (monitoring and debugging of server), robustness (recovering from network or application failures), and scalability are improved. Servers and applications can of course be stateful, this constraint simply requires the state to be addressable.

- **Cacheable.** Caching has been a key factor in the performance (loading time), thus usability of the Web today. Both clients and intermediaries can store data locally, which boosts their loading time, and server can define policies when data expires and when updates must be reloaded from the server. This results in higher performance, as reduced client-server interactions improves server scalability and reduces latency, while limiting the maximal data age that can be acceptable.

[9] According to [120], in fact there is an optional sixth constraint about code-on-demand, where clients can download and execute code locally such as Java Applets or ActiveX controls, but this aspect will not be discussed here.

- **Layered system.** An uniform interface makes it easy to design a layered system, which in turn allows to use intermediary servers to improve scalability and response time via load balancing and shared caches or distributed content delivery networks (e.g., Akamai[10]), as clients do not need to know if they interact with the target server or some proxy along the way. Besides, a layered system enables encapsulation of legacy protocols and systems (for example gateways to proprietary protocols) or simply enforce various security policies [137].

The most important of these constraints is the uniform interface, and there are four guiding principles that any REST interface must follow to achieve these goals:

1. **Identification of Resources.** A *resource* is every concept or piece of data in an application that is important enough to be referenced or used. Every resource must have an unique identifier and should be addressable using a unique referencing mechanism in order to maximize visibility, reusability, and simplicity. On the Web this is done using the URI specification.

2. **Manipulation of resource through representations.** One interacts with services using various representations of resources suited for different uses or platforms. Commonly used representations include (X)HTML for browsing and viewing content on the Web, XML which is more suited for machine readability, and its lighter alternative JSON.

3. **Self-Descriptive Messages** When applying REST over HTTP, one must only use the methods provided by the protocol (`GET`, `POST`, `PUT`, and `DELETE`) and stick to their meaning as closely as possible.

4. **Hypermedia as the engine of the application state.** Servers should not keep track of each client's state, as stateless applications are easier to scale. Instead, every possible application state should be addressable via its own URI, and each resource contains links and information about what operations are possible in each state and how to navigate across states.

Thanks to their simplicity, the use of a uniform interface, and the wide availability of HTTP libraries and clients, RESTful services are truly loosely-coupled [198]. This concretely means that services based on RESTful APIs can be re-used and re-combined in a quite straightforward manner, without requiring prior knowledge about the specifics of any resource (which can be discovered and "understood" on-the-fly, see Section 3.2.4).

Because of the underlying simplicity of this architecture, using the REST style and HTTP as application protocol for NEDs provides many advantages desirable in pervasive computing applications. First, minimal coupling helps to design more scalable and robust applications. Second, one can use a Web-like mindset to design applications (markup languages, event-based

[10] Online: http://www.akamai.com

browser interactions, scripting languages, URIs, etc.). Third, HTTP traffic on port 80 is the only protocol that is (almost) always allowed by firewalls in companies. Fourth, using REST one inherits the benefits of a resource-oriented architecture directly on embedded devices which makes it easy to hide low-level protocol details behind simple high-level abstractions, thus fostering openness, programmability, and reusability of NEDs.

### 3.2.1 Addressable Resources

Resource-oriented architectures are well suited for data-centric applications, as every element of an application that has to be addressed explicitly (e.g., a sensor, its sampling frequency, a variable, etc.) can be modeled as a resource addressable via a (globally) unique identifier. On the Web, this is done using the well-known URI [83] standard scheme that allows granular and transparent access to data elements in a simple manner. Using the same standard naming scheme as other Web resources allows to seamlessly blend devices into the Web, as their functions or sensors can be linked to, shared, and bookmarked exactly like any other Web resource.

These are examples of URIs to address different parts of an application:

```
# User with the ID No. 12
http://webofthings.com/users/12

# Sample No. 77654 from october 2009
http://webofthings.com/samples/2009/10/77654

# Device called lamp
http://webofthings.com/devices/lamp
```

These examples identify unique resources (i.e., objects). However, one can also identify *collections* of resources, which are also resources themselves:

```
# a list of sensors on a device (all the sensors on device ID 24)
http://webofthings.com/devices/24/sensors

# a list of devices in an area (building 4)
http://webofthings.com/building4/devices/

# a list of sensor readings
http://webofthings.com/devices/4554/samples
```

We illustrate this principle with an actual embedded device in Section 3.3. All the components of a NED (Sun SPOT platform) are mapped into a resource tree, where each sensor, actuator, and system property of the device is assigned its own URI. This way, the physical device fully blends into the Web, as it becomes completely accessible via a RESTful Web API.

### 3.2.2 Uniform Interface

REST emphasizes a uniform interface between components to reduce coupling between operations and their implementation. This requires every resource to support a standard, common set of operations with clearly defined semantics and behavior. HTTP defines a fixed set of operations that every resource can support (also called *verbs*), the most commonly used among them are:

- `GET` is a read only operation. It is both an idempotent and safe operation. *Idempotent* means that no matter how many times you apply the operation, the result is always the same. The act of reading an HTML document shouldn't change the document. *Safe* means that invoking a `GET` does not change the state of the server at all (read-only).

  **Example:** Read a resource, for example the temperature sensor of a device (minimal HTTP example).

  ```
  GET /temperature HTTP/1.1
  Host: mydevice.ch

  200 OK HTTP/1.1
  Content-Type: text/plain
  Content-Length: 4
  37
  ```

- `POST` is both non-idempotent and unsafe operation of HTTP. `POST` usually models a factory service, where the URI of the newly created is not known in advance.

  **Example:** Create a new resource, for example a rule that triggers a call-back when a threshold is exceeded encoded using a fictive rule language (the URI of the created rule is returned in the answer via the `Location:` header):

  ```
  POST /rules HTTP/1.1
  Host: mydevice.ch
  Content-Type: text/plain
  IF light>150 THEN CALL http://example.com/altert-handler

  200 OK HTTP/1.1
  Location: mydevice.ch/rules/2210
  ```

- `PUT` is usually modeled as an idempotent, unsafe insert or update method. When using `PUT`, the client knows the identity of the resource it is creating or updating. It is idempotent because sending the same `PUT` message more than once has no effect on the underlying service.

  **Example:** Update a resource, e.g., change color of LED No. 4 with its new RGB value as parameter encoded using comma-separated values (CSV, see next section):

```
PUT /leds/4 HTTP/1.1
Host: mydevice.ch
Content-Type: text/plain
0,128,128

200 OK HTTP/1.1
```

- `DELETE` is an idempotent, unsafe method used to remove a resource.

  **Example:** Delete rule No. 24 running on a device.

```
DELETE /rules/24 HTTP/1.1
Host: mydevice.ch

200 OK HTTP/1.1
```

Although using only these four operations might seem insufficient to write distributed applications, the success of other CRUD (Create, Read, Update, Delete) systems proves otherwise. For example, SQL has only four main operations (SELECT, INSERT, UPDATE, and DELETE), or message-oriented middleware (MOM) are based on only two core operations: send and receive. In both cases, the actual complexity is found in the data model (SQL) or message body (MOM). Most of these systems actually define many (optional) helper operations.

HTTP also defines a list of standard status codes to be returned by the server upon reception of the request (see Chapter 10 in [121]). The most commonly used are:

`200 OK.` Returned upon successful completion of a request.

`201 CREATED.` Returned when a new resource has been successful created. The header `Location` contains the URI of the resource that has been created.

`401 UNAUTHORIZED.` The request requires user authentication or the authorization failed using the given credentials (see [125]).

`404 NOT FOUND.` The requested resource or document has not been found on the server.

`500 INTERNAL SERVER ERROR.` The server encountered an error that prevented it to fulfill the request.

`501 SERVICE UNAVAILABLE.` The request cannot be handled by the server at this time due to maintenance of temporary overload.

Limiting all possible interactions to a subset of generic operations simplifies the overall system architecture and improves the visibility of interactions. This makes it more difficult to implement the complex workflows commonly used in business applications as they must be mapped into a sequence of simple, atomic operations. Nevertheless, a standard interface for Web services with a minimal set of operations offers desirable advantages for building large-scale heterogeneous applications, among which:

- **Interoperability.** Virtually any programming language or environment comes with an HTTP library. As long as standard data exchange formats are used (MIME types seen in next section), interacting with RESTful services doesn't require service-specific libraries as is the case with `WS-*` services. This way, developers can focus on data and operations and worry less about interoperability.
- **Familiarity.** Given the URI of any service, one knows exactly how to interact with the URI and use the service (and find about the operations it accepts using the `OPTIONS` HTTP verb) without requiring an IDL or WSDL document or specific stubs – only an HTTP client library suffices. Following links to discover at run-time all the services available is a straight-forward procedure.
- **Scalability.** The limited set of methods supported by REST makes behavior predictable which translates into performance increase. Idempotent and safe methods such as `GET` make content cacheable by HTTP proxies and browsers, which drastically reduces network traffic and minimizes server load using front-end caches. The looser coupling between client and server enabled by the generic interface makes it also easier to interact ad-hoc with any HTTP-compliant service.

### 3.2.3 Representations and Encodings

A central challenge in computer-based communication is how to encode some information so that it can be universally decoded and understood. On the Internet, *Multipurpose Internet Mail Extensions* (MIME) types have been introduced as standards to describe various data formats transmitted over the Internet such as images, video, or audio. The MIME type for an image encoded as `png` is expressed with `image/png`, and mp3 audio with `audio/mp3`. A list of the all the official MIME media types is maintained by the *Internet Assigned Numbers Authority (IANA)*[11].

Web Resources are only a concept – an abstract idea of a thing, not the thing itself. The tangible instance of a resource is called a *representation*, that is a standard encoding of a resource using a MIME type. Web browser typically support a few basic representations (HTML, GIF, MPG) or can use plugins or external applications to render them (for example PDF, vCards, Flash). The same resource can also be available in various languages or encodings.

HTTP defines a simple mechanism called *content negotiation* for allowing clients to request preferred data format they want to receive from a specific service (see Chapter 12 in [121]). Using the `Accept` header, clients can specify the format of the representation they desire. In the same way, servers specify the format of the data they return using the `Content-Type` header. To retrieve information about a user with the ID 1234, one can issue the following request (minimal HTTP):

[11] Online: www.iana.org/assignments/media-types/

```
GET /users/1234 HTTP/1.1
Host: webofthings.com
Accept: texl/html

200 OK HTTP/1.1
Content-Type: text/html
<html>
...
```

By default, browsers request HTML files they can render and allow user to interact with. In some cases, HTTP clients might request a machine-readable document (when the client is an application and not a Web browser), for example encoded as XML. This is done as follows:

```
GET /users/1234 HTTP/1.1
Host: webofthings.com
Accept: application/xml

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

<user>
  <name>Bob Doe</name>
</user>
```

```
GET /users/1234 HTTP/1.1
Host: webofthings.com
Accept: application/json

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

{
  "name" : "Bob Doe"
}
```

**Figure 3.3: Left:** An XML representation of a user. **Right:** A JSON representation of a user.

Recently, JSON has become the preferred[12] format to encode simple information as it can be easily parsed and processed using Javascript, and *"this combination of a simple and short specification with a dynamic programming language is probably the key to its ease of integration and adoption across the Web community"*, according to an online source[13].

In order to simplify the usage of content negotiation mechanisms, an alternative method for specifying the desired representation is to append the appropriate file extension directly in the resource URI (e.g., *spot11/sensors**.json*** or *spot11/sensors**.xml***). This is intuitive to use and extend, and can be used directly in browsers to facilitate debugging.

Such a layered approach for addressability, along with flexible control of data formats, allows a more decoupled protocol which enables a large variety of clients and applications to interact with a service in a consistent way. There are many benefits to provide several representations for the same resource. Providing simultaneously XML, HTML, and JSON allows not only Web browsers to consume information, but any application that supports HTTP. Low-semantic data formats should be favored over rigidly specified formats or newly-invented document types,

[12] See: blog.programmableweb.com/2010/12/03/json-continues-its-winning-streak-over-xml/
[13] See: http://blog.jclark.com/2010/11/xml-vs-web_24.html

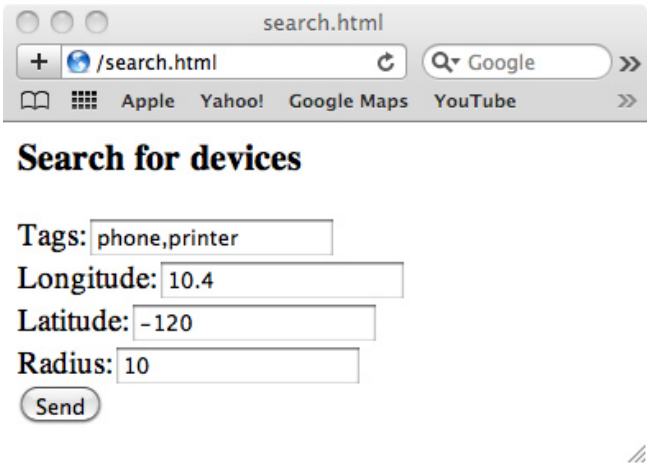



**Figure 3.4:** A simple HTML form to submit a query.

else the scalability and loose coupling would be drastically reduced. Likewise, no site-specific semantic constructs should be encoded in URI directly (for example defining custom, well-known URI that all sites are assumed to support), which would increase coupling between components.

### URI as User Interface

Most Web applications allow users to submit queries to retrieve specific information, usually via an HTML form to input query parameters as shown in Figure 3.4. When users click the "*Send*" button, the browser will send to the server an HTTP request that contains the form data. The HTML code behind this form is as follows:

```
<form action="search.html" method="GET">
       Tags:<input type="text" name="tags" /><br />
       Longitude:<input type="text" name="long" /><br />
       Latitude:<input type="text" name="lat" /><br />
       Radius:<input type="text" name="rad" /><br />
       <input type="submit" value="Send">
</form>
```

Note that the method used is `GET`, which instructs the browser to encode the query parameters directly in the URI (the query being anything after the "?" symbol). This generates the following HTTP request:

```
1 GET /search.html?tags=phone%2Cprinter&long=10.4&lat=-120&radius=10
2 Host: www.webofthings.com
3 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
4 Accept-Language: en-us,en;q=0.5
5 [...]
```

If the query does have an effect on the server (for example inserts a new device in a database), the query must use the `POST` verb and parameters must be encoded in the request body and not in the URI (in which case `GET` on the same URI would not be idempotent). This is usually done using `application/x-www-form-urlencoded` content type, and the equivalent HTTP request is as follows:

```
POST /search.html HTTP/1.1
Host: www.webofthings.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: application/x-www-form-urlencoded; charset=UTF-8
Content-Length: 49

tags=phone%2Cprinter&long=10.4&lat=-120&radius=10
```

Forms are important for the Web of Things, because the Web application that handles requests from an HTML form can be accessed from any HTTP client transparently over the same Web API. This allows to maintain a single API to handle both requests from humans via a browser or other machines, which drastically reduces discrepancies between different APIs for different applications or clients. Using semantic annotations, one can turn a simple Web page into a self-explanatory, general-purpose Web API that any program can read, interpret, and use. This allows the whole configuration API of device functions to be accessible at the same URI via forms in a browser or via shell scripts for example at the same.

### Payload Format

In many cases, interactions between a Web client and server can be quite complex and an efficient method to encoding structured data from devices is necessary. Various more or less complex encodings can be use to describe sensor data so that it can be understood and processed by other applications. For example, the Open Geospatial Consortium (OGC) proposed the Sensor Web Enablement (SWE) standards that allow to discover, access, and obtain sensor data and services [99]. Such standards allow to describe semantically sensor data in a machine-readable way, nevertheless they specify a custom, SOAP-based application protocol on top of HTTP which is not based on REST. This limits the native Web integration of devices by introducing a stronger coupling between components, which all need to implement SWE standards.

The simplest way to encode sensor information while retaining structure and semantics is to use *comma-separated values* (CSV), which is the most efficient encoding for plain text content. This makes it a convenient choice to encode data for embedded devices and also it is convenient

for developers to read and debug. Binary encoding of data can further reduce the data to transmit, but this requires extra processing power. The downside of CSV or binary encoding is their lack of flexibility, because a fixed and unambiguous data schema must be shared between all components and known in advance and hard-wired in the implementation. This reduces the loose coupling introduced by REST in the first place as all participants must rigorously stick to a common schema, therefore should be avoided unless necessary.

The alternative we suggest for Web of Things applications is JSON, which is particularly suited for Web applications as it is lightweight, portable, self-contained, and can be easily parsed in browsers using Javascript. It is a lighter alternative to XML which requires less processing power and bandwidth.

### 3.2.4 Hypermedia as the Engine of Application State

The fourth core aspect of REST is centered around the notion of *hypermedia*, in other words the idea of links to bind related ideas. Hypermedia was proposed in the early 1960's by Ted Nelson [190] as a generalization of hypertext that includes various media formats in addition to text, such as video, images, or sounds. Links have become highly popular thanks to Web browsers and are by no means limited to human use (for example UUIDs used to identify RFID tags).

Consider the following HTML fragment:

```
1 <html>
2       <h1 class="device-name">spot-living-room</h1>
3       Root <a href="http://webofthings.com/devices/spot-living-room" class="
            self">URI</a> of this device.
4       View the list of <a href='sensors/' class="sensors">sensors</a> and <a
            href='actuators/' class="actuators">actuators</a> on this device.
5
6       Or read the <a href='about/' class="info">documentation</a>.
7 </html>
```

With this initial representation of a device, one can easily "follow" links to retrieve additional information about sub-resources of the resource http://webofthings.com/devices/spot-living-room. In a WSDL file, everything is described in a flat structure and the whole document has to be retrieved and parsed every time the structure changes. Using such a tree-based model, every layer in the tree acts as a proxy that hides the layer beneath.

The application state refers to a step in a process or workflow (similarly to a state machine), and REST requires the engine of application state to be hypermedia driven[14]. This means that

[14]See: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

each possible state is available though a URI where a representation of the current state and possible transitions to other states is available. Resource state (e.g., the balance in a bank account) is kept on the server and each request is answered with a representation of the current state and with the necessary information on how to change the resource state (e.g., make a payment, file a check).

In other words, applications can be stateful as long as client state is not kept on the server and state changes within an application happens by following links (which meets the self-contained messages constraint). This way, each state becomes a RESTful resource that can be seen as a step in a workflow (see [123, 15]). This allows to scale servers, as new machines can be added to a cluster and there is no need to keep and synchronize client states across machines, as anyone can answer requests and no session context has to be maintained, only the state of the resources themselves.

In early RPC-based applications, a fat client was invoking functions on a central machine where a stateless application was running. The server simply answered requests by fetching some data in a database or other applications and return it to fat clients who handled the processing. This model was appropriate for fixed applications that were limited to a set of operations, but it was pretty difficult to maintain large deployments as any change in the backend (adding a new functions on the server for example or patching a bug) required all clients to be updated. In other remote applications (e.g., VNC or remote desktop applications), the entire client state was kept on the server and client only were used to display the state and and send commands to the server, which drastically reduced the server scalability. With Web applications, as the code was delivered dynamically from a central location and rendered locally in browsers, it became easy to upgrade both the clients and server, and in many cases state was maintained on the server which allowed lightweight clients to be used. Interestingly, more performant browsers along with frameworks such as Adobe Flex [1] or Microsoft Silverlight [27] made it easier to maintain application state on client-side along with fat clients, yet benefited from the code-on-demand feature introduced by the Web model. Nevertheless, these technologies are not fully integrated into the Web as they still require plugins to interpret them, very much like Flash.

## 3.3 Web-enabled Devices

After having described the core design principles of the modern Web architecture, we now show how to apply them for interacting with embedded devices. After that, we conclude this chapter with a discussion about the limitations of the current Web when used to build a scalable and efficient Web of Things.

A sensor manages data from one source (temperature, acceleration, pressure, humidity, sound level, etc.) via one or more channels (for example a 3D acceleration sensor has three channels).

As discussed in Section 3.2.1, one can easily map the functionality of any embedded device into a RESTful application. We define *Web-enabled devices* as any NED whose hardware and software resources (sensors, actuators, etc) are identified by URIs and can be manipulated using HTTP.

The first step in Web-enabling a device consists of making every property of a NED addressable via an URI. Devices usually have a set of common properties: an identifier (network name, URI, etc.), a description, a location in a three dimensional space (devices can be fixed or mobile), but also various sensors and/or actuators.

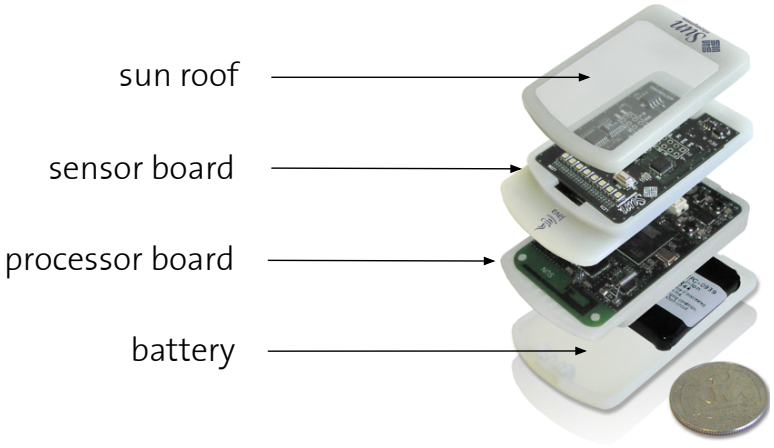



**Figure 3.5:** Sun SPOTs are small, battery-powered wireless devices with various sensors and actuators. They run an embedded virtual machine and can be programmed using Java.

We illustrate this notion with a Sun Small Programmable Object Technology (Sun SPOT, shown in Figure 3.5) device[15], which is a small, battery-operated wireless device running a Squawk Java Virtual Machine (VM) without any underlying operating system. Sun SPOTs can be easily programmed using Java, and support various functions such as mesh networking, public-key cryptography, and various data analysis routines. These devices were designed to overcome the challenges of traditional sensor network platforms by encouraging fast prototyping of powerful NED applications with advanced functions.

Every component of the Sun SPOT is organized into the resource tree depicted in Figure 3.6, where each sensor, actuator, and system property of the device is mapped to a hierarchical URI path. This allows to create a dynamic, resource-oriented Web API for Sun SPOTs, where every component can be directly accessed via its URI. For example, the URI of the 4th LED of a device is `http://[DEVICE_URI]/actuators/leds/4`, where `[DEVICE_URI]` is the **device root URI** (the equivalent of the default `[...]/index.html` landing page on a Web site). Resource collections are shown within curly braces as for example the LED number (e.g., `/leds/{led#}/`).

[15] Official Web site: `http://www.sunspotworld.com`

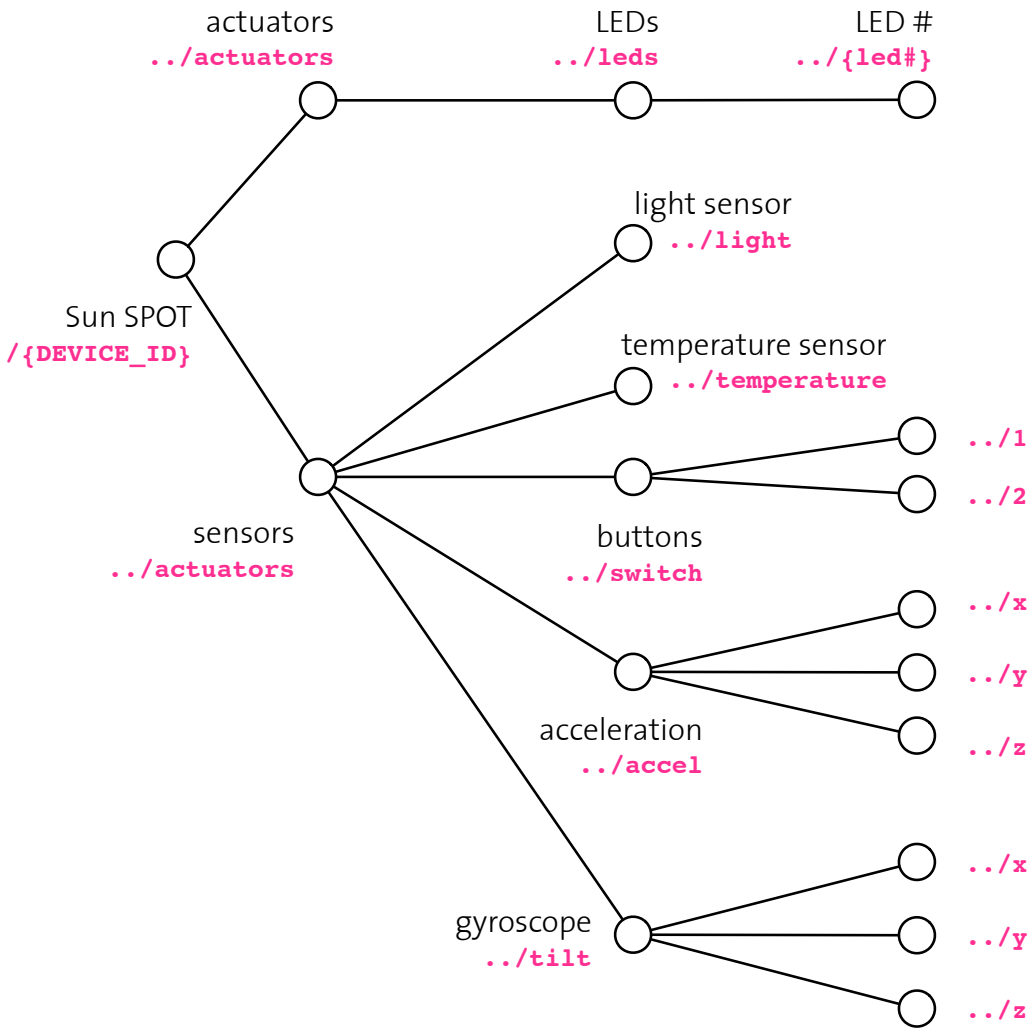



**Figure 3.6:** URI structure for the representation of a Sun SPOT and its associated sub-resources, sensors, and actuators.

As shown previously, the various functions and properties of a device shall be accessible via a Web API:

```
# GET to retrieve the list of events on device ID
http://[DEVICE_URI]/events

# POST to create a new "rule" on device ID
http://[DEVICE_URI]/rules/

# PUT to update the rule ID on the device
http://[DEVICE_URI]/rules/{rule_ID}
```

If the device is directly connected to the Web, the `[DEVICE_URI]` will be directly its IP address, such as `http://192.168.44.32` or `http://device.webofthings.com`. It can also be a more complex URI path, for example when a Web gateway is used as proxy for several devices that

are not directly connected to the Web (see Section 4.3), as follows:

```
# A list of devices connected to a gateway
http://[GATEWAY_URI]/devices

# GET to retrieve the device page of device ID
http://GATEWAY_URI]/devices/{DEVICE_ID}
```

A typical complete URI of the light sensor on a Sun SPOT (named `spot31`) attached to a gateway will look like this: http://gateway.webofthings.com/devices/spot31/sensors/light.

Once every component of the devices is assigned a unique URI, the second step is to implement the basic HTTP verbs to interact with each resource, as was described in Section 3.2.2. Once a basic RESTful API has been designed for a device, more advanced features can be implemented on top. For example, some resources could support eventing, authentication, or resource discovery mechanisms.

As will be shown in Section 6.1, the ability for devices to generate events upon specific preconditions is necessary for many applications. Rather than continuously polling sensor readings on a device and analyze data on a remote application, it would be efficient to register simple rules directly on devices who will trigger an event handler in another application. HTTP was not designed to work as an asynchronous notification systems, and by default it does not support push-based interactions. As will be explained in detail in Section 6.1.2, various mechanism are possible to implement Web-based push notifications on devices. The simplest solution is to use HTTP callbacks, that is enabling devices to act as clients that can post notification onto callback URL specified at registration time. The callback is a simple HTTP request handler that can process incoming HTTP requests from devices.

Once the API is defined for each resource, various encodings can be used to represent them. If the device should be directly accessible via a Web browser, an HTML page with Web forms should be designed for each resource to interact with them. In the case where only applications will interact with devices, XML or JSON formats are sufficient. For example, events from devices can be encoded using the following JSON object:

```
{event:{
        timestamp:309482309, /* if not specified, assigned by the server */
        type:sample|error|trigger|critical|heartbeat,
        value:{
                model:"SunSpot",
                vendor:"sun",
                description:"this is a sunspot",
                vendor-url:"http://webofthings.com/devices/sunspot/"
        }
}
```

Rules do not contain an action, they only generate notifications that are forwarded to a single endpoint (in the cloud or on a gateway), where the actual event handling (action taking) takes place. This allows to use a publish-subscribe notification mechanism directly on devices which decouples further the event from its reaction (device only needs to know one endpoint to notify – or none if the gateway offers an endpoint by default to devices), and the actual notification dispatching to subscribers operation is "outsourced" to an external application.

Finally, the whole state of the application should be encoded in each representation, so that users (humans or machines) can find all the possible functions available in each state encoded using hyperlinks.

## 3.4 Case Study: Physical Mashups

RESTful Web services are more loosely-coupled than their SOAP-based counterparts [197]. In practice, this means that services based on RESTful APIs can be re-used and re-combined in a quite straightforward manner. Thanks to the simplicity of HTTP and the wide availability of HTTP libraries and clients, we believe that Web technologies are an excellent way to interconnect the physical world with applications.

A Web mashup is an application based on common Web languages and tools, such as Java Script and XHTML, that combines data from two or more different sources on the Web (*e.g., RESTful services*) into an integrated application. Physical mashups [139] are composite applications involving devices that are part of the physical world such as sensor networks. For example, a physical mashup could integrate electricity measurement data from a sensor network, with electricity prices from an energy market data source, with the Google maps data to form an application where a real-estate company can monitor electricity usage and costs on a map.

```
1 while [ true ]
2 do
3     RANDOM_LED=$[ ( $RANDOM % 8 ) ]
4     RANDOM_R=$[ ( $RANDOM % 255 ) + 1 ]
5     RANDOM_G=$[ ( $RANDOM % 255 ) + 1 ]
6     RANDOM_B=$[ ( $RANDOM % 255 ) + 1 ]
7
8     curl --basic --request POST --data "switch=on&Green=$RANDOM_G&Blue=$RANDOM_B
          &Red=$RANDOM_R" http://webofthings.com/spot31/actuators/leds/$RANDOM_LED
9 done
```

**Listing 3.1:** A shell script that blinks periodically a random LED of a Sun SPOT by assigning it a random color (using a random RGB combination).

## 3.5 Discussion

In this chapter, we have presented the basis of the Web of Things, as an accommodation of the core design principles of the Web architecture to support the interaction patterns required by NEDs. Extending the Web to the real world by enabling Web-based interaction with embedded devices will greatly simplify the development, deployment, and maintenance of pervasive computing applications. In practice, when using Web standards on embedded devices, several limitations prevent the implementation of elaborate and scalable distributed sensing applications. In particular, various shortcomings of current Web tools and technologies need to be addressed with the requirements of scalable and participatory ecosystems for devices. Besides, various practical issues need to be tackled to incentivize the participation of developers to integrate devices and services into the Web of Things.

The problems that we explicitly address in this thesis towards the development of such a participatory ecosystem are the following:

- **Generic framework for Web-enabling devices.** Current prototypes implement Web patterns directly on specific devices, in a fairly rigid and custom manner. A generic framework to easily integrate various devices into the Web of Things without reimplementing the same functions anew for every different device is missing. To address this problem, we propose in the next chapter to use *gateways*, which are extensible software applications that bridge NEDs to the Web regardless of the actual communication protocol used natively by the device. Support for various functions (caching, optimization, data aggregation, etc.) and devices can be easily added to gateway via plugins, thanks to the modular software architecture design.

- **Extensible Web infrastructure.** Gateways can then be interconnected to form a scalable infrastructure where new devices and applications can be added easily thanks to the RESTful interfaces used throughout the Web of Things components, as shown in Chapter 5. The infrastructure supports various service that facilitate the management and usage of heterogeneous devices in large applications. In particular, we propose solutions for semantic description and discovery of devices in a way that supports the physical location of device and their services, and also users. A query mechanism to support location-aware search of devices is also proposed. Obviously, the originality of our contribution relies on the fact that our infrastructure builds naturally upon the existing network in place, by being fully integrated in the Web thanks to its RESTful interface.

- **Lightweight Web messaging.** Efficient solutions for event-driven communication between devices that fully comply with the REST and HTTP principles are lacking in the Web of things community. Furthermore, no support for data streaming on the HTTP level (HTTP-push) is provided/supported by existing solutions. Future NED applications, will not only require to support scalable, push-based notification mechanisms that

are fully-based on HTTP, but also directly integrate with future Web applications to collect, analyze, visualize, and share sensor data over social networks. This limitation will be addressed in the first part of Chapter 6, where a minimal Web push notification mechanism is proposed.

- **End-to-end application development framework.** Having an infrastructure to integrate, search, and use heterogenous devices over the Web is necessary, but not sufficient to develop complete applications for end-users. In particular, a programming model that supports declarative queries over the REST will enable Web developers to easily access, program, and integrate streaming data into traditional Web applications. This is discussed in the second part of Chapter 6, where we provide a modular distributed application framework to collect, process, share, and store real-time data using only Web standards.

# CHAPTER 4

# Web-enabling Embedded Devices

*"A great pleasure in life is doing what people say you cannot do."*

Walter Bagehot (1826-77)

Building the Web of Things requires to implement the various patterns described in Chapter 3 to interact with the devices. In this chapter we present and evaluate the various approaches to connect devices to the Web and make their functions accessible via a standard RESTful API. In particular, we propose to use a gateway-based approach to enable Web interaction with NED regardless of the actual communication protocols they support, and further show how performance and functions of applications can be augmented using a smart proxy to handle complex processing.

## 4.1 Web-enabling Devices

We define a *Web-enabled device* as any NED whose sensors, actuators, and properties are identified via an URI and can be manipulated using HTTP. As pointed out by Wilde [244], devices that are Web-enabled *by design* would facilitate the integration of NEDs with existing Web content and applications as there would be no need for any additional API or descriptions of resources/functions. Another advantage to use Web protocols to interact with embedded devices is that one inherits many of the mechanisms that made the Web scalable and successful such as caching, load balancing, indexing, and searching, as well as the stateless nature of the HTTP protocol. Besides, one can also leverage search engines to register, index, and search for physical devices and service (e.g., environmental monitoring sensors), for example by using semantic annotations to describe the functionality and interaction possibilities of each device.

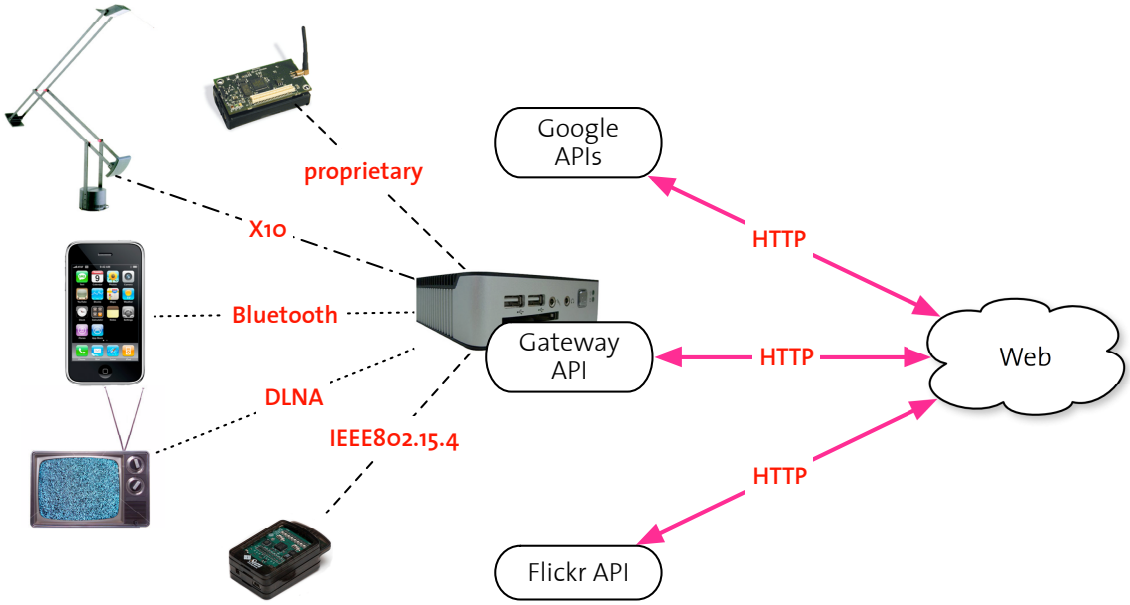



**Figure 4.1:** A *gateway* is a Web application that bridges embedded devices with the Web. By abstracting the various low-power protocols used by devices behind a RESTful API, one can interact with devices transparently using HTTP – as if they were running a Web server and were directly accessible over the Internet (i.e., have a public URI).

Embedded devices usually have limited resources, thus often call for optimized protocols to exchange data. Additionally, as HTTP or IP might not be available or appropriate for such devices, gateways might be required to enable Web-based interactions with low-power devices. A *gateway* (cf. Figure 4.1) is nothing more than a Web application that enables access to heterogenous devices through a simple and uniform RESTful API. This hides the complexity of the various protocols used natively by devices (such as Bluetooth or ZigBee). The gateway application is lightweight enough to run on any computer with a TCP/IP connection that can host a Java virtual machine, such as programmable wireless routers, network-attached storage (NAS) devices, or networked media players.

In many cases – even when an HTTP server can run on a device – serving HTML pages directly from the device does not make much sense, unless users want to access directly a particular device directly with their browser. When users want to interact with a set of devices and only get their aggregated data, then devices can serve much lighter machine-readable formats, such as JSON or XML. In that case, the data collected from a multitude of devices can be more easily aggregated, filtered or processed at intermediate nodes in the network or more powerful gateways. This aggregation is actually significantly facilitated when using REST in place of other protocols – as intermediary components are supported to reduce interaction latency, enforce security, and encapsulate legacy systems, which are all desirable features for NED applications.

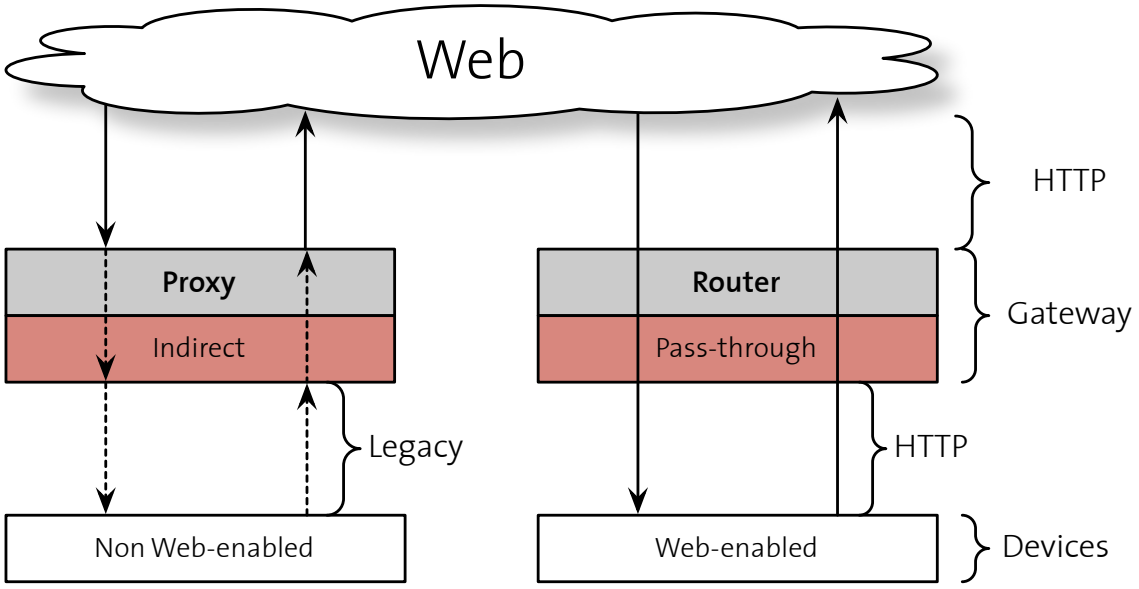



**Figure 4.2:** Web-enablement of devices. **Right:** devices run an embedded RESTful Web server on board, so the router only forwards messages across the two interfaces. **Left:** devices do not support directly HTTP, therefore a protocol translation is required.

As illustrated in Figure 4.2, there are two methods for *web-enabling* devices:

1. **Router-based:** end-to-end HTTP enablement via embedded Web servers running on the device. Requests are only routed from the Web the different physical (transport) interfaces without any application protocol translation (see Section 4.2).

2. **Proxy-based:** a proxy translates HTTP requests from the Web into the actual transport and application protocol. For example, the device uses a proprietary application protocol over Bluetooth or ZigBee (see Section 4.3).

The actual location where the Web server is running (directly on the device or only on the gateway) does not matter much (obviously besides performance issues), as long as the interactions remain transparent to the client. This property is guaranteed by the layered system property of the HTTP protocol, which allows a more flexible system design.

## 4.2 Case I: Router-based Enablement

The simplest method to connect NEDs to the Web is to implement a lightweight HTTP server directly on the device. Although using HTTP instead of optimized legacy communication protocols requires more resources to transmit the same amount of information due to HTTP's verbosity, there are many benefits for devices to understand HTTP natively as ad-hoc interaction with unknown devices becomes greatly facilitated. This approach is the most desirable because there is no need to use an intermediate to translate HTTP requests from Web clients into the appropriate protocol for each device (obviously beyond switching data packets across physical transport interfaces, e.g., from Ethernet to ZigBee).

As many powerful NEDs have an Ethernet of Wi-Fi connectivity, implementing a Web server on them is straightforward. However, the devices used in traditional WSN applications have much more constrained resources and are usually battery-powered, in which case a protocol that minimizes data to transmit should be preferred. Nevertheless, recent work has shown that embedded Web servers on resource constrained devices is feasible with a very low footprint [112, 115, 142]. Besides, it is likely that in the near future most NEDs will have native support for TCP/IP connectivity, therefore a Web server on each device is a plausible scenario.

### 4.2.1 IP-based NEDs

Internet connectivity has become increasingly cheap and ubiquitous and most mobile devices today have support for TCP/IP networking and even embedded HTTP clients. IP-based communication to connect embedded devices has become a viable solution even for sensor networks, and lightweight IP implementations have been successfully implemented on different platforms [110]. Support for IP connection is desirable because it allows sensor networks to become an integral part of the Internet and behave just like any other computer connected to it. In particular, tools such as ICMP could come for free, which would significantly facilitate the development of larger WSN applications [154]. Unfortunately, TCP/IP only allows connectivity at the physical layer (how they *exchange data* with each other), but a common application layer (how they *understand* each other) is still needed for a global interoperability of IP-based sensor networks.

Many NEDs increasingly come with an embedded Ethernet or Wi-Fi interface such as printers, alarm clocks, digital photo frames, or networked multimedia player. In spite of its drawbacks, Wi-Fi is becoming popular for embedded applications as it reduces significantly the cost to network devices, because the technology is well-known and the equipment required is inexpensive and omnipresent. Recently, several companies have proposed Wi-Fi embedded chips that can be directly connected to the Web (Microchip [25], Lantronix [22], Gainspan [14], or Round solutions [46]), or even products based on such chips (Aginova [2]). This makes it easy to

implement full HTTP servers directly on devices, especially when only a Web API is needed, without also serving HTML pages and multimedia content.

### 4.2.2 Embedded Web Servers

HTTP was designed as an application protocol, not a transport protocol. Even though HTTP is mostly used over TCP/IP, nothing prevents HTTP requests to be transported over other protocols such as 802.5.14 or ZigBee, therefore IP-based sensor networks are useful – but not required – to "give the impression" that devices are directly connected to the Web.

Many NEDs can serve directly Web pages, and various lightweight HTTP servers for embedded devices have been designed as for example RESTduino [42], SMEWS [114], or Lighttpd [24]. Because of the constrained resources on sensor nodes, some components such as the HTTP server or XML parser are optimized or limited to set of minimal functions. As a proof of concept, we have implemented oREST (optimized REST), a lightweight RESTful framework running both on TinyOS and Contiki that allows to send HTTP requests to resource constrained sensor nodes (TMotes, see Section 4.2.3). In essence, oREST is similar to RESThing [250], but uses an optimized RESTful application protocol (easily mappable to HTTP by a gateway) adapted for devices, very much along the lines of CoAP [211]. This work has been used in [161, 163] to explore Web-enabled devices, but this thesis does not address the implementation of an embedded Web server, and more information about native Web integration of embedded devices can be found in [201, 251, 250].

According to [154], it is likely that in the near future a lot of embedded devices will natively support TCP/IP connectivity as 6LoWPAN introduces an adaptation layer that enables efficient IPv6 communication over IEEE 802.15.4 LoWPAN links. This approach is desirable from an architectural standpoint as devices can be directly integrated into the Web and HTTP requests must no longer be translated to device-specific protocols [139].

### 4.2.3 Evaluation

In this section we measure various aspects of Web-enabling objects using an embedded Web server to evaluate and discuss the feasibility and performance of using HTTP to interact with embedded devices.

#### Hardware and Software

We have explored the router-based Web enablement with a simple gateway that connects classic sensor nodes the Web. Using the TMote nodes [28] (Device No. 4 in Figure 2.1), we have implemented a simple application where Web clients can send read/write HTTP requests to different

nodes that sample the environment. TMotes are equipped with a 8MHz TI MSP430 micro controller with 10 kB RAM, and an integrated antenna with a 250kbps 2.4GHz IEEE802.15.4 Chipcon Radio.

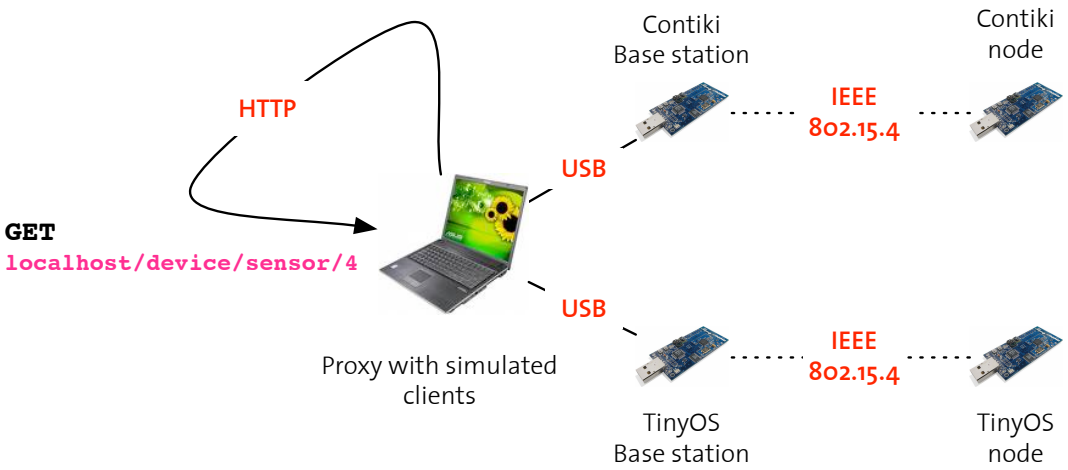



**Figure 4.3:** System setup for the proxy-based evaluation. The resources of TMotes are randomly accessed by Web clients simulated on a machine

The experimental setup we used is shown in Figure 4.3, where the proxy has been installed on a laptop (Intel Core Duo 2.2 GHz). To illustrate the flexibility of our approach we used both TinyOS and Contiki nodes running oREST (a minimal subset of HTTP that uses only the server URI, a service name, an HTTP verb, and an optional list of key-value pairs as parameters). Two TMotes were plugged in the USB ports of the gateway to serve as base stations – one for TinyOS and the other for Contiki. Both Base Stations forwarded IEEE802.15.4 wireless packets from/to the gateway via the serial-over-USB port. A simple multithreaded application that simulated 50 concurrent clients was run on the same machine, and each client accessed once per minute a random service on one of the devices. The application was run for 15 minutes (totaling 750 requests).

Figure 4.4 shows a typical measured response time for 400 requests, and one can see that even though most requests are answered within 100-350 ms, there is quite some variation up to 800 ms. The cumulative density function (CDF) of the response time is shown in Figure 4.5 (No cache), and one can see that 99% of the request were answered within 737 ms. This shows that even with no optimization on router beyond simply transforming HTTP requests into oREST and buffering requests for the same device, a sub-second latency to query resource-constrained devices using a subset of HTTP is feasible. Afterwards, we have done the same experiment again but with using a cache on the router to answer read requests from devices (sensor readings were cached for 1 minute), and the CDF is in Figure 4.5 (Cache). In this case, 95% of all messages were returned within 148 ms (mean response time 43 ms). This is an excellent result which shows that with additional optimizations, a gateway could efficiently handle many concurrent users accessing a sensor network, while minimizing the actual communication with devices.

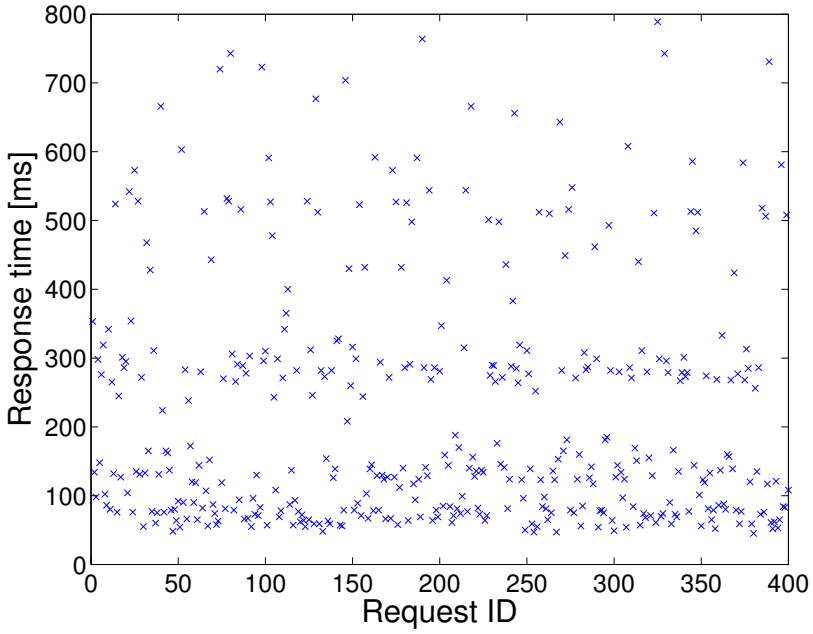


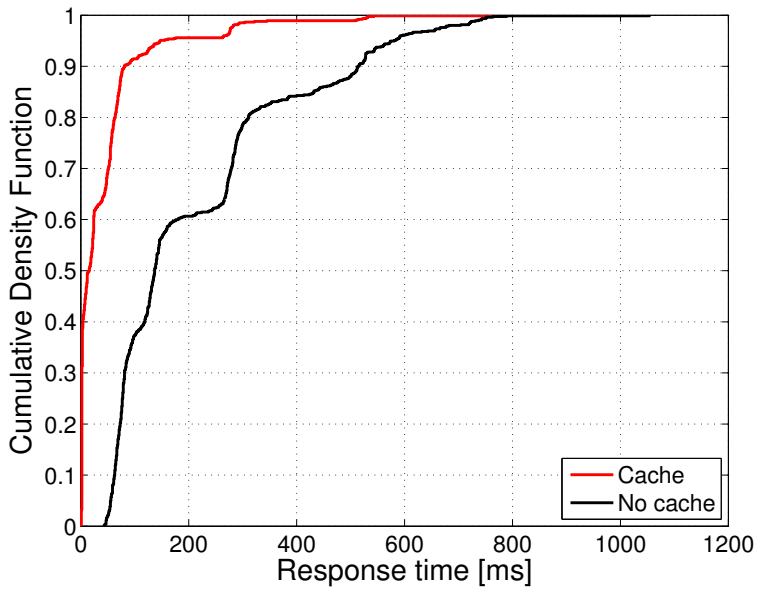

**Figure 4.4:** Measured response time of 50 clients accessing randomly a service on two devices (each client does it randomly once per minute).

These aspects are further explored in the next section.

## 4.3 Case II: Proxy-based Enablement

Embedded HTTP servers running on resource-constrained devices are not always possible or desirable because of the large overhead of HTTP. In this case, Web integration is done via a *proxy* (also called *gateway*), which is a stand-alone application that abstracts the actual communication protocol used to interact with devices (e.g., Bluetooth or ZigBee) behind a RESTful API. From the Web clients' perspective, the actual Web-enablement process is fully transparent, as interactions are HTTP regardless whether the RESTful server resides directly on devices or on an intermediate gateway.

In many cases, gateways are a better integration choice because they can virtually support any kind of device regardless whether the device supports HTTP or not. Besides, a gateway can be the best (or only) solution for Web-enabling an existing NED deployment with little or no change to the system in place. As part of the research presented in this thesis, we have built various gateways [243, 199, 101, 184], and Web-enabled various devices ranging from sensor networks [142], to energy meters such as Ploggs [162, 240], to RFID tag readers [138], or even robotic development kits such as LEGO Mindstorms [186].




| | Cache | No cache |
|---|---|---|
| Min | 1 ms | 43 ms |
| Max | 768 ms | 1054 ms |
| Mean | 43 ms | 215 ms |
| q80% | 64 ms | 310 ms |
| q95% | 148 ms | 581 ms |
| q99% | 510 ms | 737 ms |

**Figure 4.5:** Response time CDF for 50 clients accessing randomly sensors on two TMotes running a oREST (each client issues one `GET` request per minute). When a cache is used, the most recent sensor value is returned from the cache if available, else every single request is forwarded and executed on a device. qX% refers to the Xth quantile (e.g., q99%=$Y$, means that 99% of the requests have a response time below $Y$.)

### 4.3.1 Proxy-based Architecture

Our proxy has been implemented in Java using a modular approach. As illustrated in Figure 4.6, three central modules have been implemented. The *Device Layer* implements the actual connection between devices and the the proxy using connectors (TinyOS/Contiki), along with their management. The *Control Layer* is the core applications that handles and does the protocol translation between Web requests and issuing their corresponding low-level messages. Finally, the *Presentation Layer* handles requests from the Web, and implements both the Web API and an HTML-based GUI to control and visualize the status of the sensor network through any Web browser.

**Device Layer** handles the actual communication with the devices. Using a connector architecture, it can interface with various WSN devices, where each connector (driver) implements the native communication protocol used by a device (for example TinyOS and Contiki). First, drivers handle the low-level discovery of devices using a custom discovery procedure for each connector. Then it keeps monitoring the presence of these devices using a heartbeat mechanism, removing them in case they do not respond within a predefined time.

As soon as a new device has been discovered, a new *device thread (DT)* is created for that device, which acts as the internal (local) representation of the device. We have used threads because

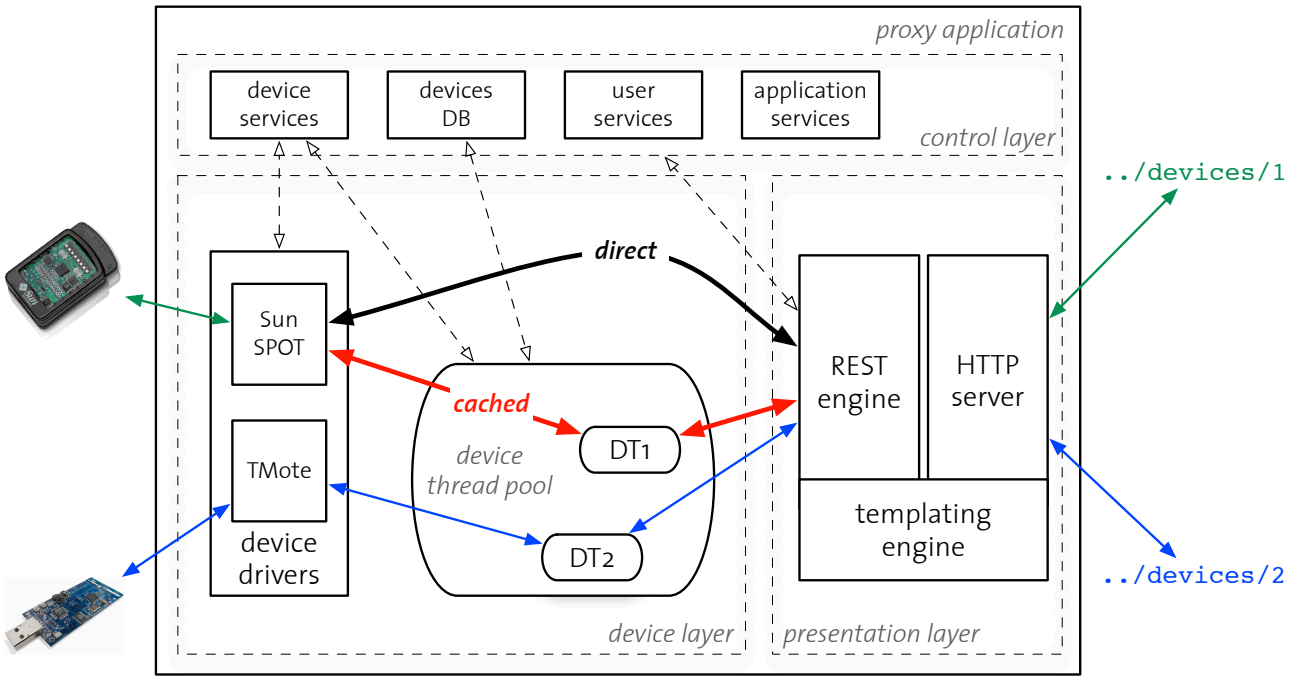



**Figure 4.6:** High-level architecture of the proxy, with interactions between the control, presentation, and device layer. Various devices interact via specific drivers, and for each device an associated device thread (DT1,2) is associated. Each device is then accessible via specific URIs.

they simplify the management and implementation of device-level interaction, and offers a generic and performant metaphor to interact with objects. The thread handles all the events and messages received from the device, and sends them to the Control layer using callbacks (handlers). The same threads are used in the whole system and all have the same behavior, API, functionality, independently of the connector, therefore they are protocol agnostic. Connectors are simple software modules that must implement a specific API used internally and implement the various functions and callbacks defined (details about the implementation can be found in [101, 184]).

Every thread stores information about its own device, a description of the device (device metadata, see Section 5.3.2) and the services and functions offered by the device, along with the API description of the services. Besides, threads also handle (and buffer) all the requests from Web clients to the devices. Many other features can be implemented in each driver (for example WSN transmission failure, load balancing, etc.) or directly a gateway module (authentication, multi-device data aggregation, debugging, etc.) which can significantly improve the performance of the WSN or simply augment a basic WSN deployment with various features.

**Control Layer.** This layer is the core processing unit of the proxy. It manages the threads spawning and freeing them as new devices appear or disappear, and coordinated the interactions between the device layer and presentation layer. This module is responsible for the interaction

between the upper layers and the device threads. It maintains a list of all the devices which are bound to the system, as well as all the information covering the services they offer. Higher layers are directly informed through querying that module about all the necessary information concerning the devices connected to the system. In other words, it offers an abstract way of reaching the devices by means of a simple internal and generic interface. This module runs in the background, initializes and monitors the various modules, and other "administrative" tasks.

**Presentation Layer.** This layer is the Web-facing part of the proxy and handles the HTTP interaction from Web clients. It implements a Web server that handles all the incoming HTTP requests from the Web. In particular, it implements a REST engine (RESTlet [43]) to enable RESTful access for interacting with devices. More details about the presentation layer are presented in [243, 101].

### 4.3.2 A Smart Proxy for the Web of Things

An initial version of the Sun SPOT gateway has been implemented in the context of the EU project Socrades [199, 142], where Sun SPOTs were running a full HTTP server that hosted HTML pages. Because of the various performance and stability issues of this gateway, an improved version has been designed and presented in [184]. This improved version has been used as a component of the proxy for Web-enabling and augmenting the Sun SPOT platform presented in this thesis.

The Sun SPOT driver is based on a *synchronization-based architecture* where a virtual copy of the device is maintained on the gateway and the status is updated (pushed) by the device. Updates are sent either sporadically when specific events are detected or periodically according to a fixed synchronization interval depending on the specific applications and devices at hand which calls for a tradeoff between data *freshness* (i.e., how old is the cached sensor value), latency, and battery lifetime. This allows to decouple the HTTP requests to the proxy and the actual communication between the device and proxy, which doesn't need to be HTTP-based. Every HTTP request from clients that access resources from devices (for example reading the temperature sensor using a `GET`) is served directly by the proxy, which returns the most recent state from the virtual copy that acts a cache. This way only a fraction of all read requests will be forwarded to the device and the communication between devices and the proxy is minimized.

The Sun SPOT runs a small java application that handles the communication with the proxy to update its virtual representation (datagram-based on top of IEEE 802.15.4). A simple discovery module looks for proxies nearby on a predefined port and uses a handshake protocol to negotiate the port that will be used to send updates to the proxy and receive commands from it. Once the connection with the proxy is open, the SPOT will periodically monitor the sensors

and update the proxy periodically or every time an event is detected. The device will push its newest status update encoded using JSON, and the acknowledgment for the update can also contain requests that were buffered for the device from the proxy. Using only device-initiated communication allows devices to use low duty cycles as they do not need to listen for incoming requests all the time.

In this section, propose the notion of *smart proxy*, which is refined version of the gateway presented in Section 4.3, but has been implemented using the OSGi framework, and augmented with various additional functions. A modular software architecture has been used and the overall software architecture is shown in Figure 4.6. The various modules interact with each other using OSGi Declarative Services (DS), which is a mechanism to share services offered by various bundles inside the same OSGi framework. More details about the implementation of specific parts of the smart proxy can be found in [161, 184, 101].

The main function of the *device driver* bundle is to handle the actual communication with the Sun SPOT, which is based on JSON-encoded messages. Each device driver must implement a generic interface declared as an OSGi DS interface that can be accessed by any module of the smart proxy. This interface defines generic methods to retrieve the list of connected devices to the driver, their resources and informations (metadata), and generic read and write methods for any of the resources offered by each device. It also contains more generic driver-specific functions, for example the synchronization interval or various key-value pairs to specify how the driver should connect to the physical interface (for example on which serial port is connected the base-station).

New devices are found by listening to a pre-defined port for connection requests from devices. For every new device found, a connection attempt will take place which consists of sending to the device in response to the connection parameters chosen by the proxy (e.g., the datagram port that will be used for that device and the synchronization interval), and a *device thread* (from the device threads pool) is allocated for that device. The thread will create and keep updated the local representation of the device, buffer incoming write HTTP requests for the device, and answer read requests directly with the data from the representation similarly to a cache.

### 4.3.3 Evaluation

Although a Smart Proxy is not necessary when devices support HTTP and/or TCP/IP directly, in many cases proxies can help to improve the performance of an existing NED deployment by handling locally processing and scalable notification when many users want to access sensor data and services. Besides, Web integration can be further improved by offering additional functions such as Web-based authentication, social-network integration, data caching, or simply load balancing.

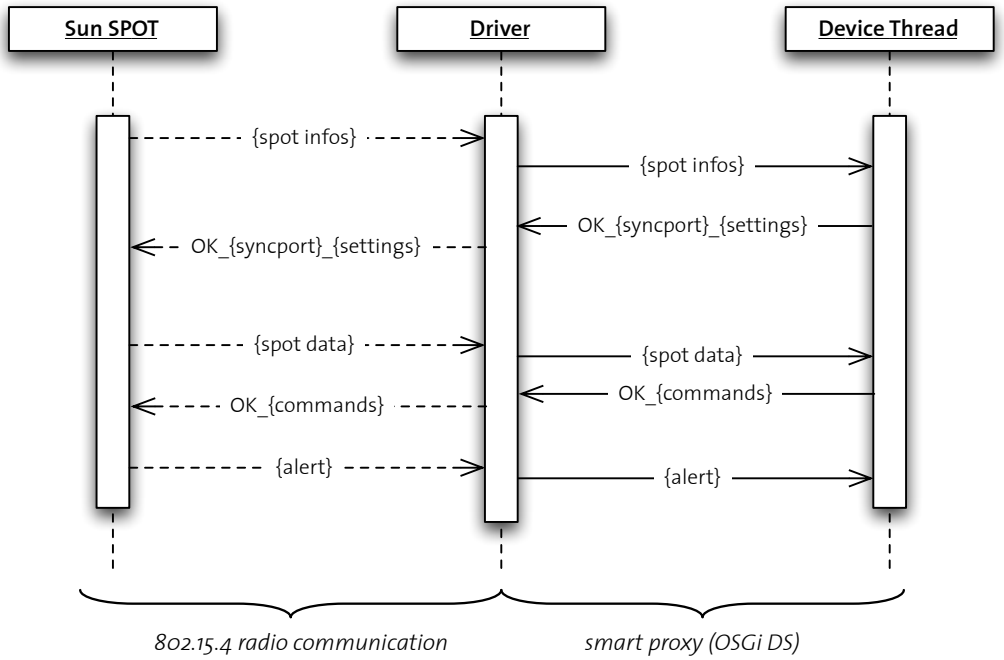



**Figure 4.7:** Handshake and synchronization between a Sun SPOT and the Driver Core Bundle.

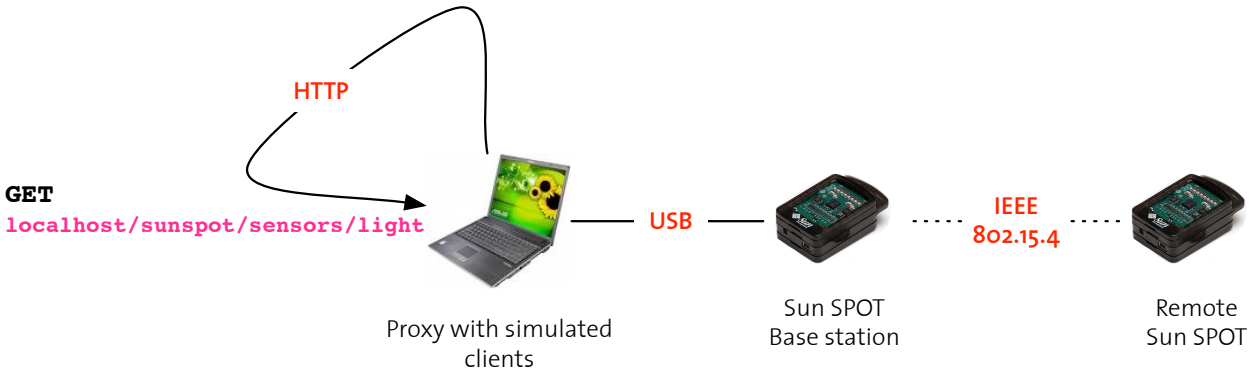


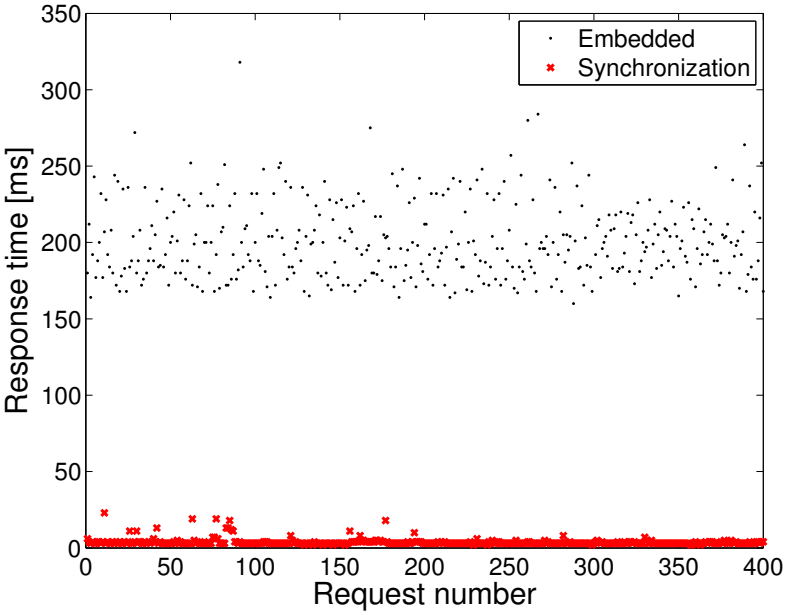

**Figure 4.8:** Experimental setup for the proxy-based evaluation. The resources of a Sun SPOT are randomly accessed by Web clients running on the gateway machine.

### Response Time

In many use cases, human users are the main actors that read data from and send commands to NEDs. A major challenge in early Web design has been responsiveness of Web sites, which was perceived as an essential component in interaction quality. As long as the response time can be below 100 ms, users hardly perceive any latency[1].

As discussed in the previous section, the actual response time for accessing NED can vary largely depending on many factors, a major one being the instability of the radio communication, which might require elaborate acknowledgement and retransmission mechanisms. If every single

[1]See discussion: http://www.useit.com/papers/responsetime.html.




| | Sync. | Emb. |
|---|---|---|
| Min | 2 ms | 159 ms |
| Max | 36 ms | 3075 ms |
| Mean | 3.9 ms | 203 ms |

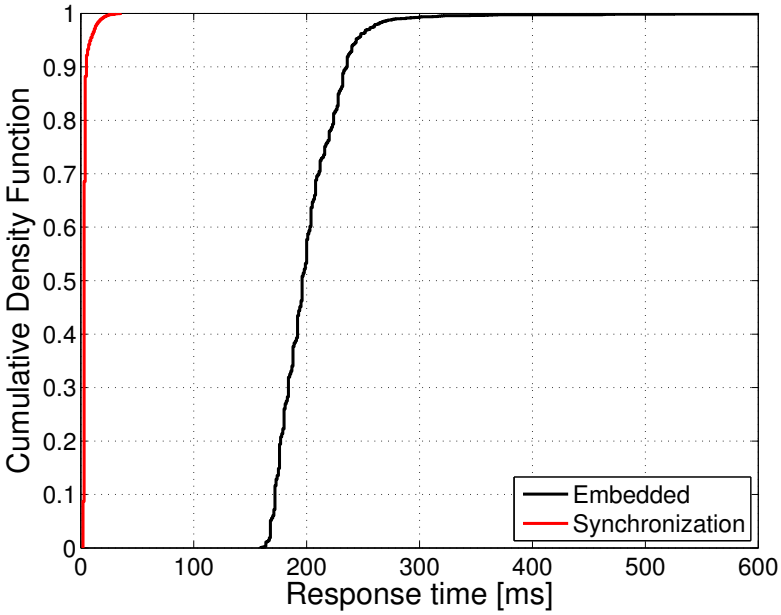
**Figure 4.9:** Comparison of the two driver architectures with respect to the round-trip time of local HTTP requests (400 requests shown out of 5000).

request has to be executed on the device, the response time will grow significantly for many concurrent users, as all write requests have to be queued to be transmitted to the device one at a time.

In many pervasive computing applications, sub-second response times are necessary. We measured the actual response time via an experiment where 5'000 consecutive read requests are sent to a Sun SPOT one hop away from the proxy, as seen in Figure 4.8. The requests are all run from the same machine and are simple sensor read requests ("`GET http://localhost/sunspot/sensors/light`"). Figure 4.9 shows the measured response time for a subset of 400 requests (embedded), and the extrema is between 159 ms (min) and 3075 ms (max), with an average response time of 203 ms. The distribution of measures is shown in Figure 4.10, where one can see that over 99% of the request are answered within 244 ms.

We contrasted these measures with the average response time using the same setup, but with a sync-based architecture to cache the most recent state of the device. As explained earlier, the proxy receives periodically updates from each device and keeps an local copy of the device data that is used to serve read requests. The synchronization-based drivers differs from a cache because all the data sent from devices via update messages is kept in memory regardless if they are requested or not (caches only keep data already served to clients, and this for a limited amount of time). In this case, the average response time was 3.9 ms (min 2 ms, max 36 ms), and 99% of requests are returned within 19 ms.

Our measurements show that extremely low response time can be obtained. As all the data is served directly by the proxy application, one can leverage easily Web load balancing techniques




| | Sync. | Emb. |
|---|---|---|
| q80% | 4 ms | 224 ms |
| q95% | 9 ms | 244 ms |
| q99% | 19 ms | 282 ms |

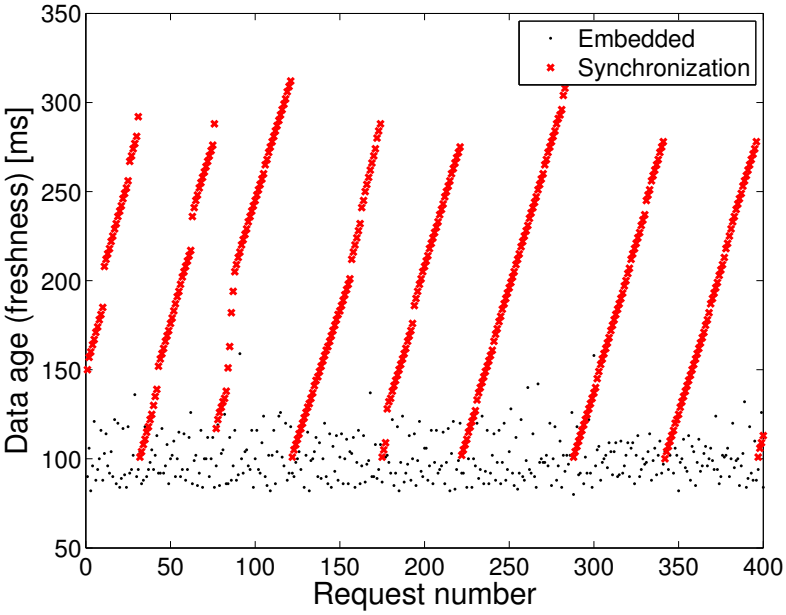
**Figure 4.10:** Response time CDF for 5'000 consecutive `GET` requests on the machine hosting the proxy.

to support thousands of concurrent users. Obviously, this is not possible for write access, where commands have to be queued.

Our results do not directly leverage the caching techniques used commonly on the Web (see Chapter 13 in [121]), but future work in this area will enable direct integration with the Web. This allows to leverage standardized caching control mechanisms used for traditional Web sites (ETag headers) directly within the Web of Things, this way applications and devices can specify individual preferences regarding maximal response time and data staleness they are willing to tolerate.

### Age of Retrieved Data

Even though caching by the smart proxy reduces considerably the response time, another major issue to consider is the age of the data clients are willing to tolerate. Pushing updates periodically from the devices minimizes network traffic, but the downside is that the data in the cache grows old until a new update arrives. The maximal age of data is bound by the update frequency (assuming no transmission delays occur), and we have compare the data age for both driver architectures. Using the same experiment as above, we have also measured the data age, which gives the typical pattern shown in Figure 4.11. With the embedded architecture, the data age varies little around the average radio propagation delay and has the advantage to minimize the data age as every response contains the most recent sensor reading.



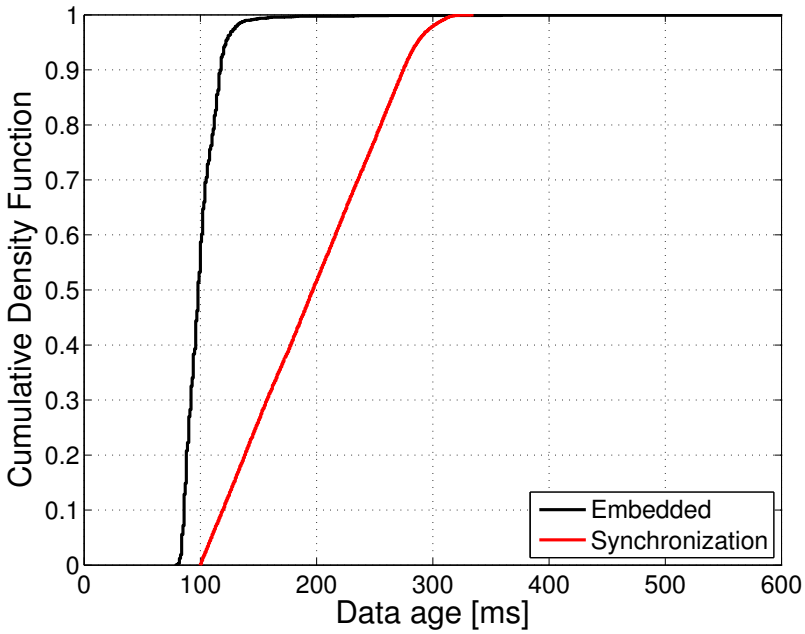

| | Sync. | Emb. |
|---|---|---|
| Min | 100 ms | 79 ms |
| Max | 335 ms | 1537 ms |
| Mean | 197 ms | 101 ms |

**Figure 4.11:** Age of the sensor data when using HTTP and sync-based architectures (400 requests shown out of 5000). The round-trip time of the Sun SPOT communication is approximately 200 ms.

One can see in Figure 4.12 the CDF of the data age for both architectures. In the worst case, the age of the sync architecture is bound by the update period (200 ms), which added to the transmission delay gives an upper age limit of 308 ms for 99% of all requests. In comparison, the age of data in embedded architectures is mainly bound by the radio transmission, in which case the maximal age for 99% of all requests is 140 ms.

## 4.4 Case Study: Energy Monitoring Mashups

As a proof of concept for Web-enabled devices, we have implemented a fully Web-based energy monitoring application that leverages the advantages offered by the Web protocols. A handy feature of HTTP is the ability to encapsulate legacy systems behind a uniform HTTP interface. Because of that, support for HTTP on sensor nodes is not necessary and traditional WSN protocols and applications can be used for the sensor network. The application consists of two networks of TMote sky motes that simulate smart meters measuring the energy consumption of electronic appliances on two floors of a building. In this case, a **proxy** (Figure 4.13, Floor A) must be implemented to translate HTTP request into the legacy WSN protocol and vice-versa, which can be a tedious process. Each time a device has a new reading it will issue a HTTP request (either directly or generated by the proxy) and `POST` its data on the internal Web interface of the **gateway**. The internal part of the gateway implements a simple publish/subscribe broker where devices can publish their data that are tagged with the devices'




| | Sync. | Emb. |
|---|---|---|
| q80% | 255 ms | 112 ms |
| q95% | 287 ms | 122 ms |
| q99% | 308 ms | 140 ms |

**Figure 4.12: Left:** Cumulative density function of the age of retrieved data when using the embedded HTTP and synchronization-based driver. **Right:** q-80, q-95, q-99 quantiles for the sensor data age.

physical location and name (tags use a URI-like structure, such as `FloorA/tmote4/energy`). The main role of the gateway is to offer a publicly accessible Web interface to an intranet where different sensor networks are connected (see Figure 4.13), and serve as a gate keeper where security and data access information for the different deployments could be implemented, and offers the data produced by the sensors with the desired granularity (e.g., authenticated administrators can read and write data to any device, whereas unregistered can only read the high-level energy consumption). For this prototype, a `GET` request on the public URI of the gateway `http://myWSN.com/energy` fetches the following JSON file containing energy-related data of the two floors in a building:

```
[{"name": "FloorA","currentWatts": 406.4, "KWh": 2.119,
   "time": "2001 JAN 01 06:57:05","status": "on", "maxWattage": 6192},
 {"name": "FloorB","currentWatts": 3506,"KWh": 1190,
   "time": "2001 JAN 01 06:57:05","status": "on","maxWattage": 4092}]
```

Finally, we have used an existing mashup called Energie Visible[2] to illustrate how very simple interactive Web applications can be built to interact with heterogeneous sensor networks directly from your browser [240]. Energie Visible is a Web mashup (written in AJAX using Google App) that fetches energy data from a gateway of the network of sensors and plots in real-time the energy consumption of the two floors of the building collected by the two different sensor networks.

[2]Online: `http://www.webofthings.com/energievisible`

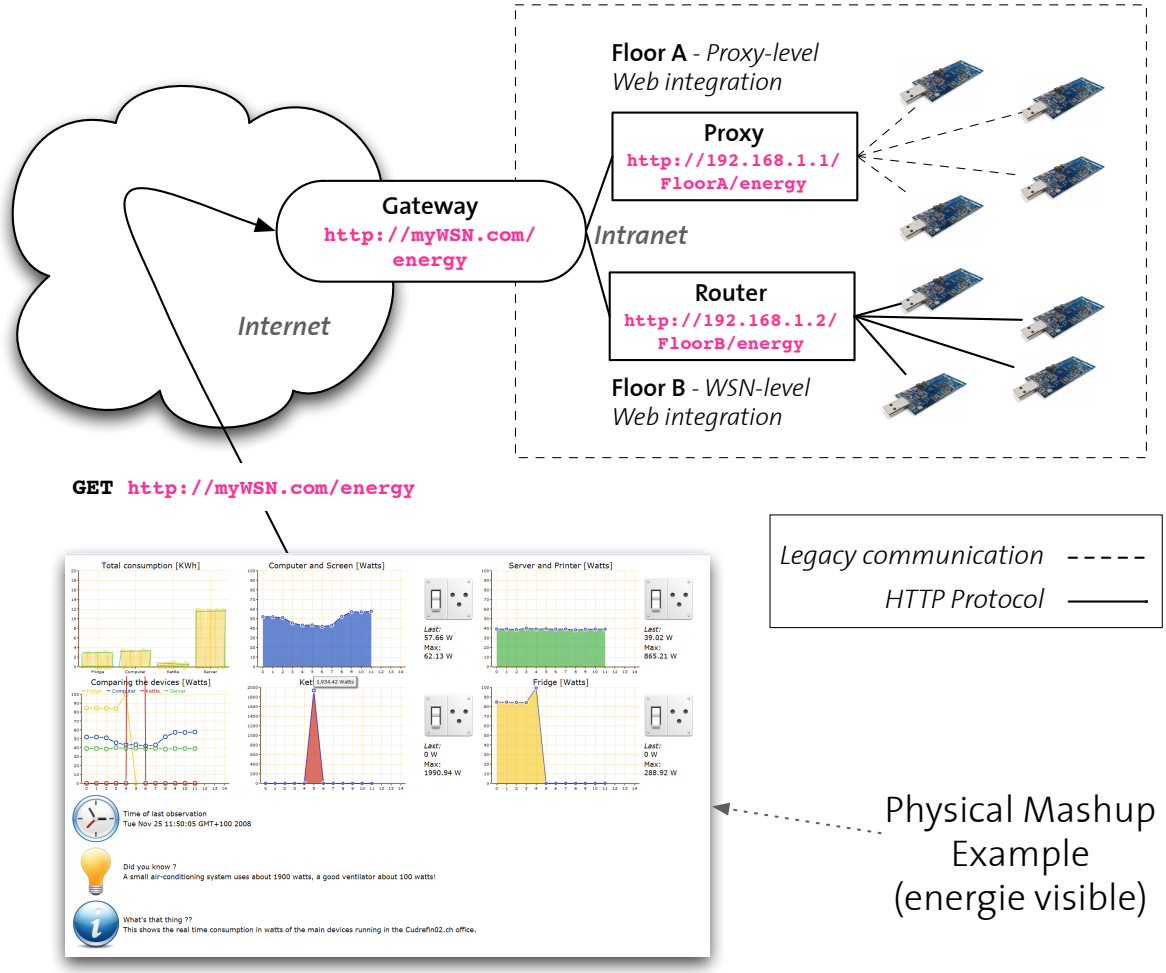



**Figure 4.13:** Complete system architecture of a multi-layered energy monitoring application. The first layer is a standard Web application (mashup), that fetches energy consumption data from a **gateway** using HTTP. The gateway offers machine-readable aggregated data from different sensor networks on the intranet through a **proxy** or **router**.

## 4.5 Discussion

In this chapter, two different paradigms for connecting devices to the Web of Things have been demonstrated and compared with respect to several different metrics. The evaluation of these architectures regarding the average data age, power consumption and concurrency shows that one can provide strong real-time guarantees on the data retrieved from a device by directly embedding an HTTP server but that this leads to disadvantages regarding scalability, response time and battery lifetime. An advantage of the synchronization-based architecture is its increased flexibility that allows to employ it within a wider range of applications, specifically in deployments where a higher data refresh period is tolerable and/or long battery lifetimes are desirable. However one has to keep in mind that this architecture introduces the need for a host software gateway that translates incoming HTTP requests to device-specific synchronization messages whereas the alternative is HTTP-compatible without compromise. Due to the decou-

pling of the communication between the HTTP client and the host on the one side and the synchronization between the host and the device on the other, synchronization-based drivers allow for considerably lower data-access times and thus faster servicing of HTTP clients. Besides, the distinction between embedded and synchronization server could be blurred by using the cache control feature of HTTP. Clients can specify the staleness they are willing to tolerate, and depending on the data age of individual sensor readings, the proxy can serve directly data from the cache or trigger updated values from the device.

# CHAPTER 5

## Distributed Infrastructures for the Web of Things

*"In these days, a man who says a thing cannot be done is quite apt to be interrupted by some idiot doing it."*

Elbert Green Hubbard (1865-1915)

The previous chapter has explored the usage of gateways and proxies to enable Web-based interactions with NEDs. We have further described how gateways can add functionality and improve the performance of distributed sensing applications. However, as the size of applications and networks grow, a large-scale infrastructure (backbone) for connecting heterogeneous NEDs and applications will be necessary. This infrastructure will also need to offer services useful for the Web of Things such as ad-hoc search, discovery, and interaction with physically distributed NEDs. Finally, to encourage reuse and flexibility a modular software architecture will be required to allow users to integrate their devices in this common infrastructure easily.

To move beyond *Intranets* of Things towards a unique and unified *Web* of Things, we propose a solution for binding disparate gateways, routers, and other components together to form an application-level distributed infrastructure for NEDs. Using a hierarchical tree mapped to physical locations, we can create a *location tree* that enables to easily implement location-aware applications for the Web of Things that build upon the existing network infrastructure. As our solution fully integrates with the Web, users can use URIs as a flexible context-aware search method to find and use things in specific location. Parts of this chapter have been reproduced from our early work published in [229, 226, 228], however, we detail and propose further refinements over our previous work here.

## 5.1 Structuring the Web of Things

On the Web, the location of data is irrelevant, since a powerful mechanism (URIs) is in place for accessing data regardless of where it is stored. In contrast, physical objects are always somewhere, and their location or state can change quickly. In addition, people also often change location (home, office, car, etc.), and to fully leverage the physical nature of objects and people their current location must be known. Search engines made it possible to index and search the static content of the whole Web which contributed largely to its success. However, the use of centralized repositories to keep track of the location and state of billions of NEDs would not scale. As access to timely information becomes indispensable in many disciplines, tools to search and filter information about the physical world in real time are needed. Such tools would be particularly useful in a large-scale Web of Things, where physical objects must be be searchable in real-time [205], in other words one needs a "Google for the real-world".

Localization of objects and people has always been a tedious technical challenge, and only recently GPS receivers have become a commodity allowing objects to be localized more easily in outdoor environments. However, for indoor applications GPS are unusable, thus indoor localization system based on Wi-Fi fingerprinting [168, 86] have become popular as they require no hardware infrastructure to be installed other than a Wi-Fi network. With a localization accuracy of a few meters, this technique allows room-level localization, which is sufficient for most pervasive applications, as shown in ubiquitous computing surveys [67]. Although indoor localization techniques improve over time, adoption of location-aware applications is hindered by the lack of robust and open standards for modeling and representing locations on the Web beyond geographical coordinates [245]. Due to the lack of support for modeling the physical location of things, discovering devices present in a location and interacting with them in an ad-hoc manner is a complex problem that requires customized applications. While solutions such as Bluetooth, Apple's Bonjour or Universal Plug and Play support discovery of devices on the same network and interacting with them, a common ground on which devices using different protocols could be discovered, searched, and interacted with globally and transparently is necessary.

As demonstrated by the success of the Web, loosely-coupled distributed applications are massively scalable. In this chapter, we explore how to leverage this characteristic to build a scalable location-aware infrastructure for the Web of Things. The general idea is to interconnect the gateways described in the pervious chapter and their associated devices, in order to form a large distributed infrastructure for interconnecting seamlessly all kinds of heterogenous devices. Using the Web as middleware enables to search and use devices depending on their current state, location or overall context on a global scale and in real time regardless of the actual protocols used natively by the devices.

### 5.1.1 Hierarchical Location Modeling

Although no formalized standard for modeling indoor locations prevails, one can find many location models in the literature [80, 93]. However, most of these location models have been designed to match the specific application needs of these projects. The NEXUS project [78] proposed an open platform for context-aware applications, with a location model that supports hierarchical naming schemes and different levels of detail for indoor and outdoor applications. In the AURA project [158], a hybrid location model and a formal representation that combines the advantages of symbolic and geometric location models is proposed. The clear separation of model and representation is what separates this approach from others. Consequently, the *AURA Location Identifier (ALI)* uses a formatted URI to represent both geometric and symbolic locations. Ubisworld [215] describes an interesting Web-based hierarchical model for locations, however it was not explicitly designed to be used on mobile Web devices. Other approaches for location modeling on the Web have been proposed to build a geographic Web that merges abstract information with geographical, such as KML [21] or GeoRSS [63]. Unfortunately, they were designed primarily for spatial (geometric) locations and are not suited for hierarchical models. Besides, their integration with the Web is limited, as they are not based on a RESTful architecture (see Section 3.2).

A central property of the Web is the use of hyperlinks to connect related resources on the Web, possibly using semantically annotated links, for example using the *friend of a friend* (FOAF) standard [55]. To create a distributed infrastructure for smart things, we propose to bind gateways together in a similar manner. In previous work, we have explored how gateways can be linked to realize a distributed location-aware infrastructure for devices [226]. By mapping each gateway to a unique location and linking gateways together according to their spatial disposition, one can model the relations between places in the real world. In practice, this requires each gateway to maintain a list of links (URI) to the gateways of (physically) adjacent places, and optionally to annotate semantically the nature or type of these links.

As illustrated in Figure 5.1, such a Web-based hierarchical model of places enables interaction with the real world with different levels of granularity (*country, region, city, street, building,*

*floor, room, object*). Thanks to the layered system constraint of the REST architecture, each *node* (i.e., gateway) in the tree offers a layer of abstraction to interact with the devices and gateways contained in its subtree, thus refines its parent by offering a finer granularity to clients of the infrastructure. By restricting component behavior such that nodes cannot *see* beyond the nodes in the layer above or below (nodes only know about their direct parents and children), we bound the overall system complexity and promote substrate independence, as layers can encapsulate legacy services or protocols.

Such location-aware gateways are also called *location proxies*, and both terms are used interchangeably throughout this chapter. We differentiate between two types of gateways: *virtual* and *physical*. Although identical from a software point of view, the difference lies in the fact that physical gateways (also called *terminal* gateways) must run on a computer (e.g., a wireless router, a PC, etc.) physically present in the area it maps to and can connect with devices in that location using short-range radio protocols such as Wi-Fi, ZigBee or Bluetooth. Terminal gateways can discover mobile devices in their surroundings and turn them – and their resources – into Web resources accessible over HTTP at runtime. Virtual gateways on the other hand, do not need to be installed at the location they map to, unless they need to connect directly to NEDs in that location. The virtual gateways in the tree shown in Figure 5.1 can be hosted anywhere in the world transparently as long as the logical structure of the tree is maintained.

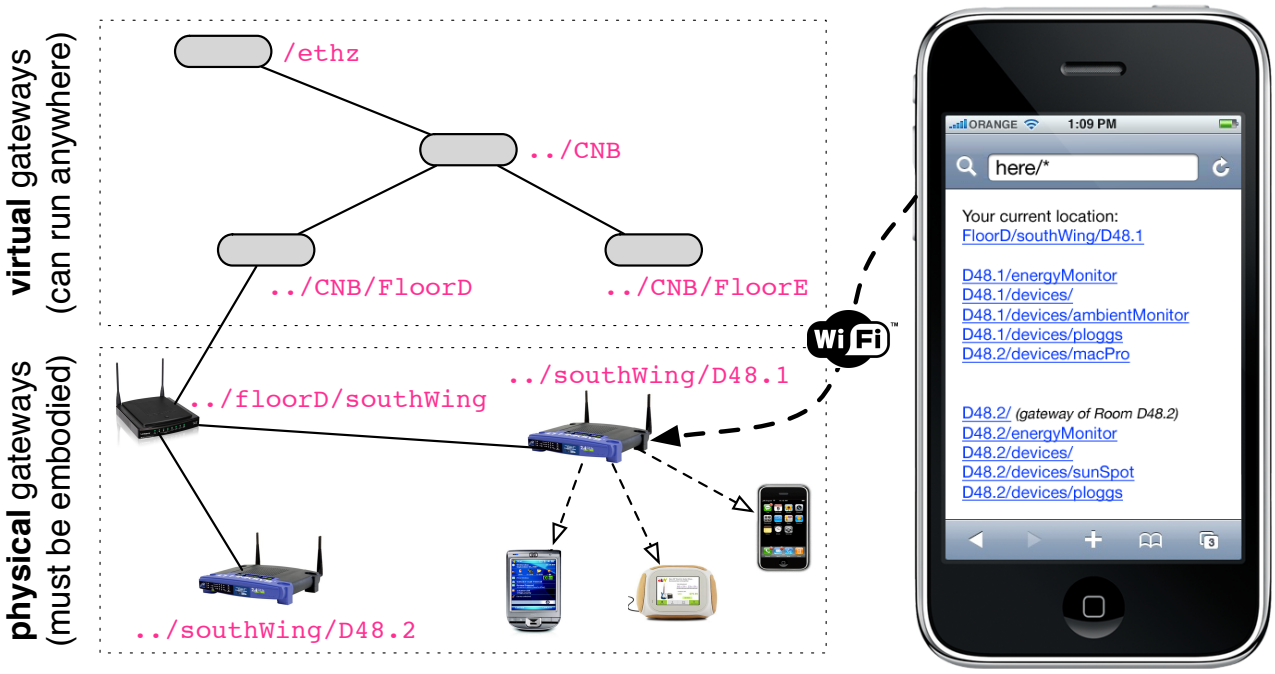



**Figure 5.1:** Example hierarchical tree of gateways from our building. The top gateway covers the gateways for each floor, and is composed only of virtual gateways. The `southWing` gateway runs on the router that bridges the local sub-network of that area, thus can access all terminal gateways running on computers physically located in each room. Terminal gateways have various physical interfaces to access mobile devices nearby.

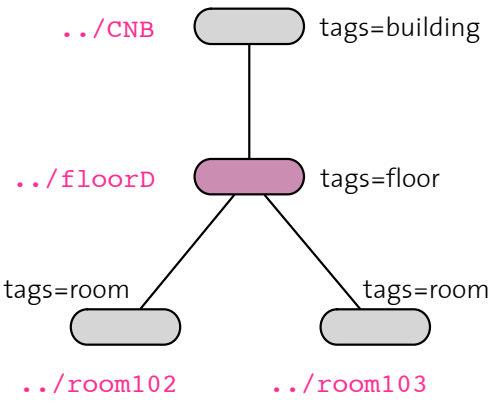
The mapping process that assigns the logical place name (*"room 44", "floor D", "east wing", etc.*) to gateways must be done once manually by an operator at setup time. Fortunately, since gateways are not mobile and the structure of their connections is rather static, little effort is required to maintain the tree structure once in place. The tree can be navigated by following links to surrounding gateways simply by clicking the links on a Web page or typing a URL in any Web browser.

On top of such an infrastructure, one can easily build a system that supports range and lookup queries for mobile devices that allow to restrict the scope of search to a fixed radius from a given point or to polygonal area [181]. Unlike most other hybrid models for spatial queries, our approach does not rely on a centralized database to store information about the system. Thanks to their RESTful interfaces, gateways are loosely-coupled components responsible for managing the devices (and other gateways) located in the area they are associated with. The higher up in the hierarchy, the less often things are likely to change, which forms naturally an efficient load-balancing system, as users only access the location proxies for the area of interest without soliciting the rest of the system for each request. The loose coupling between location proxies increases scalability, robustness, and flexibility of the infrastructure, which makes it particularly suited for ad-hoc interaction with/from mobile devices that move across locations.

### 5.1.2 Localization

Given that many different localization techniques exist for different applications, the representation of the location information must be kept agnostic of the localization technique used in order to maximize flexibility and interoperability. Although many formats to represent outdoor locations have been developed recently, no standard has been proposed to represent indoor locations using both symbolic and geometric models that is based on Web technologies. As geographic coordinates (longitude/latitude) are not practical for dealing with location concepts used in everyday life (for example a room number or a building name), a flexible model that supports user-generated symbolic annotations of places is needed. Sharing semantics of places can be a tedious problem in case a central authority has to maintain a repository of place names, besides it would conflict with the Web's decentralized nature.

To solve this problem, we propose to use the Web itself as a lookup service to find and explore locations, as well as to obtain information about places and the devices therein. Following the idea formulated in [158], we use URIs to represent locations and their containment relations as a logical path according to the URI definition. Consequently, RESTful URIs can be created dynamically by navigating the hierarchical tree formed by the gateways. For each URI, both machines and people should be able to retrieve a description of the identified resource. This is essential for a shared understanding about the location identified by the URI, where machines can retrieve semantically annotated data (e.g., using RDFa or Microformats) while people can retrieve a human readable representation (HTML).




```
<h1 class="current:name">floorD</h1>
<h2>Parents<h2>
<span class="parent" tags="building">
       <a href="../CNB">CNB</a>
</span>
<span class="children" tags="room">
       <a href="room102">room102</a>
</span>
<span class="children" tags="room">
       <a href="room103">room103</a>
</span>
```

**Figure 5.2: Left:** An simple location tree. **Right:** The semantically annotated HTML page of the "floorD" location proxy (HTTP `GET` at the URI `../CNB/floorD`).

Once the gateway hierarchy is in place, a simple mechanism to determine the current location of a mobile device on the tree is required. In other words, one needs a method to retrieve the URI of the location proxy associated to any specific location. We call this the *bootstrap problem*, and a simple method to associate the URI of a gateway to any location is necessary. One possibility would be to always connect automatically to the gateway with the highest signal strength. In practice this turns out to be very unstable as the signal strength is subject to significant and unpredictable fluctuations. Another solution would be to attach a bar code or an RFID tag into each room which acts as a physical bookmark that points to the URI of the location proxy associated with that room, but this requires to use an external application to read the tag. The actual spatial localization process is not part of this thesis and is not discussed further. We only assume an indoor localization system in place that can be leveraged to retrieve one's position with room-level accuracy. For example, RedPin [86] can be used as lookup service to retrieve the URI of the location proxy associated with the current location based on Wi-Fi fingerprinting.

In addition to using standard URIs, we introduce a concept called `here/*`. The idea is to use a fixed symbolic string (namely "`here`") in place of the host name in the URI to identify the current location proxy of the client accessing the URI. This is conceptually similar to a dynamic bookmark that is constantly updated so that it always points to the URI of the gateway associated to that physical location. Such a location-dependent URI can be constructed using the following syntax:

```
http://here/{location}[query]
```

The generic `here` hostname is automatically mapped to the URI of the gateway of the location the resource been accessed from (i.e., IP address or network name). To traverse the location structure, `location` is used to represent a hierarchical path of arbitrary length (for example `/building44/room3/`). Finally, by specifying an optional `query`, the user can search for devices

and services that match a specific expression. To instruct a gateway to return all the links to its sub-resources (devices or gateways directly connected to it), the wildcard character "*" can be appended to the URI. With this simple syntax, URIs become a flexible search bar to browse and search the physical world. For example, to retrieve all the devices tagged with the keyword phone located on the same floor, one can simply type the following URI in any browser:

```
http://here/floor/phone/*
```

This URI can be resolved by any location proxy, which will often be the Wi-Fi router the user is connected to. Such requests are routed to the appropriate proxy using either the name of the gateway explicitly (for example "floorD", which will be mapped to the closest – in number of hops – gateway that matches this name) or by leveraging the semantic annotations on links between gateways (using the generic tag "floor"), which is more flexible but also more ambiguous. The same URI will return different results depending on the node in the network that it is routed from. This allows to create fixed URIs that actually point to different resources depending on the geographic location where it is issued, which is a useful and flexible metaphor many location-aware Web applications could benefit from, as will be shown in the case study presented in Section 5.6.1.

## 5.2 InfraWoT - an Infrastructure for the Web of Things

A central requirement for the Web of Things is a meaningful structure on top of individual resources attached to the WoT. Because it matches the layered architecture of the Web [120], we opted for the hierarchical location model described above where each node is responsible for all the resources (devices, gateways, etc.) in its vicinity and the lower levels. When physical locations are mapped to URIs, networks of gateways form rooted trees, where the root represents the highest level of hierarchical location (for example the headquarters of an international organization). The hierarchical approach has been proposed in early research [229, 226] and shows benefits with respect to *load balancing* and *scalability* as users mostly access devices located in their surroundings and regarding the loose coupling between the infrastructure nodes. An important design choice was that every communication between proxies is local (i.e., forwarded across neighboring nodes in the tree). This helps to scale the infrastructure, as each gateway only needs to know about its direct neighbors and can ignore the rest of the hierarchy. Further efforts in designing and implementing such an infrastructure eventually led to the development of the *InfraWoT* system with Simon Mayer, which is proposed in [184].

### 5.2.1 Modular Software Architecture

A fundamental design choice is that every node in the tree must be a RESTful resource itself, regardless of its implementation. As long as every component comes with a single, uniform RESTful API, every node can be directly accessed both by other nodes using a machine readable format (JSON) or by humans users using a Web browser. This is a necessary condition for integrating the components of the infrastructure into the fabric of the Web. As flexibility is a key requirement for such an infrastructure, location proxies could be reconfigured on-the-fly without requiring a restart. To achieve this level of flexibility, we have chosen the OSGi framework [35] as its highly modular approach enables every component to be developed and upgraded individually at runtime. This ensures "hot-pluggability" with other software developed for the Web of Things which allows dynamic reconfiguration of the network and its applications.

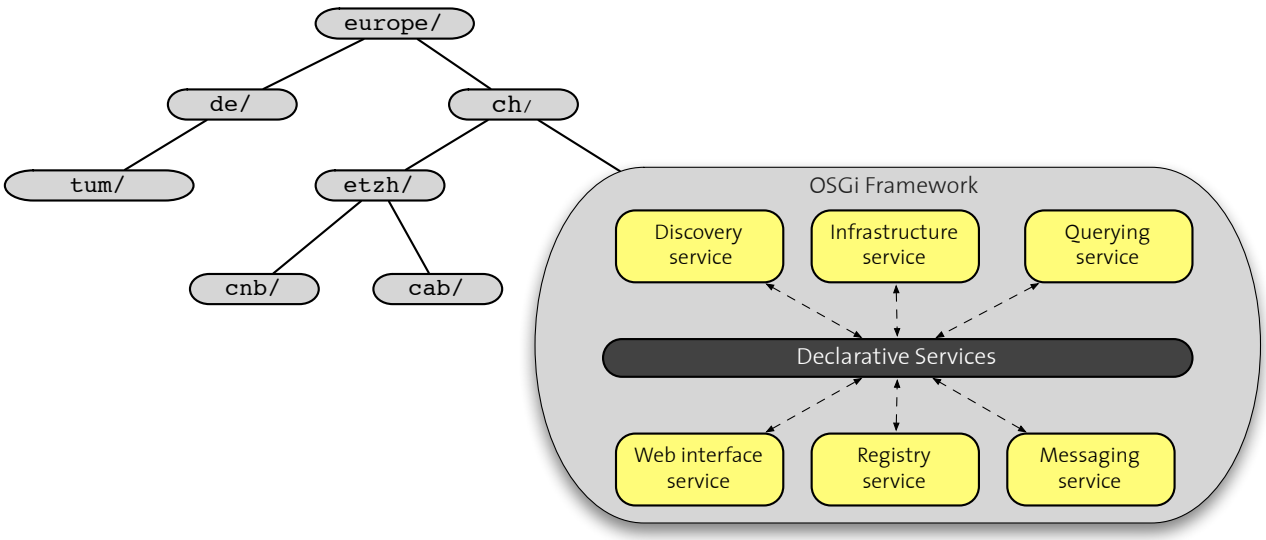



**Figure 5.3:** Modules of the gateway software powering each InfraWoT node. Modules can access each others' services using OSGi Declarative Services.

The software applications running on each node consists of several modules that can interact with each other via OSGi-based messages. The gateway software is a refinement of the gateway proposed in the previous chapter augmented with additional functions and modules. Each module is responsible for a specific task and implements an interface that gives access to a limited set of infrastructure-wide functions. Figure 5.3 presents an overview of the different modules in each InfraWoT node.

- **Infrastructure Service Module (ISM).** This module maintains the correct tree structure with respect to the hierarchical locations of other proxies within its scope. It takes care of registering parents and children and monitor the traffic between proxies (i.e., between parents and their children).

- **Discovery Service Module (DSM).** This component handles the discovery of resources, in particular the retrieval of information on resources that are to be integrated into the infrastructure and the mapping of this data to internal representations. Through this process, newly discovered resources get attached to the tree hierarchy via an InfraWoT node and thus can benefit from infrastructure-wide services.
- **Registry Service Module (RSM).** This component manages data about attached resources (both locally connected devices and neighboring gateways) and stores this information into an embedded database.
- **Messaging Service Module (MSM).** This module offers a transparent interface to set up a messaging (i.e., publish/subscribe, see Section 6.2) system between client applications, gateways and physical devices attached to the Web of Things.
- **Querying Service Module (QSM).** This module is responsible for handling incoming queries. It retrieves local resources that correspond to the query and forwards the query to suitable sub- or super-nodes.
- **Web Interface Module (WIM).** This module provides a Web interface that allows to access the various functions offered by the gateway, either via a RESTful API or via an HTML user interface accessible from any browser.

We describe briefly the Infrastructure and the Web interface services in the remainder of this section. The Discovery and Querying services are detailed in Sections 5.3 and 5.4. More technical details about InfraWoT are available in [184].

### Infrastructure Service

The *Infrastructure Service Module* (ISM) is used to initialize the tree structure at startup time and ensures that the correct structure is maintained during operation. In particular, this service allows the overall structure to recover from node failures and eventually re-establish the initial tree configuration (self-stabilization). After the initial setup, all gateways initialize their IS bundles which start the registration process with their assigned parents by sending an HTTP `POST` request that includes their own URI. Every gateway that receives such requests, forward the received URI to the *Discovery Service Module* (DSM) which will parse and analyze the information about the resource, as described in Section 5.3.

Furthermore, the ISM is responsible for the process of attaching new sub-resources (i.e., other proxies or devices) that are found by the DSM or registered manually. Any resource that is encountered and analyzed by the DSM is passed to the ISM which uses the resource's hierarchical location information to determine whether to attach it to the current proxy or forward the request to a more appropriate gateway. In the latter case, the infrastructure takes care

of routing that resource to the proxy whose hierarchical location corresponds best to the resource's, as shown in Figure 5.4. If a registering resource does not provide explicitly a location information within its Web representation, the DSM automatically assigns the location of the proxy itself.

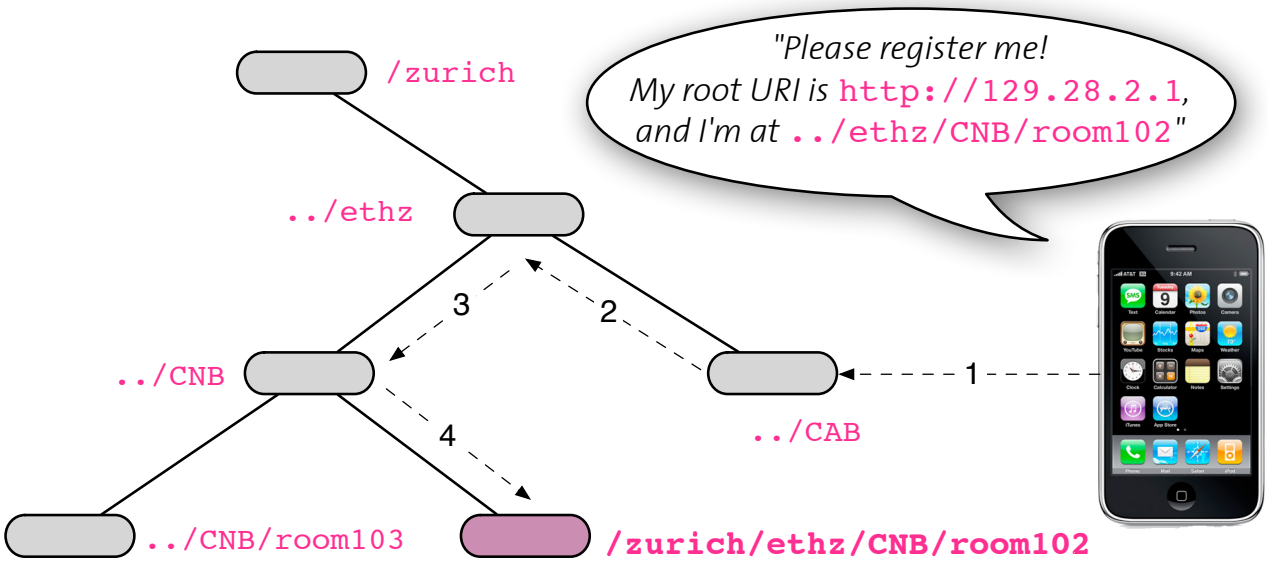



**Figure 5.4:** Infrastructure-assisted discovery. Any device can POST its root URI (where a semantically annotated description of the device properties can be found) to any node in InfraWoT. If an optional location is specified, the registration command is forwarded to the node corresponding to that location.

The ISM also serves as *garbage collector* by regularly contacting the sub-resources (or receiving heartbeats from them) and removing them from InfraWoT when they become unavailable. The Infrastructure Service starts two threads that regularly contact the parent node, all registered children nodes, and all attached resources. If the connection to any of these resources is lost, the corresponding entity gets black-listed and will be removed if contact cannot be re-established after a timeout period.

The *Web Interface Module (WIM)* enables to access the infrastructure and the various resources connected to InfraWoT using only RESTful requests. In particular, the Web Interface Service enables the RESTful configuration of InfraWoT location proxies. The Web server is built upon Restlet [43] and offers various device- and gateway-specific functions. The *root page* of any gateway (the URI of the gateway, abbreviated as "/" for clarity) provides general information on the current proxy (name, hierarchical location, connected sub-nodes, attached resources, etc.). From the root, one can access four different sub-resources (in addition to `/query` described in Section 5.4):

- `/locations` is the list of all attached location proxies. Child nodes send HTTP `POST` requests to this address to be registered by the proxy. The HTML representation of this

resource can be used to navigate (browse) the infrastructure. The individual gateways registered to any node are represented as child resources of the `/locations` resource and are mainly used to configure child nodes. For instance, to remove the gateway called *floorD*, an HTTP `DELETE` request should be sent to the resource `/locations/floorD`.

- `/resources` resource represents a list of all sub-resources attached to the current gateway. Similarly to the `/locations` resource, Web of Things resources may send HTTP `POST` requests to this resource to be registered by the gateway. Likewise, these resources are represented as children of the `/resources` resource and can be interacted with via requests to their respective endpoints within the local gateway.

- `/infrastructure` is mainly used internally by the InfraWoT software to send and receive maintenance information. One of its sub-resources plays an important role in the Web-based configuration system of InfraWoT that enables clients to configure a proxy by sending HTTP `PUT` requests to its configuration interface at `/infrastructure/configuration`. When a client `PUT`s a string of data to this endpoint, the proxy relays that data to the Infrastructure Service to retrieve the resource encoded in the transmission and applies that information to its own representation. Although the currently preferred way to configure a gateway is to `PUT` the desired configuration as a JSON-encoded resource, a gateway can be configured using any representation that is supported by the Infrastructure Service (using the strategy pattern as is done in the Discovery Service).

- `/wms` resource and its sub-resources handle all the interactions related to the InfraWoT Messaging Service, that is the creation, updating, and deletion of queues and all the messages received or transmitted by the gateways, client applications or devices (see Section 6.2 for more about messaging).

## 5.3 Ad-hoc Device and Service Discovery

When a new device is connected to a network, an automated mechanism to detect the device and to extract information about it is necessary. For example, Universal Plug and Play (UPnP) [58] and Zeroconf/Bonjour [7] support convenient network-level device discovery mechanisms and service interaction primitives. Although, they are widely used, their usage is limited to internal networks (LANs) and are not interfaced with HTTP. UDDI [57] and similar Web service repositories run on top of HTTP and are usable on the global Internet network, but are heavyweight, overly complex, and in conflict with REST. Many other device and service discovery standards and protocols are available, each specialized for a single or a few particular problems, but are not appropriate as a general solution for RESTful discovery.

HTTP does not define any discovery mechanism (on the Web, resources are discovered by following URIs). However, an HTTP-based automated discovery method is necessary to support

ad-hoc interaction and integration of heterogeneous NEDs into the Web of Things. Although, various methods for describing RESTful services have been proposed [147, 134], none of those focus on the requirements of NEDs as autonomous and mobile entities.

In the next section, we propose a simple two-step RESTful discovery procedure for connecting automatically Web resources (in particular Web-enabled NEDs) to the InfraWoT system that can be implemented with minimal infrastructure changes. This simple procedure can be used as a basis for a searchable, large-scale Infrastructure for the Web of Things. The advantage of the procedure we propose is that devices do not need to implement any specific discovery protocol, only provide semantic information about themselves in their root document using semantic annotations or an external document/format that can be understood by the gateway as shown in the next section.

### 5.3.1 Step I: Network-level Device Discovery

The first step of the discovery process is in charge of monitoring any new WoT devices that are being connected to a network. For simplicity reasons, we only consider Ethernet/Wi-Fi-enabled devices, as for other physical interfaces a gateway is necessary.

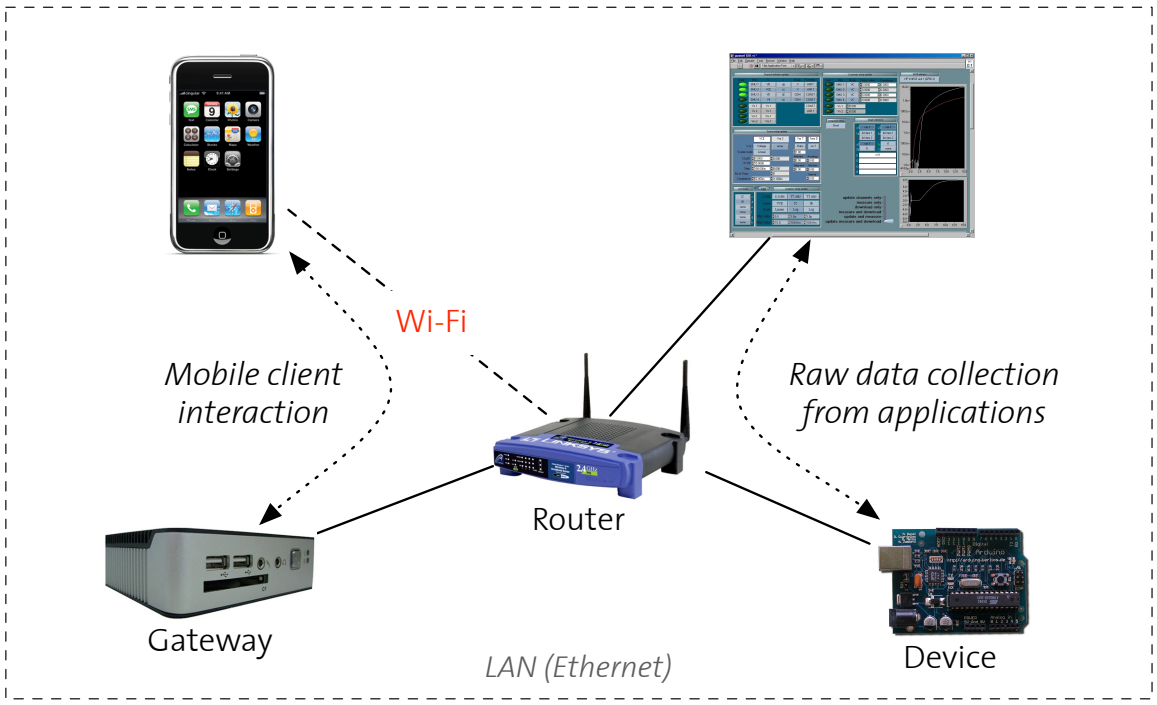



**Figure 5.5:** Setup of the experimental network.

Most existing discovery solutions rely on devices multicasting UDP messages over the network [126]. However, as such messages are not part of HTTP, they can often be blocked by firewalls. SoaM [234] is an HTTP-based architecture for semantic devices, however, it was not

based on a RESTful architecture, therefore its integration with the Web is limited. We therefore propose a REST-based protocol to perform network device discovery. We assume that in each network, the router is always knowledgeable of the connected network devices (usually a table of automatically assigned IP addresses), and as such can provide all required discovery information. To access this information, we used an ASUS WL-500G Premium router on which we installed *OpenWrt*[1], which is a widely used open source Linux distribution available as firmware for many modern network routers. Its user interface, *LuCi*[2], exposes some of its libraries and functions to external applications through a JSON-RPC API.

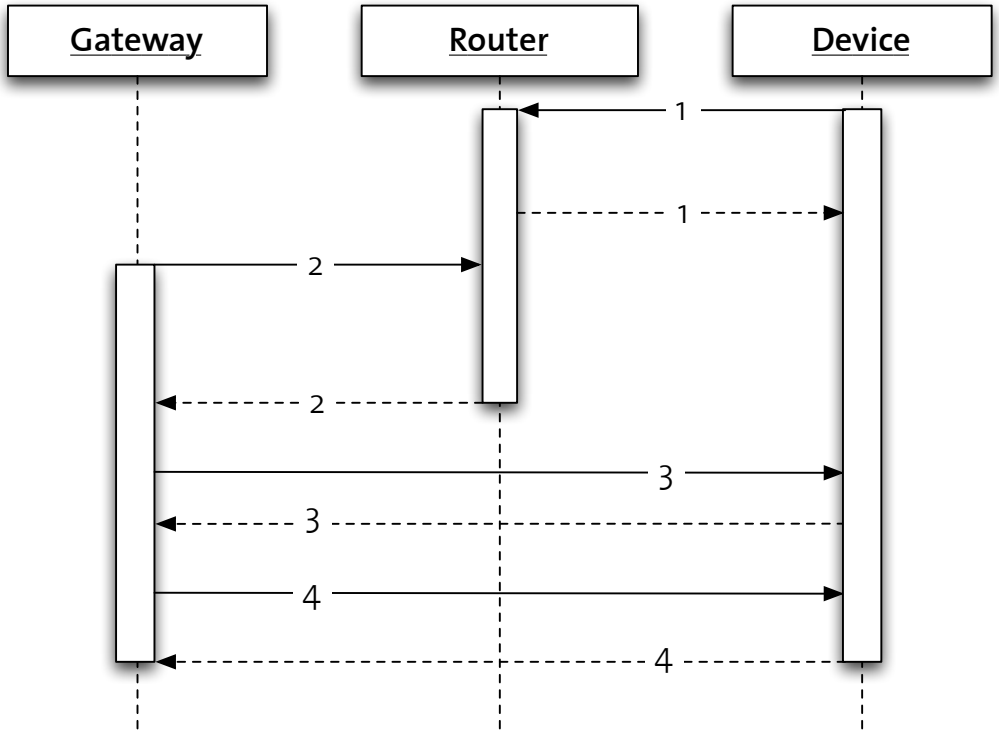



**Figure 5.6:** Sequence diagram of the RESTful discovery process of devices. 1. Device connects to LAN/Wi-Fi and gets an IP address from the router using DHCP. 2. The gateway monitors the router's DHCP table. 3. For each new device found, the gateway retrieves the root device page (by default a HTTP server running on port 80) and parses it to find information about the device. 4. The gateway retrieves the semantic description of the device.

To retrieve the list of all connected devices, the following HTTP request[3] is sent to the router:

```
POST http://router/cgi-bin/luci/rpc/sys?auth=EBAE1814FA625E73CA0514004428D64A
Content-type: application/json
Content: {"jsonrpc": "2.0", "method": "net.arptable", "id": 1}
```

[1]Online: http://www.openwrt.org
[2]Online: http://luci.subsignal.org
[3]Note that this is a typical RPC and not RESTful use of HTTP, as a `GET` should be used (read-only operation), and the resource id ("method") should be part of the URI, not encoded in the message payload. The current version of the LuCi is not RESTful, but as it is an open source project, the RESTful equivalent of this procedure can be easily implemented.

This request will execute the RPC method `net.arptable` on the router, which will return a list of all the devices connected to the router encoded using JSON:


```
{"id":1,"jsonrpc":"2.0","result":[
  {"Flags":"0x2","HW type":"0x1","Device":"br-lan","Mask":"*","HW address":"00:
     E0:4C:45:57:EF","IP address":"192.168.1.114"},
  {"Flags":"0x2","HW type":"0x1","Device":"br-lan","Mask":"*","HW address":"00:1
     C:B3:25:F6:9B","IP address":"192.168.1.149"},
  {"Flags":"0x2","HW type":"0x1","Device":"eth0.1","Mask":"*","HW address":"00:0
     D:66:22:38:01","IP address":"89.211.57.1"}]}
```


The response includes a list of the IP addresses allocated by the DHCP server on the router. Once this list is retrieved by a gateway, the root page of all newly allocated IP addresses (by default on port 80) is parsed by the Discovery Service to find a WoT-compliant semantic description of the devices and its services using the procedure described in the next section.

This is certainly an inefficient procedure, and a refined version should enable the gateway to subscribe to changes in the DHCP allocation so that every time a new device connects, the router will trigger a callback procedure described in the next section, which would eliminate the need to poll the DHCP table manually.

### 5.3.2 Step II: Resource Discovery

Once a new device has been connected to the network and found, the second discovery step (resource discovery) is carried out to retrieve various information about the device (metadata about functions/services, description, etc.) and make this information available within Infra-WoT. In case it cannot be triggered automatically by the device discovery process described in the previous section (for example when the IP allocation table is not available on the router via a Web API), one needs to manually `POST` the URI of the device root page to the `/resources` endpoint of a gateway. Such requests are processed by the Web Interface module, which forwards the root URI of the device to the Discovery Service module. Once the Discovery Resource module is triggered with an URI, it retrieves the device root page and searches for semantic about the device in the root HTML page.

Microformats[4] are a set of lightweight open standards that leverage XHTML to *"make it easier to publish, index, and extract semi-structured information such as tags, calendar entries, contact information, and reviews on the Web"* [165]. Microformats were designed to enable and encourage the sharing, distribution, syndication, and aggregation of various content on the Web using a simple, human-centric format. Similarly to RDFa [68], Microformats can be directly embedded in the HTML code of the page in such a way that it does not change the rendering of

[4]Online: http://microformats.org/2010/07/08/microformats-org-at-5-hcards-rich-snippets

the visual elements. However, an application can parse exactly the same HTML code to extract and "understand" these annotations. In this section, we propose to extend the microformat metaphor to publish information about devices, by simply embedding a machine-readable minimal description about the device (metadata). Likewise, we can embed a machine-readable description of the RESTful resources on the device, in particular which verbs and parameters each resource accepts and returns. For example, the semantically annotated HTML description of the light sensor resource of a NED is shown in Listing 5.1 (a more complete description of the NED is given in Listing A.1). It is worth noting that most of this information could be inferred by crawling the HTML representation of a device by following the links to all the resources and using the HTTP `OPTIONS` method to retrieve the verbs supported by each of them. Embedding directly this information in the HTML representation of a device presents some advantages such as minimizing the HTTP calls on the device (similarly to retrieve a WADL description) or being able to generate user interfaces on the fly based on the parameters accepted, their type, and range among others.

```
<span class="hrests">
  <span class="service">
    <span class="operation">
      The
      <span class="label">Light Value</span>
      operation returning the
      <span class="output">current light value</span>
      can be invoked using a
      <span class="method">GET</span>
      at
      <span class="address">../{device}/sensors/light</span>
    </span>
  </span>
</span>
```

**Listing 5.1:** Microformats annotations used to describe a device and its operations, in this case a photovoltaic sensor of a sensor node.

In some cases, a more structured and rigid format might be required to fully describe the API of a device, for example WADL [147]. For this, we use a technique called *feed autodiscovery* which was popularized by blogs to reference machine-processable resources associated with a particular web page (e.g., its RSS or ATOM feeds). This technique has been recently standardized as the link types in the HTML 5 specification[5]. This is done by using the `<link>` tag in the header of the root page of a device (`http://device-ip/`), which informs the spider that the resource `http://device-ip/about` contains a meta-description of the device and its services, as follows:

```
<html>
```

[5]See Section 4.12.4 in http://www.whatwg.org/specs/web-apps/current-work/

```
<head>
...
<link rel="meta" type="application/rdf+xml" title="about" href="about" />
<link rel="meta" type="application/text" title="about" href="about" />
...
</head>
```

This advertises that the meta description is available both in text (HTML documentation with microformat annotation) or RDF (whose format/namespace will be specified in the RDF document). In order to minimize the coupling for the discovery procedure, we do not enforce any particular standard to encode this information (many formats are available). Instead, we propose to use the *strategy* pattern (see [128]) to allow users to extend the discovery procedure by adding their own formats to describe the device and implement an associated *discovery strategy* to parse it.

When InfraWoT parses the header above, it will iterate over each `<link>` element to find the first one which it can understand (similarly to when a browser tries to find an appropriate external application that can open a document type not supported by the browser). Once a matching strategy to parse one of those documents is found, it will launch a specific handler to parse the document, retrieve the metadata about the device, create a virtual entity (representation) and attach it to the gateway.

In the current version of InfraWoT, two strategies have been implemented. The first one is the default strategy where InfraWoT searches the HTML resource representation found at the device URI for Microformats as described above. Several Microformats can be used, each for a particular domain; a *geo* and adr Microformat for describing places or an *hProduct* and *hReview* microformat for describing products and what people think about them. The default InfraWoT understands a compound of several (optional) Microformats that can be used to better describe devices. This helps for devices to be searched by humans using traditional or dedicated search engines[6], but it also helps them being "discovered" and understood by InfraWoT in order to automatically index and use them. Currently, InfraWoT supports five Microformats; *hProduct* is used to describe the device itself (brand, name, picture, etc.). *hReview* reflects the quality of service or experience users or applications had with the device, *hCard*, *Geo*, and *adr* specify the location context of the device (address, region, country, latitude, longitude, etc.)

The second type of discovery strategy that is currently supported by InfraWoT is based on interpreting the resource representation as a JSON object according to a pre-defined, fixed, schema. While this is not realistic on a Web scale, it can be used in controlled environments (e.g., in an intranet or behind proxies such as gateways) as it is much more efficient than the Microformats-based discovery because there is no need to parse the entire device root page to find the embedded semantic annotations.

[6]Google, Yahoo, or Bing are building a common HTML markup schema, see: http://schema.org/

The semantic description and discovery is not the topic of this thesis, but it is one of the core motivation for the Web of Things. It is a powerful method for introducing programmable semantics into the Web, and for this reason has been briefly described in this section. A more thorough investigation on this topic have been done by Dominique Guinard and Simon Mayer, and further information can be found in [184, 185, 136].

## 5.4 Querying Service

Querying for resources within the scope of specific locations (such as *"find all printers in this room"*) is a central feature of any infrastructure for smart devices. InfraWoT enables such queries using various parameters such as the name of resources, their description, or the RESTful operations and parameters they accept. Additionally, InfraWoT defines several query types that encapsulate scoping information (i.e., *where* to search for resources). The handling of a search request is thus a two-step procedure that consists of routing a query to the most appropriate gateway first (e.g., the location proxy responsible for a specific building or a certain room) and then execute it there and return the discovered resources.

A client can submit a query by sending an HTTP `GET` request to the `/query` endpoint of a proxy that contains a description of the query either as a JSON object or using a collection of form parameters. Each query may contain various parameters such as a device ID, tags, or the URI of the initial gateway that issued the query. HTTP responses to client queries can be delivered in multiple formats, depending on the HTTP `Accept` header specified in the request (usually JSON/XML in queries from another node/application, HTML for queries given as a URI in the browser).

To illustrate this principle, we show in Figure 5.7 a typical InfraWoT search query. The mechanism used is inspired and leverages the practical implementation of standard HTTP request, which heavily relies on intermediate proxy nodes and caches. This way, not everything has to be retrieved from the original node each time, and the actual communications remains transparent. In particular, we suggest to use the URI forwarding mechanism, which uses the `3XX` range for error codes, in particular the `302 Found` status code used to indicate a temporary redirection. Browsers follow these redirections automatically and also update the address bar with the new URI. The example shown in Figure 5.7 is based on the following (minimal) HTTP interactions:

```
1 # Initial request to CAB gateway
2 GET /query
3 Host: [CAB]
4 Content: location=../ethz/CNB&keyword=meter
5
6 HTTP/1.1 302 FOUND
```

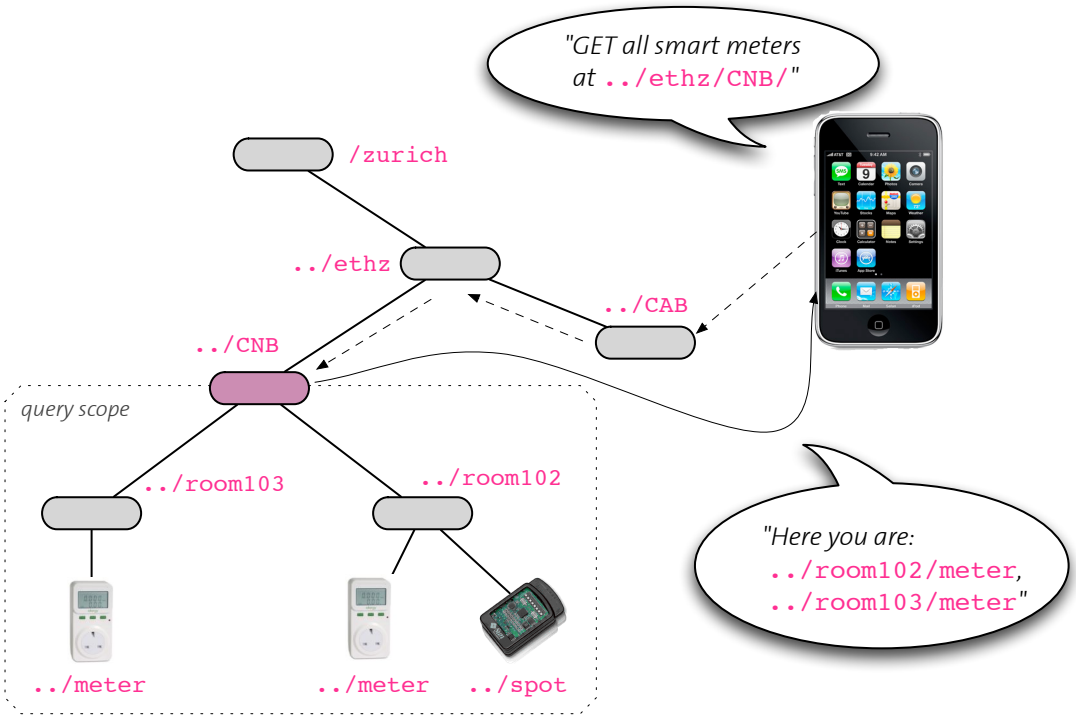



**Figure 5.7:** A client at location ../zurich/ethz/CAB wants to issue a search query at location ../zurich/ethz/CNB. When the request is issued at any node in InfraWOT, the nodes will first route the query to the appropriate location where the query shall be executed. The mechanism used to forward a query is the same as the original query, but by proving a target location as a parameter and keeping the original endpoint where the answer should be transmitted, one ensures the response can be sent back to the original query issuer.

```
7  Location: [CNB]
8
9  # Request is posted again to CNB gateway
10 GET /query
11 Host: [CNB]
12 Content: location=../ethz/CNB&keyword=meter
13
14 HTTP/1.1 200 OK
15 Content:{"results":["../room102/meter","../room103/meter"]}
```

This is a two-step procedure, where the client (mobile phone browser) must follow the redirect information, and post the same request to the resulting gateway, which is usually done automatically (therefore only one request is issued explicitly by the clients).

In principle, proxies should enable querying for all parameters that occur in the internal representation of resources. *Structured* queries (i.e., classical database queries) are quite complex for humans, who would rather provide textual information about the object in demand, and let the querying mechanism carry out the interpretation of this data. Our implementation simply

uses key-value parameter pairs that can commonly occur in InfraWoT. There are several types of query that are supported in InfraWoT (note that they can be combined as all of them can be encoded using key-value pairs):

- **Keyword Queries.** This is the most general type of queries that are specified using key-value pairs as parameters that are matched. For example, this type of query is used to search for keywords, tags, device names, or UUID (machine-friendly resource IDs, for example EPC codes).

- **Spatial Queries.** This type of queries enable to specify the *scope* (location) where a query should be carried out. This can be done explicitly (either using a parameter to specify a symbolic or geometric location or by appending it to the resource URI) or implicitly (if no scope is given, the location proxy which processes the request is assumed to be the query scope by default).

- **REST Service Queries.** This type of queries allows to specify the application-level (REST) properties supported by resources (e.g., the input or output format of a resource, or the HTTP verb it accepts).

Support for complex and expressive queries to search for specific devices is an essential feature for the Web of Things, and in addition to the discovery service, is another core service that will be needed. Nevertheless, querying for the Web of Things is not the core focus of this thesis, thus has been only marginally investigated in Sections 6.3.2 and 6.4. More information about querying and search are to be found in Guinard's work [143, 140, 136].

## 5.5 Web-based Ad-hoc Interaction

An essential feature of an infrastructure for interconnecting smart objects is to enable ad-hoc interaction with the infrastructure with minimal knowledge. As mobile Web browsers have become common, we suggest to use the Web to enable such interactions.

Recently, frameworks such as jQtouch [19] and Sencha [48] have become a popular alternative to native code for developing mobile applications. Mobile Web applications based solely on (X)HTML, Javascript, and CSS are faster and simpler to develop, run on multiple devices without changing the application code, and benefit from direct integration with the Web. Besides, as Javascript frameworks for mobile applications keep improving, the touch and feel of these applications and use of mobile touch interactions are becoming comparable to those of a native application. In combination with Phonegap [38], which allows to turn a Web application into a native one, this development method lowers the access barrier to develop mobile applications rapidly.

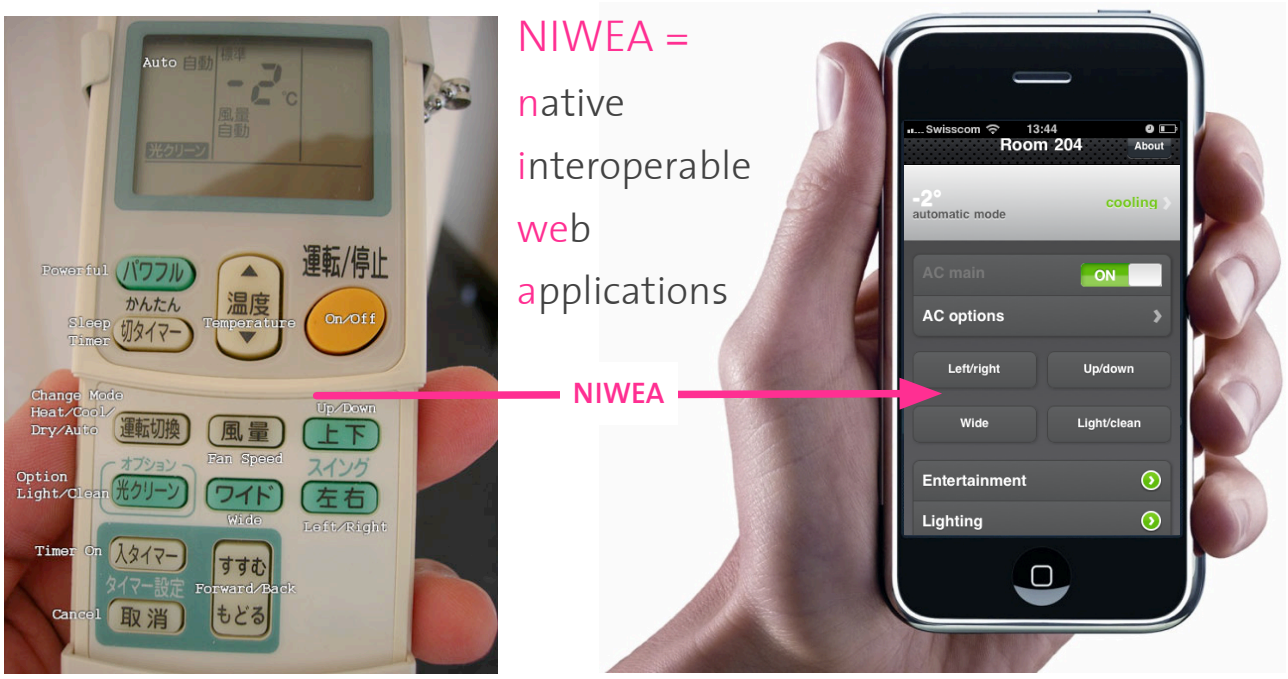



**Figure 5.8: NIWEA.** A standard mobile Web application accessed with a smart phone is a more flexible user interface to interact with pervasive services present in any location.

Hannes Gassert coined the term NIWEA (Native Interoperable WEb Applications) to refer to these hybrid native applications[7], and we will use this name throughout this thesis as well. NIWEA is an excellent paradigm for the Web of Things, as it enables fast prototyping of user interfaces that integrate natively with the Web of Things. The Microformats embedded in device pages can be easily parsed using Javascript to automatically generate user interfaces to control devices based on the semantic descriptions of the services and their controls. As an example, in Figure 5.8 a mobile user into an unknown hotel room in Japan can connect to the Wi-Fi network of the hotel, and log in to the Web application of his room. A simple Web interface is then generated that allows him to control the various Web-enabled devices in his room, from the entertainment, to the lightning, to the heating and air conditioning systems.

## 5.6 Location-aware Physical Mashups

Once devices are part of the Web, one can easily develop a new type of physico-digital applications that merge resources from the Web and from the real-world seamlessly [139]. In this section, we describe a prototype implemented on top of the patterns introduced earlier.

### 5.6.1 Case Study: Mobile Ambient Meter

[7] See: http://liip.to/niwea

As proof of concept, we illustrate how simple location-aware applications can be built by using a Web-based infrastructure. Our system is composed of a mobile energy meter that displays the energy consumed in the room it is located in.

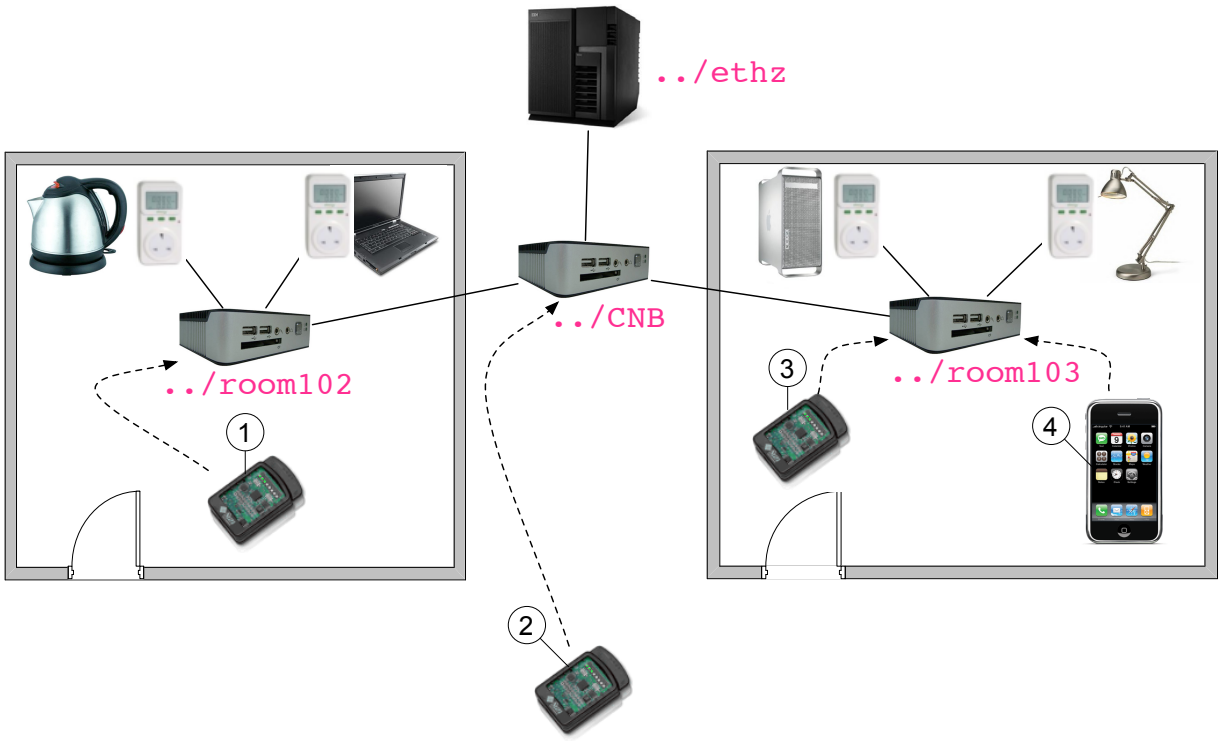



**Figure 5.9:** The prototype deployment on the floor of a building, with two physical gateways in two different rooms and a third one in the hallway. Two energy meters are connected to the gateways in the rooms. As and Ambient Energy Display moves from one room to the other, its color changes according to the level of energy consumption in the current location (① - ③). Mobile users can also interact directly with the infrastructure and pervasive services in their current location using a mobile phone (④).

As shown on Figure 5.9, two gateways are deployed in rooms 102 (`../room102`) and 103 (`../room103`), and communicate wirelessly with the *Ambient Energy Display* (AED). The AED is simply a Sun SPOT sensor node with an embedded Web server to enable HTTP-based access to its various functionalities, for example by entering their URI in any Web browser. The energy consumption of electric appliances are monitored using Ploggs[8], which are sensor nodes that combine an electricity meter and a data logger (white boxes next to the laptop and the kettle in Figure 5.9). Ploggs connect to the gateway in the room using Bluetooth which turns them into URI-identified resources that can be fully controlled using HTTP.

The AED meter is located in room 102 and is connected with the gateway of that room ① (we assume that only one gateway is within signal reach in each room). The AED gets the energy consumption of the electric appliances in that room from the gateway that aggregates

[8]Online: `http://www.plogginternational.com`.

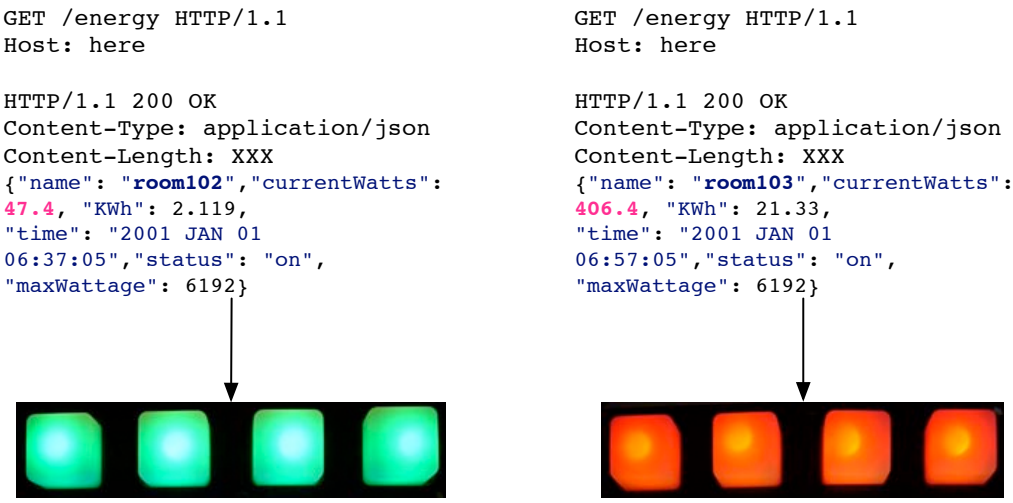



**Figure 5.10:** The AED retrieves periodically the same URI (here/energy), which gets routed automatically to the gateway of the location the AED is in. Depending on the current consumption in that room an appropriate color pattern is displayed on the Sun SPOT.

the individual consumption of all Ploggs in the room. Depending on the total amount of energy consumed, the Ambient Meter changes its color from green (little energy is consumed) to red (a lot of energy is consumed), as shown in Figure 5.10. The meter retrieves this information by periodically issuing a HTTP `GET` request on the here/energy resource. Since the gateway knows its location, it automatically resolves this URI to: http://192.168.99.6:8081/energy, which refers to a resource that aggregates dynamically the energy consumption collected by all the Ploggs present in that room. The meter is then moved into another room. On its way, the meter connects to the gateway of the CNB building, so that it displays the energy consumed in the area ②, which is the aggregation of the individual consumption of each room. The AED is then taken to room 103, where it reconnects automatically to the gateway of that room ③. The consumption of a lamp and a desktop computer located in room 103 is then displayed on the AED. Note that the AED does not actually deal with any explicit location since it only asserts the energy consumption its current location – which depends on the closest gateway and is always accessible at the URI here/ as explained in Section 5.1.2.

In the last part of this scenario, a user enters room 103 ④. The user connects to the gateway of that room (../room102) using his mobile phone, and gets back a Web page containing links to the resources in the current room, but also to neighboring locations (e.g., room 102, CNB, etc.). By clicking on the room103/meter he retrieves the amount of energy consumed by the lamp and the desktop computer. As the AED is also currently located in this room the user can click on its URI (room103/spot) to access its services, for example the temperature currently sensed by the AED in room 102 or 103. This illustrates how users can leverage the gateways' structure and the concept of Web of Things to browse (and bookmark) their physical location as they would with other Web pages.

## 5.7 Discussion

In this chapter, we have described how the gateway software proposed in Chapter 4 can be extended to allow several gateways to be interconnected and form a large distributed infrastructure for NEDs. This network of gateways forms a flexible backbone for the Web of Things that can overlay on top of the current Web infrastructure with minimal changes. By structuring the connections between gateways, one can create a hierarchical tree that can be mapped to physical locations (for example symbolic place concepts in a building such as floors, rooms, etc). When new devices connect to this network through a gateway, they inherit automatically the location of the gateway that gateway.

Using RESTful patterns to interact with gateways, flexible and self-stabilizing models of the spatial hierarchy between places and things can be integrated into the Web fabric in a natural and efficient manner. Thanks to the layered system offered by REST, which bounds the overall system complexity and promotes the loose coupling between components, different parts of the network can be implemented independently according to the specific requirements of different applications. On top of this hierarchical place model, we illustrate how Web clients can use the HTTP/URI mechanism as a lightweight and simple, yet powerful, flexible, and expressive combo to perform context-aware searches to find and use relevant objects at specific locations in real-time. Various scenarios are made possible by enabling RESTful access to this infrastructure and its sensors, for example searching for restaurants in the vicinity according to their real-time situation (crowded, noise, etc.). By turning the local network infrastructure into an ad-hoc query service, one does not require centralized Web sites to mediate all the information anymore.

By describing how various functions useful for building more interactive pervasive applications can be implemented using REST, we have shown the practical advantages and flexibility offered by REST when applied to physical computing. In particular, we describe why a RESTful architecture is an excellent solution for leveraging an existing Wi-Fi infrastructure to build a loosely-coupled infrastructure for searching and interacting with networked devices. Even though the Web was designed as a hyper-linked system for multimedia documents, this chapter shows that a distributed location-aware infrastructure for embedded devices can be built solely using Web standards.

A world where every device, gateway, or router in a network could host a local Web server offering a JSON-based API for applications and an HTML-based user interface for human users, is technically feasible today. As evidenced by the increasing number of routers, printers, and consumer appliances on the market today have embedded Web servers, or at least a Wi-Fi/Ethernet interface, Web-enabled devices are likely to become commonplace. This way network-level information (e.g., routing tables or network load) and real-time data from the physical world (through sensors, etc) could be seamlessly integrated into Web applications,

therefore opening a whole new range of design possibilities to make the Web *more physical* and *more real-time*.

# CHAPTER 6

## Web-based Sensor Data Streams Processing

*"There ain't no rules around here! We're trying to accomplish something!"*

Thomas Edison (1847-1931)

So far, we have described an infrastructure for interconnecting, searching, and using heterogenous embedded devices using only Web standards. Nevertheless, to build end-to-end pervasive interactive applications, a low-latency and scalable push mechanism combined with an expressive and flexible querying system is needed for event-driven and streaming applications. In the first part of this chapter, we explore push-based messaging for the Web to evaluate their suitability for the Web of Things and propose a minimal Web-based messaging framework for NED applications. In the second part, we explore the state of the art in stream processing systems and propose a framework for collecting, processing, sharing and storing real-time sensor data streams, which facilitate the development of end-to-end applications for the Web of Things.

## 6.1 Part I: Event-driven NED Applications

Ensuring low-latency push communication between various actors in large distributed applications (e.g., Internet chats, music and video streaming, social networks, etc.) has been throughly explored for decades. As discussed in Chapter 2, interactive and reactive applications for NEDs require a lightweight notification mechanism suited for resource-constrained embedded devices. The need for an event driven infrastructure in the context of smart objects has been already proposed a decade ago by Langheinrich et al. [171]. Various solutions and tools have been proposed for many-to-many communication and event notification, as we describe later in this section. Nevertheless, to become part of the Web of Things, this messaging solution must be Web-compliant – that is not only use HTTP to transport data encoded with other protocols, but actually use HTTP as an application protocol by implementing the messaging system solely using RESTful patterns. This way, the framework can be integrated seamlessly with the deployed Web infrastructure. Finally, various interaction patterns must be supported (one-to-one, one-to-many, etc.) so that a unified framework suffices to build almost any type of application.

Before exploring the related work, we first present the basic terminology and introduce the core mechanisms commonly used in messaging systems. A common technique to deal with this problem is to use a *publish/subscribe* (hereafter: pub/sub) system to decouple data consumers from the publishers [119, 152]. Publishers send *messages* that encapsulate the data to be transmitted to a broker which is in charge of routing messages to the various subscribers, depending on the type or content of messages. Most messaging systems allow publishers to specify *topics* or *keywords* that are used for routing purposes. According to [189], four basic mechanisms of event notification – each with different strategies to determine the type of the event and the routing procedure:

- **Channels:** A publisher selects a named channel and publishes its notifications into this channel. Consumers subscribe to a channel and then receive all the notifications pushed by producers in that channel. This approach can somehow be compared to television, where the consumer tunes into a specific channel.

- **Subject-Based Filtering:** Each notification is annotated with a subject string that denotes a rooted path in a tree of subjects. Subjects can also be mapped directly into channels.

- **Type-Based Filtering:** Notifications are modeled in a type hierarchy with possibly multiple inheritance. The filtering process checks subtype inclusions and can use XPath expressions on the type hierarchy to determine the notification.

- **Content-Based Filtering:** Evaluates the notification as a whole by applying XPath expressions, matching templates, arbitrary programs and mobile code.

The channels, topics or subjects can be simple keywords or *tags*, but can also be more elaborate semantically annotated tags (machine tags, see Trend 2 in Section 3.1.1), for example `room:empty` for all the empty rooms, `error:battery` for all messages that are related to a problem with batteries. One can also use hierarchical topics, that is a structured subject tree that models the hierarchical relations between concepts, which is exactly the model introduced in Section 5.1.1. This way, all the events generated in 'Room 202' which uses the topic `/switzerland/zurich/ethz/buildingCNB/room202` are automatically forwarded to all the subscribers of that node, but also to the subscribers of the higher nodes such as `/switzerland/zurich/ethz` or `/switzerland/zurich`.

These strategies only govern the semantics of the message routing and what messages should be forwarded to which subscribers. The actual communication is taken in charge by an underlying messaging protocol, which can be simply a custom socket-based data exchange protocol, or an elaborate middleware that offers additional functions, such as quality of service control or secure authentication.

### 6.1.1 Message-oriented Middlewares

Different messaging protocols to carry out the underlying communication in publish-subscribe systems have been designed to meet the requirements of various use cases. In this thesis, we only consider Internet-based protocols that rely on the TCP and/or UDP transport protocols, therefore do not discuss non-IP or proprietary messaging systems that are often used in industrial applications. Nonetheless, the general principles we discuss here can be applied on top of any transport protocols, as long as the messaging abstractions presented above can be implemented on top of the primitives offered by the transport protocol.

More or less elaborate middleware solutions have been proposed for promoting code reuse by allowing people to work with pub/sub primitives without the need to re-implement similar functionality for each project. Among them, *Open Sound Control (OSC)* [32] is a protocol for real-time communication among computers, sound synthesizers, and other multimedia devices. OSC provides an open-source alternative to the highly successful MIDI protocol[1]. OSC uses subject-based filtering to deliver notifications to a group of listeners. The *Java Message Service (JMS)* [16] provides a message-oriented middleware, connecting producers and consumers by means of queues. Connections between producer and consumer can be either point-to-point (producer knows consumer and delivers the message directly to it) or publish/subscribe (with an intermediary *JMS Provider*). Producers select a channel (*JMS topic*) and deliver the message to the JMS provider which will then dispatch the message to all the clients subscribed to that channel.

[1] See: http://en.wikipedia.org/wiki/Musical_Instrument_Digital_Interface

The Advanced Message Queuing Protocol (AMQP) [5] is an open standard for message-oriented middleware. While most messaging protocols attempt to standardize the middleware at the API level only, AMQP is a wire-level protocol which only defines how the data is transmitted as a stream of octets. This means that is anything that can read and write this format can interoperate which each other regardless of the API or implementation language. Among others, AMQP features secure and reliable communication, and various message routing and queueing techniques (point-to-point and pub/sub). ZeroMQ (0MQ) [62] is a very lightweight embedded library for socket-based connection, but with support for various message-centric interaction patterns, such as N-to-N, pub/sub, and multicast. It acts as a powerful concurrency framework particularly suited for developing clustered applications as its asynchronous message-based model allows to build scalable processing tasks. The main difference with other messaging middleware solutions is that the socket centric approach it follows (explicit connection between components) emphasize a broker-less architecture[2]

The messaging protocols we have described above are well-known and widely used in industrial applications. Besides, various implementations are available for all them (both commercial and/or open source) that provide additional QOS (delivery guarantee, persistence, transactions, etc.). Unfortunately, they were all primarily designed for desktop computers and mainframes, with little concern about lightweight and simple messaging required for NED applications. More recently, pub/sub solutions for resource-constrained devices have been proposed. For example, IBM's *MQ Telemetry Transport protocol (MQTT)* [29] is a system to enable messaging for tiny devices such as sensors and actuators. MQTT uses a channel-based eventing scheme, where devices publish their events together with a topic (channel). In the background, a message broker[3] receives the events and dispatches them to all the subscribers. MQTT-S [155] is an extension of the MQTT that supports hierarchical topics and attempts to integrate WSNs into traditional computer networks. MQTT-S was designed to be integrated with any message broker that supports MQTT using a gateway to translate between both protocols. The difference lies in the fact that MQTT-S is a simplification of MQTT that requires much less memory and processing this way it can run directly on resource-constrained devices. *TinyDDS* [87] is a topic-based pub/sub mechanism designed for sensor networks, which attempts to reduce the coupling in traditional WSN deployments. Based on the Data Distribution Service standard [194], TinyDDS provides a higher level library for data aggregation and event detection to simplify the development of WSN applications. *Messo* and *Preso* are two protocols for messaging in WSN applications [206]. Messo is a publishing protocols that uses MAC layer acknowledgments to guarantee delivery at the application level, which might not be suited for all applications. Preso is a subscription protocol which distributes reliably messages to all subscribers of a topic.

### 6.1.2 Web-based Streaming and Messaging

[2]See discussion: http://www.zeromq.org/whitepapers:brokerless
[3]For example IBM MQ WebSphere http://www.ibm.com/software/websphere/.

As mentioned in Section 3.1, the client-server nature of HTTP makes the Web a priori an excellent candidate to read/write data from/to embedded devices [139]. However, this paradigm falls short when it comes to support the event-driven nature of many NED applications. The request-response (pull-based) interaction type is of limited use for devices that usually have low-duty cycles (i.e., sleep most of the time), and limited bandwidth or energy.

Various media streaming have been used on the Internet to transmit video and audio content, using protocols such as the *Secure Real-time Transport Protocol (RTSP)* [79] or the *Real-Time Transport Control Protocol (RTCP)* [159]. Streaming media protocols have enabled transmission of potentially infinite multimedia documents, such as Internet radio stations or video streaming. Sensor streams are similar to streaming media in this respect, however, streaming media mainly support '*play*' and '*pause*' commands, which is insufficient for sensor streams where more elaborate control commands are needed. Other Internet-based protocols such as DPWS were designed to support more elaborate streaming and eventing for embedded devices. Unfortunately, these solutions were not designed for resource-constrained devices such as sensor nodes, but for more powerful appliances that are connected using Wi-Fi/Ethernet and have fairly powerful CPUs, such as video games consoles or set-top boxes. Besides, as mentioned in Chapter 2, these protocols do not integrate with the Web as they reduce the role of HTTP only as transport and use a custom, complex application protocol on top of it.

To fill this gap, different Web-compliant techniques for implementing asynchronous, push-based interaction patterns have been developed and help realize the real-time Web. We survey the various techniques for real-time Web-based messaging and discuss their applicability to the data-centric, stream-based nature of sensor-driven applications.

### RSS/ATOM

Feeds have become the most popular format for machine-readable data on the Web and are increasingly used for any time-ordered information streams of different content types. For example, photo management services such as *Flickr* or *Picasa* provide access to the stream of published photos with any service or tool supporting feeds. Feeds can be based on various data formats, such as *RSS* or the more well-defined *Atom*. The *Atom Publishing Protocol (AtomPub)* is an extension of Atom which covers all REST verbs, so that users can interact with an AtomPub-enabled collection by creating, updating, and deleting entries. This model is interesting for sensor data streams, which are usually a time-ordered collection of sensor readings. Using the same paradigm to interact with that data – both from the application and sensors' point of view – is appealing as it simplifies greatly data integration and development of new applications. Devices could `POST` new data to a collection using AtomPub, and clients pull the data from the collection using Atom feeds.

Even though AtomPub provides an interesting metaphor and concrete tools to access time-ordered data, it is based on a polling model (see Figure 6.1) which is inappropriate for low-latency notifications. Besides, it increases network traffic because clients need to continuously request updates without knowing when they are available. This is inefficient for embedded devices, where a push-based model is necessary to minimize resources and extend battery lifetime.

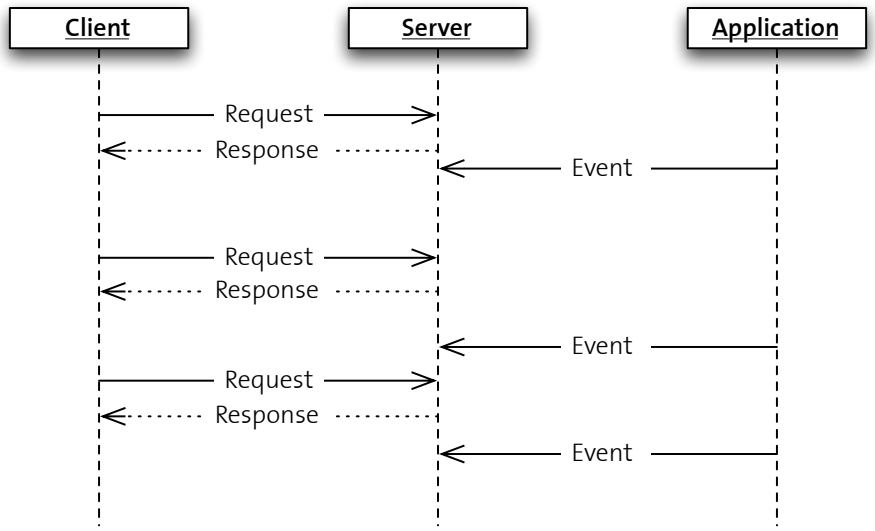



**Figure 6.1: Classic Web polling.** Usually implemented using AJAX, the client polls the server for new data periodically at a fixed time interval (e.g., *"fetch new data every second"*). This generates much traffic in both directions even when there is nothing new on the server for a while. New events will be received at the next polling period only (so the delay to receive an event is bound by the polling frequency).

**Extensible Messaging and Presence Protocol (XMPP)** is an open communication protocol for messaging based on XML messages and powers a wide range of applications including instant messaging[4]. Although successfully used by various services, XMPP is an overly complex standard that is too heavy for the limited resources of devices used in sensor networks. Various attempts to use it for sensor data have been proposed [60, 195, 153], unfortunately, no concrete implementation and along with a performance evaluation of the usage of XMPP on embedded devices has been found in literature.

### Comet

*Comet* is an umbrella term that refers to a range of increasingly popular techniques for circumventing the limitations of HTTP polling, by introducing a real-time dimension to the Web with push-based communication. This model enables a Web server to push data back to the

[4]Google Talk is based on XMPP, see: http://code.google.com/apis/talk/open_communications.html

browser without the client requesting it explicitly. Since browsers are not designed with server-sent events in mind, Web application developers have tried to work around several specification loopholes to implement Comet-like behavior, each with different benefits and drawbacks.

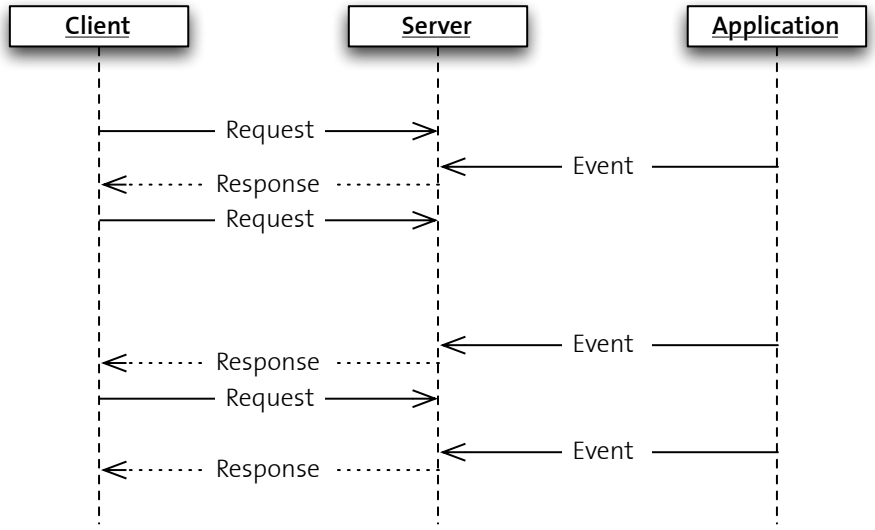



**Figure 6.2: Long polling.** The client issues a new request as soon as it receives the answer from its previous request. The new request remains pending until the server receives a new event from the application. The network traffic is reduced and the latency is minimized, however, the client must be put on hold with an open HTTP request that is waiting for an answer (the loading icon in a browser.)

The first technique, called *long polling* is shown in Figure 6.2, where a client requests information from a server using a classic HTTP request, but instead of receiving the response immediately with no updates, the server will hold the request until an update is received from the application, at which time it will finally return the update to the client by answering to the request which was being held idle. As soon as the client received the response, it will immediately send a new request for an update which will be held until the next update comes from the application.

Another general class of techniques called *HTTP server push* (or also *HTTP streaming*) exploits the possibility for a Web server to keep the TCP connection open after the response data has been served to a client, and then use it to send further events to one or multiple clients, as shown in Figure 6.3. BOSH [196] is a long-lived HTTP technique used in XMPP/Jabber that defines a transport protocol that emulates a long-lived, bidirectional TCP connection between two entities (such as a client and a server) by efficiently using multiple synchronous HTTP request/response pairs without requiring the use of frequent polling or chunked responses. Another specification called Bayeux [207] is a protocol for transporting asynchronous messages over HTTP, with low latency between Web servers and clients. *Infinite* responses are used to send multiple messages via one single HTTP response, by sending chunks of incomplete HTTP responses to the client. This could be also done using the MIME type `multipart/x-mixed-replace`, which is interpreted by browsers as a document that would change every time the server had an update to push to the clients. These techniques have been implemented by the Opera browser in

2006 under the name of *server-sent events*, which has led to the development of WebSockets in HTML 5.

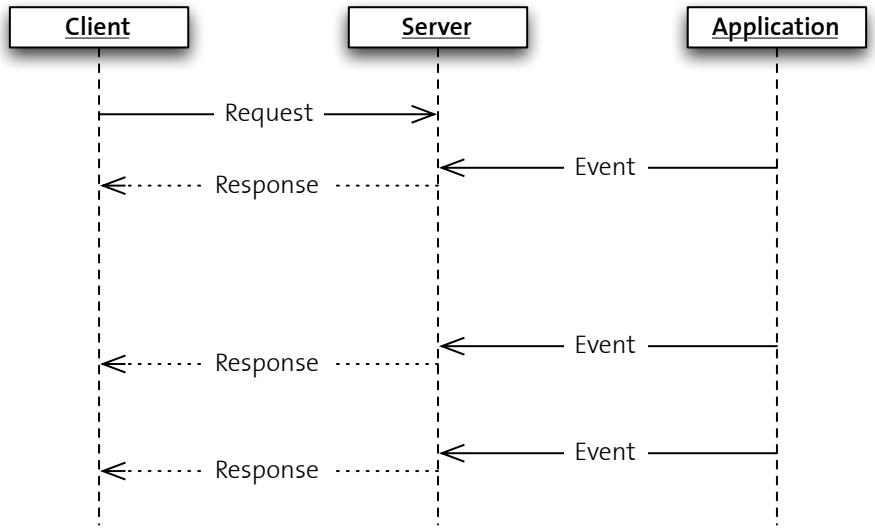



**Figure 6.3: Comet push.** The client opens a TCP-IP connection with the server, which is then kept open by the server. Subsequently, each new event from the application is then pushed to the client over the connection kept open, therefore the client does not need to issue further requests. Both event delivery latency and network traffic are minimized.

Various online services that support server side eventing and particularly Comet, have been recently appearing such as Lightstreamer [23] or Ajax push engine (APE) [3]. Some of those are highly scalable (for example, APE supports over 100K simultaneous users according to their Web site). A core advantage of such frameworks is a cross-browser support and the ability to integrate real-time streaming data in any Web-page using Javascript and other Web technologies.

Even though push solutions are known as more reactive, but less scalable [91], recent work has shown that Web push notifications on embedded devices are not only possible (for example Yaler[5] uses ReverseHttp [45] on Arduinos), but also present various advantages. Using an embedded Web server, Duquennoy et al. [113] showed that Comet-based streaming is actually quite scalable even the most scalable solution, as their solution can support 256 simultaneous clients using only 10kB of RAM. The reason is that the event-driven nature internally used by embedded applications – especially the embedded Web server – supports efficiently event-driven notifications. The actual limitation in scalability of push-based systems seems to come from the traditional OS constraints that break the native event-driven model of the hardware. This might be changing as event-driven applications will become increasingly performant and robust, especially when designed for the increasingly popular multicore processors.

The messages are routed via named channels and can be delivered between between servers and clients. By default, pub/sub routing semantics are applied to the channels, but other routing

[5]Online: http://yaler.net

models are also supported. Comet servers and clients make frequent use of topics which are useful when different objects want to send data to various interested parties subscribed to updates.

The major advantages of Comet beyond low-latency over the Web is that communications is considered as normal HTTP, that is it works even though proxies, NATs, and firewalls where everything else than Web traffic (on port 80) is blocked. Besides, as all Comet interactions are initiated by the client, every client can be individually addressed from the server, even when they do not have a publicly visible URI or IP address. The major drawback of Comet techniques is that they only specify a polling-free data transport and not a complete messaging protocol with elaborate routing strategies.

#### Web Hooks

Another solution for clients and applications to receive notifications from external Web sites is to use HTTP callbacks. By "subscribing" to an event, users specify a callback URL where the application will `POST` data to each time an event occurs. This mechanism has been used by PayPal online service which allows to specify a URI on your Website that will be triggered by PayPal servers once a payment has been confirmed. Although cleaner and simpler than Comet as it only requires clients to also have a Web server, this model is not suited for clients that do not have a public URL where data can be posted to (which is usually the case when clients are behind a firewall or in a private network).

The pubsubhubbub (PuSH) protocol [122] focuses on server-to-server publish/subscribe functionality. It uses Web hooks and leverages the Atom feed format. A data provider (e.g., a blog) declares a number of *hubs* in its feed using the `<link>` element (see Section 5.3.2). A subscriber initially fetches the Atom URL of the content provider, and if the Atom file declares its hubs, the subscriber can then subscribe the feed's hub(s) to be notified when new content is available. To subscribe, the client has to include a callback URL in the subscription request. The hub then sends a HTTP `POST` request including the most up-to-date feed to that callback URL whenever the feed was updated. The publisher side of the protocol works by the feed server pinging all hubs as soon as the feed was updated. The configured hubs then fetch the updated feed and multicast the new content out to all registered subscribers. The authors of the protocol claim that by using many *hubs*, the protocol allows a very scalable and reliable mechanism for publish/subscribe over the Web. One disadvantage of PuSH is that because a HTTP callback mechanism is used, clients have to be accessible from the Internet and they have to run a Web server, which makes PuSH suited mostly for server-to-server communication in the open Web.

RestMS [44] provides messaging (support for queuing and routing) via an asynchronous RESTful interface over HTTP. On top of HTTP, a RESTful transport layer is defined which allows to

use long polling to receive events from the server. The RestMS protocol also can be extended via various "profiles", which are different routing strategies allowing interoperability with other messaging protocols, such as AMQP. The idea of providing a Web-level messaging system is promising, however, it seems that the specification has been overly complicated, which limited its adoption. At the time of writing, the project seems frozen (no updates for over two years), no known complete implementation exists, nor services that use it.

#### WebSockets

The Web Hypertext Application Technology Working Group (WHATWG), which is currently working on the HTML 5 specification, included a polling-free server to client data transport mechanism called WebSockets [151]. Along with the embedded Web server in the Opera unite browser, these recent boom and development in Web push techniques illustrate that server-sent events are to become popular very soon. The following example shows how to open a WebSocket and process messages using Javascript:

```
1  var socket = new WebSocket("ws://webofthings.com/channels/ethz/CNB");
2  socket.onopen = function() {
3        alert("Socket is open");
4  }
5  socket.onmessage = function(msg) {
6        [... do something with the msg received ...]
7  }
8  socket.onclose = function() {
9        alert(" Socket is closed");
10 }
```

In practice, a WebSocket connection is established via an upgrade from the HTTP protocol to WebSockets during the initial handshake connection between client and server, which allows both parties to send messages in full-duplex mode over a single TCP/IP connection initiated by the client. The upgrade is done using the `Upgrade: WebSocket` and `Connection: Upgrade` headers in the initial request, which allows a smooth integration with the Web and facilitates proxy (firewalls, routers, etc.) transversal.

### 6.1.3 Discussion: Messaging

Nowadays, integration of different WSNs is done in a fairly rigid manner: gateways are highly customized for specific protocols, therefore devices can not be integrated into another WSN's gateway easily. As identified above, the loose coupling introduced by a publish/subscribe mechanism offers many advantages, among which a better scalability and a simpler programming

model thanks to higher level abstractions. These advantages are desirable for the Web of Things, where a pub/sub mechanism to exchange sensor data and commands between devices and applications will allow more scalable and flexible applications to be built.

Messaging protocols such as JMS or AMQP have been successfully used to build large-scale systems in areas such as banking and e-commerce. XML-based messaging system (XMPP) introduces artificial overheads to increase interoperability, which results in larger bandwidth usage, even though binary XML encoding can be used to compress data.

As illustrated earlier, the Web is becoming an ecosystem of services that can be queried using standard and uniform Web APIs (increasingly based on REST). However, Web content still largely follows the multimedia document-based model, which is not suited for the stream-based nature of sensor data. To support the complex data-centric queries commonly used in stream processing systems, one must explore more transparent data models to expose sensor data streams over the Web so that highly expressive queries can be expressed using simple and open Web standards. With the advent of the real-time Web, numerous protocols are making this possible. While there are HTTP bindings to most of these protocols, client libraries for these protocols are still overly complex and rather immature, which makes the adoption barrier to messaging protocols quite high. By keeping the complexity at the broker in a very simple pub/sub system, one can more easily benefit from the looser coupling offered by a messaging system. A fully Web-based messaging middleware would allow devices, gateways, brokers, and all interested applications to interact easily without any prior knowledge about each other. Furthermore, as gateways are optional, they can easily be bypassed in case they fail, increasing the robustness of the whole system.

In the next section, we propose a messaging solution designed to seamlessly blend into the existing Web. We borrowed the core ideas from the messaging semantics used in traditional message-oriented middleware and also the notion of intermediate nodes to decouple producers from consumers. Brokers are Web resources linked via URI, simplifying (re)use and at the same time providing a loose coupling. The interface between the intermediaries follows the REST principles by using HTTP. The payload of messages can then be encoded as JSON objects that are Javascript-friendly, which makes it easy to integrate streaming data into Web mashups.

## 6.2 Web Messaging System (WMS)

According to our discussion in the previous section, designing a flexible and scalable push-based notification and streaming mechanism will be necessary would be desirable for the Web of Things as devices could directly benefit from it without requiring an additional layer on top of HTTP (as is the case when using XMPP or WS-Eventing). To provide a simple solution for Web-based messaging, we developed *Web Messaging System* (WMS), a lightweight pub/sub

messaging system that can be used with embedded devices. Rather than developing a custom messaging protocol on top of HTTP, WMS only specifies the core functions of a pub/sub system using RESTful design patterns over HTTP interactions. In essence, WMS is similar (and hence directly mappable) to RestMS or pubsubhubbub, but it is more generic and simpler so it can be easily understood and implemented on devices with constrained resources. Being a genuine RESTful protocol, it can be used in many other contexts, including typical Web applications.

The central idea behind WMS is to shift the perspective by considering devices not only as passive HTTP servers that only answer requests, but also as active clients that can issue requests towards other servers, just like Web Hooks. External clients can then subscribe to events on devices and will be notified via Web hooks every time a specific event is triggered. Devices will emit events via a HTTP `POST` message to a callback URL specified at subscription time. The following (simplified) HTTP request example will add a subscriber to a fire detector device:

```
1  POST /subscriptions
2  Host: DEVICE-URI
3  Content-type: application/x-www-form-urlencoded
4  Content:
5  keyword=alert&wms.cb=http://[SUBSCRIBER-URI]/callback
6
7 201 CREATED
8 Location: http://[DEVICE-URI]/subscriptions/32746
```

The `Location` header contains the URI of the newly created subscription RESTful resource (which can then be updated with `PUT` or deleted with the `DELETE` HTTP methods). Every time an event is generated, that is it matches the keyword "alert", it will trigger the device to `POST` the event sequentially to all the callback URLs registered for that particular keyword. This allows a simple fully-HTTP publish-subscribe mechanism directly on devices. Although the scalability of this mechanism is very limited and depends on the resources available on each device, using a RESTful messaging for sending notifications facilitates the *outsourcing* of the notification process to more powerful reverse proxies or intermediate nodes.

One possible usage scenario for WMS is a back-end infrastructure with many NEDs connected to a gateway that hosts a simple WMS broker, as shown in Figure 6.4. Devices could be individually addressed using any Web client and a secure authentication mechanism, for example to reconfigure the smart meter. At the same time, devices can be configured to send notifications based on simple rules (every $n$ seconds, if above a threshold, etc.) to a specific endpoint in the cloud, or simply to the WMS broker on the gateway it is attached to (by default), which will forward it to the building gateway (that bridges the network with the external world).

The gateway offers a RESTful API to use the eventing system and provides the following resources to manage interactions:

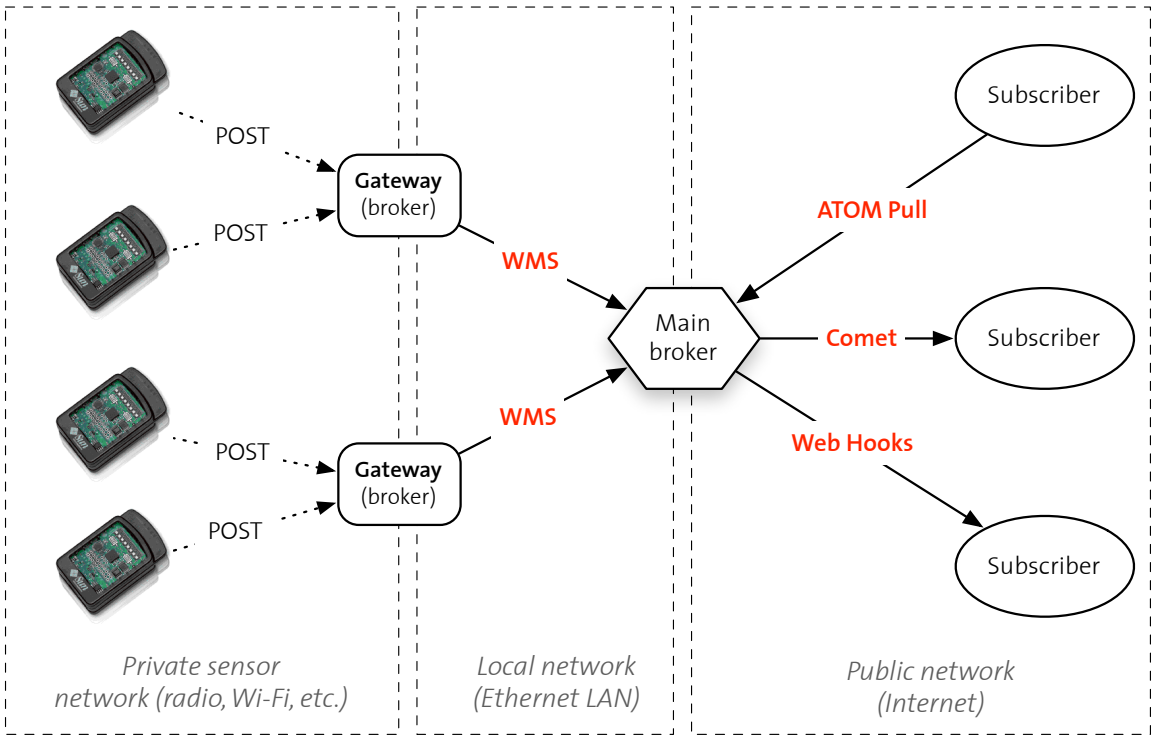



**Figure 6.4:** The general model of Web-based messaging. Devices push their data to gateways using standard HTTP `POST`, AtomPub, or other proprietary protocols. Gateways will then forward it to other gateways or to a powerful broker using WMS. External users can fetch the data using the most appropriate method for their needs - depending on the protocols supported by the broker.

- `/channels` every sub-resource under this resource (for ex. `/channels/ethz/CNB/room102`) represents a hierarchical channel where entities can post data to.
- `/subscriptions` contains each subscription of entities to individual channels.
- `/channels/[channel]/publishers` contains all entities that are publishing data on the channel.
- `/channels/[channel]/subscribers` contains all entities that are subscribed to events on the channel.

A user that wants to subscribe to notifications about a channel, creates a new subscription by `POST`ing the following HTTP request to the channel it wants to subscribe to:

```
1  POST /channels/ethz/CNB/room102
2  Host: GATEWAY-URI
3  Content-type: application/x-www-form-urlencoded
4  Content: wms.cb=http://[SUBSCRIBER-URI]/callback
```

where the `wms.cb` parameter denotes the callback URI messages shall be posted to. Note that `wms.` specifies the format that should be used for the callback. Each new message posted on

the channel will be POSTed back to this subscriber at the URL wms.cb. Publishers can then push data to the gateway by POSTing the following request on the broker:

```
POST /channels/ethz/CNB/room102
Host: GATEWAY-URI
Content-type: application/x-www-form-urlencoded
Content: pub-url=http://device1&temperature=21
```

This device has never been registered with the gateway, so it includes its root URI in the posted parameters, providing the gateway with a possibility to automatically scan it for Microformats to retrieve required information to be able to register it using the procedure described in Section 5.3. The message is posted directly to the channel /ethz/CNB/room102 without any need to previously create it. Other parameters are treated as tags (in this example temperature), which can be used by publishers to annotate data they send and subscribers to filter out messages that do not contain tags they are interested in.

The modular approach used to implement the gateway allows users to easily extend the messaging module using the strategy pattern. Just like the discovery strategies proposed in Section 5.3.2, users can easily add new messaging protocols to the gateway (for example support for Comet, WebSockets, XMPP, email, etc.), which allows to bind the proposed network with other applications that do no support HTTP or WMS natively. Obviously, support for other protocols should only be considered at the main gateway (bridge between LAN and Internet) for external clients.

### 6.2.1 Web Messaging Evaluation

We have been interested by evaluating the performance of a pure Web-based messaging system, and how various parameters affect Web-based messaging throughput and response times. First, we evaluate the performance of a Web push system to understand how they perform with NED applications. In the second evaluation, we explore more in detail our implementation of WMS and evaluate how it works with a complete application. Unlike in part A, where the evaluation is done in isolation (we focus only on the message exchanges), part B evaluates the whole end-to-end messaging, including the broker.

#### Evaluation Part A - Minimal Web Messaging

To assess the suitability of Web-based push messaging for NED application, we have evaluated the performance of a simple callback based notification initially presented in [243, 229] using a custom setup shown in Figure 6.5. We have implemented a lightweight Web hook-based notification mechanism on a gateway running on a server with 1.1GHz, 2GB RAM, Gigabit Ethernet,

and running Gentoo GNU Linux. The client machine used to simulate HTTP clients has 2 x 2.13GHz, 8GB RAM, Gigabit Ethernet and runs Gentoo GNU Linux. As we are interested in the scalability of Web-based messaging not at the WSN level, but at the infrastructure level, we do not explore in this section the scalability in the WSN, and measure only notifications between two desktop machines and not directly on the device.

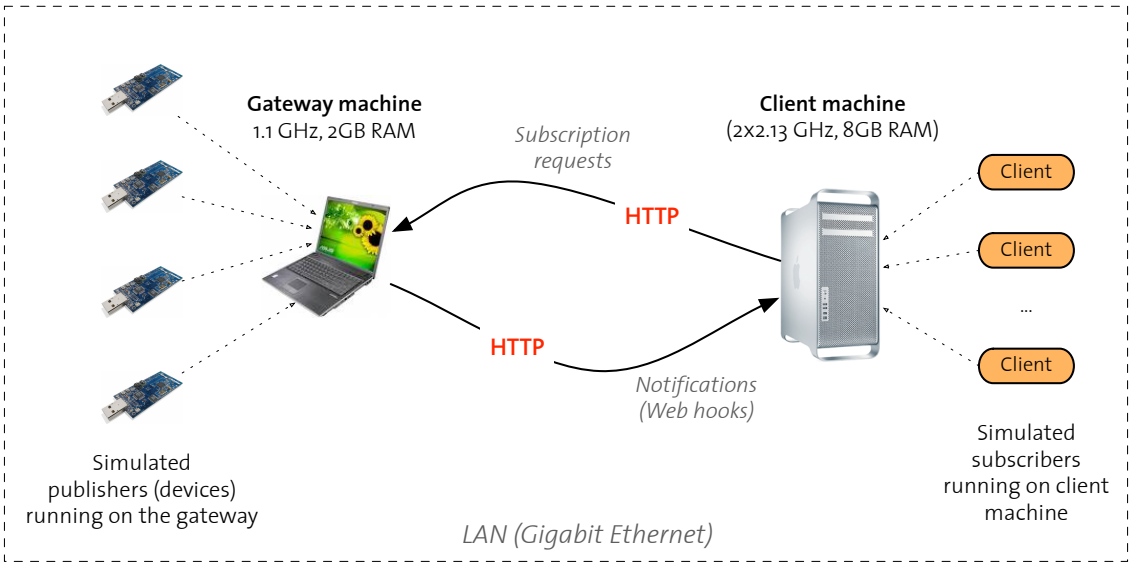



**Figure 6.5:** Network structure used in the evaluation of HTTP messaging. A client machine that simulates subscribers is connected to a gateway that simulates publishers (virtual devices). Both machines are wired using Gigabit Ethernet and are on the same local network.

**Many Subscribers.** In many scenarios, one can expect a small number of entities publishing data that is interesting to wider audience, for example a pollution sensor installed by local authorities shared openly with citizens. For obvious performance reasons, requests should not be answered from the device directly, but the load should be outsourced to a proxy that handles the notifications to many subscribers. We first evaluate the scalability of Web messaging in such a scenario, where many clients subscribe for events sent by a device.

The test client started an event sink to receive events on respectively 50, 100, and 200 different ports, and for each port an event subscription was posted to the gateway. The gateway then generated artificial events (containing the generation timestamp) that were delivered to all the subscribed ports. The test client measured the arrival time and from that computed the delay for each arriving event based on the timestamp. Figure 6.6 shows the Cumulative Distribution Function (CDF) of the delivery time to notify all the subscribers, and key statistics about the measurements are given in Table 6.1.

One can see that the performance is acceptable given a custom-made application, and in average it scales linearly with the number of subscribers. For 50 subscribers, the average time needed to send a notification to a client is 97 ms, and 360 ms for 200 subscribers, which remains very

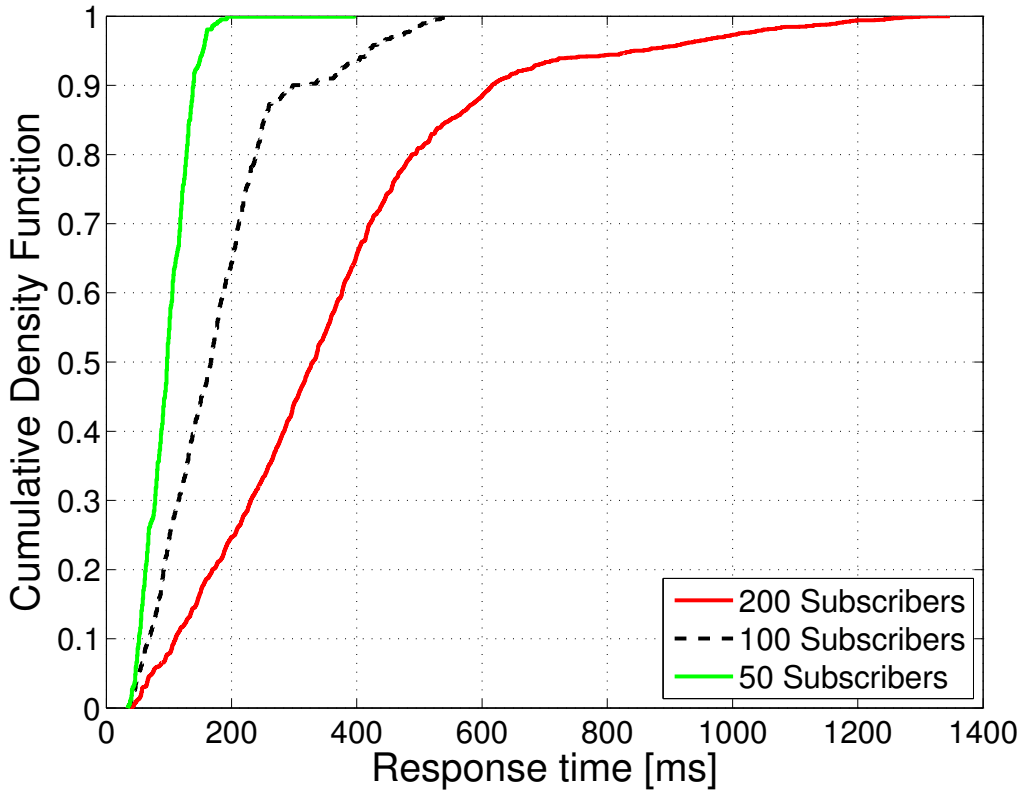



**Figure 6.6: Many subscribers (HTTP):** CDF of event delivery times measured for respectively 50, 100, and 200 subscribers.

| | 50 Subscribers | 100 Subscribers | 200 Subscribers |
|---|---|---|---|
| Min | 34 ms | 35 ms | 34 ms |
| Max | 396 ms | 553 ms | 1347 ms |
| Mean | 97 ms | 180 ms | 360 ms |
| q80% | 128 ms | 239 ms | 330 ms |
| q95% | 154 ms | 416 ms | 489 ms |
| q99% | 177 ms | 511 ms | 1169 ms |

**Table 6.1: Many subscribers (HTTP):** Various statistics for delivery times of events to many subscribers ($q - X = Y$ means that X% of the events are delivered with a latency below Y.)

reasonable. However, the scalability of the "worst-case" conditions scales less linearly as 99% of messages are sent within 177 ms for 50 subscribers, but takes 1.17 seconds for 200 subscribers. One can see that our simplistic implementation starts to behave erratically for some messages as the number of subscribers increase. One reason can be that all the simulated subscribers run on the same machine, therefore this causes a bottleneck at the network and application-level which degrades the performance. This could be solved by using a more through implementation of both the client and server.

**Many Events.** In this experiment, we have measured the gateway performance when many different events are triggered simultaneously and have to be delivered to subscribers. A virtual device generated different events (containing the time the event was received on the gateway) at regular intervals. The test client received the events and computed the delay between arrival time and generation time. Table 6.2 contains statistical measures of the measured response times, and Figure 6.7 shows the cumulative density function of the time needed for the subscriber to receive respectively 150, 300, and 600 events.

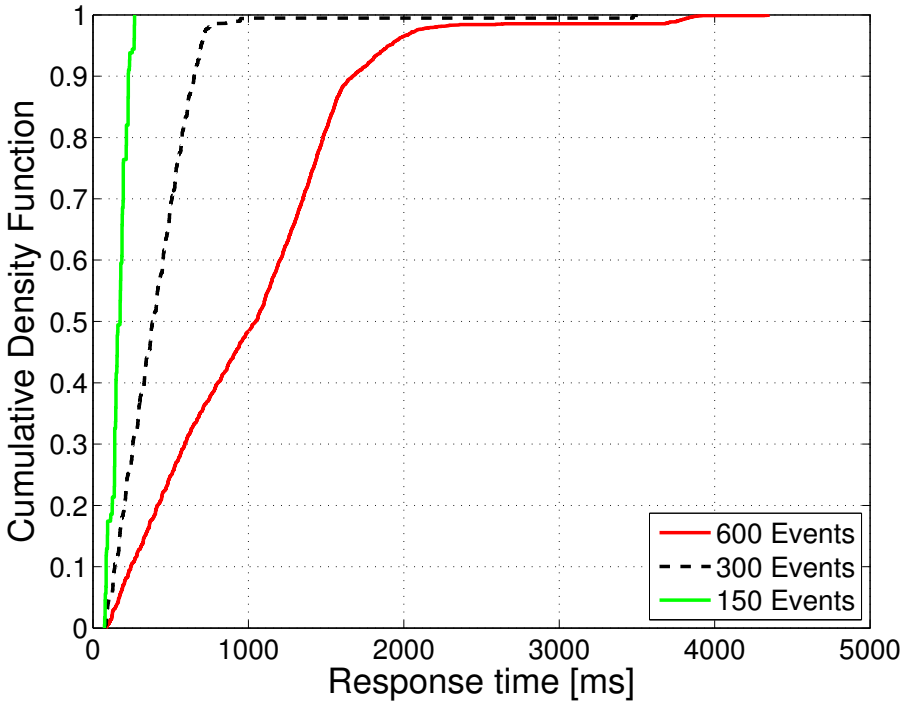



**Figure 6.7: Many events (HTTP):** Response time to deliver events to a client, with respectively 150, 300, and 600 events.

| | 150 Events | 300 Events | 600 Events |
|---|---|---|---|
| Min | 75 ms | 74 ms | 79 ms |
| Max | 268 ms | 3515 ms | 4353 ms |
| Mean | 166 ms | 408 ms | 1019 ms |
| q80% | 216 ms | 573 ms | 1481 ms |
| q95% | 266 ms | 695 ms | 1906 ms |
| q99% | 268 ms | 947 ms | 3762 ms |

**Table 6.2: Many events (HTTP):** Various statistics for delivery times of many events to a single client.

In this case, one can see that the performance remains acceptable for many events sent to the subscriber (99% of messages are sent within 268 ms). However, for many events (600), the performance degrades significantly, and it takes in average 1019 ms per message when all the

messages are to be delivered simultaneously (and 1% of messages need more than 3.76 seconds to be delivered). The reason is very likely the client cannot handle so many messages to be delivered per second, and a more efficient handler could be required.

**Many Publishers.** In this experiment, we evaluate the situation where many different devices (publishers) are attached to the gateway, and each device generate events at random intervals comprised between one and five seconds. Three runs have been performed with respectively 50 devices, 100 devices, and 200 devices attached. The delivery time was computed as the delay between the event generation and arrival times, and results are shown in Table 6.3, with the CDF plotted in Figure 6.8.

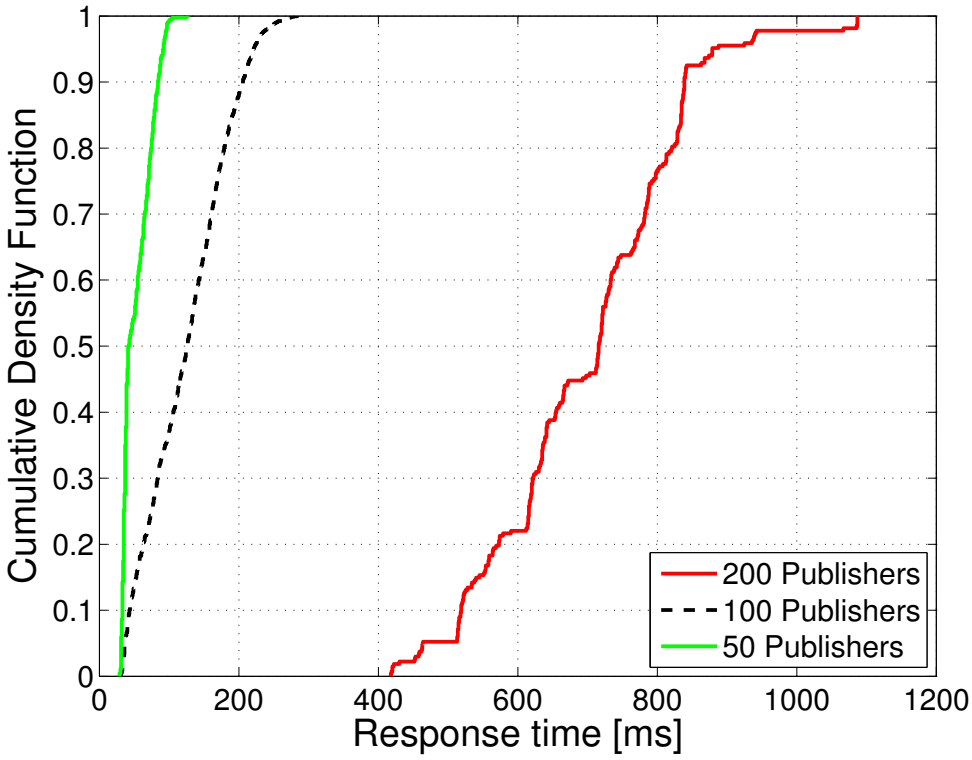



**Figure 6.8: Many publishers (HTTP):** Response time to deliver events to a client with respectively 50, 100, and 200 publishers.

In this case, we observe that up to 100 publishers, the performance is reasonably low and scales linearly with the number of publishers (devices), as 99% of messages are sent within 253 ms, with 125 ms in average. The performance degrades significantly with the number of publishers, as when doubling the publishers (200) it now takes in average 0.7 seconds to deliver all the message, and 1% of messages need more than 1.09 seconds to be delivered.

| | 50 Publishers | 100 Publishers | 200 Publishers |
|---|---|---|---|
| Min | 29 ms | 30 ms | 418 ms |
| Max | 127 ms | 284 ms | 1087 ms |
| Mean | 52 ms | 125 ms | 699 ms |
| q80% | 75 ms | 179 ms | 821 ms |
| q95% | 92 ms | 221 ms | 880 ms |
| q99% | 98 ms | 253 ms | 1087 ms |

**Table 6.3: Many publishers (HTTP):** Various statistics for delivery times from many publishers to a single client.

### Evaluation Part B: WMS

This second evaluation explores the performance of a complete embedded WMS implementation in an end-to-end scenario, not only the messaging in isolation. We have performed various experiments using the simulation environment designed by Vlatko Davidovski for his masters thesis [101]. Using the experimental setup shown in Figure 6.9, various parameters of WMS as a messaging system have been evaluated, but only the relevant summary of these results is provided here. A more detailed analysis of WMS according to different parameters such as message size or local storage usage is available in [101].

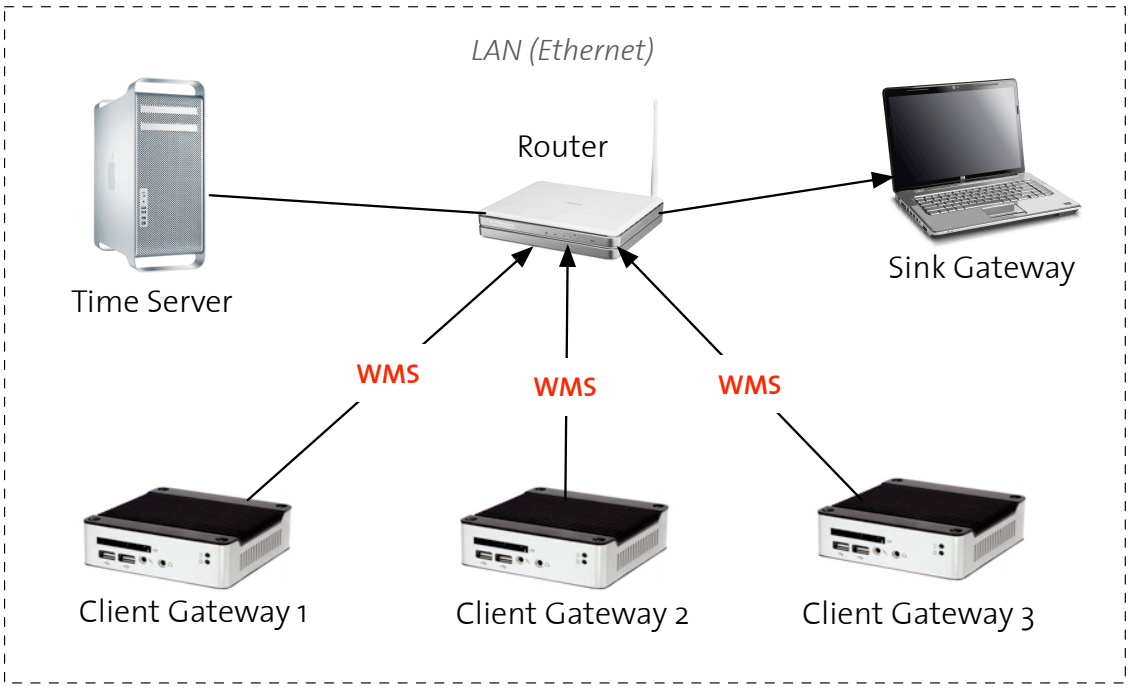



**Figure 6.9:** Network structure used in the evaluation of WMS. Three client gateways are connected to a sink messaging connector. All connections are wired using Ethernet and are in the same local network.

**Many Subscribers.** In this experiment, we explore the scalability of the system with respect to the number of subscribers. One sink (laptop) and three gateways (Norhtecs[6]), each containing one virtual entity are used in the experiment. Each entity continuously generates messages of 1KB size during the whole duration of the experiment. These messages are generated at random intervals based on the Poisson distribution (mean value 500 ms). The sink (a laptop machine) simulates respectively 20, 40, and 80 clients that subscribe each to all the events of each gateway (that is respectively a total of 60, 120, and 240 subscriptions in total), therefore each message will be sent to the sink multiple times (but to different channels).

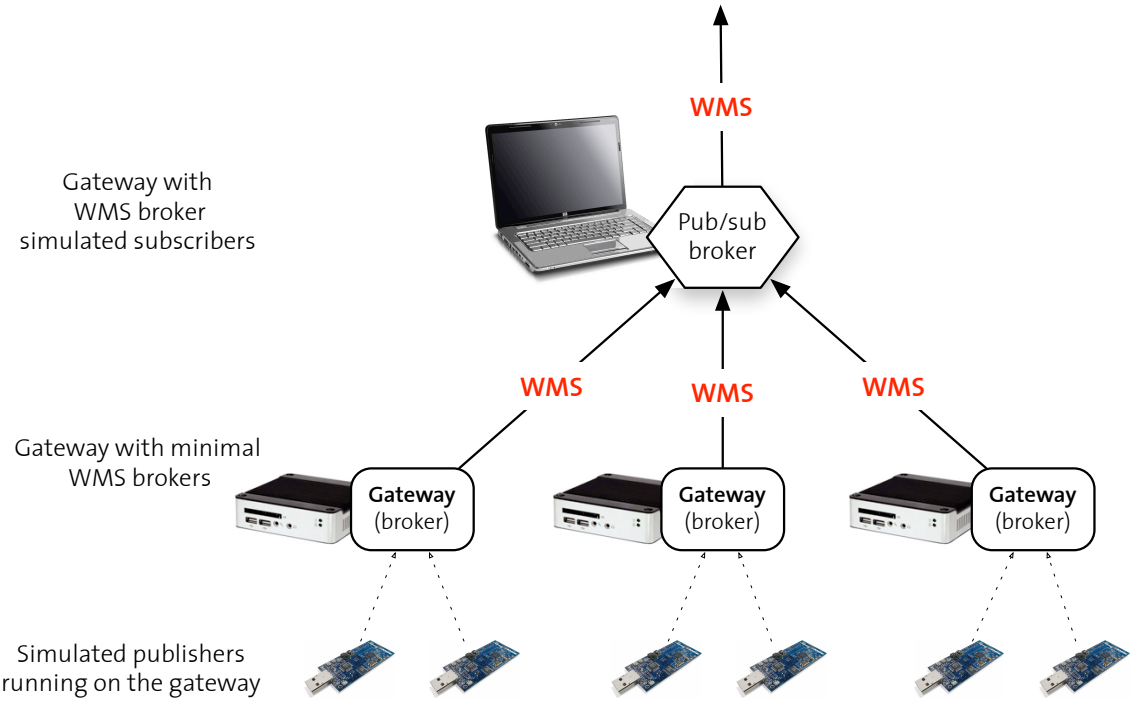



**Figure 6.10:** Application structure of the application. Each gateway runs locally a minimal pub/sub broker that receives messages from the devices. Simulated devices send data to the gateways using WMS, thus are not directly integrated in the gateway code as in the Part A. This reduces the performance, but also decreases the binding between the devices and the gateway, which allows these interactions to be RESTful.

We measured the message life cycle time (MLCT), which is the time between an event has been generated and transferred to the last subscriber (only internal times in the gateway, therefore does not take into account the network transmission time). Key statistics are shown in Table 6.4, and a CDF of the MLCT is shown in Figure 6.11

With 60 subscriptions, the system performs reasonably well (average 873 ms, 95% of messages are delivered under 1054 ms), and the MLCT scales linearly with the number of subscribers in average (3.365 seconds with 240 subscribers).

[6]NorhTec MicroClient Sr. http://www.norhtec.com/products/mcsr/index.html

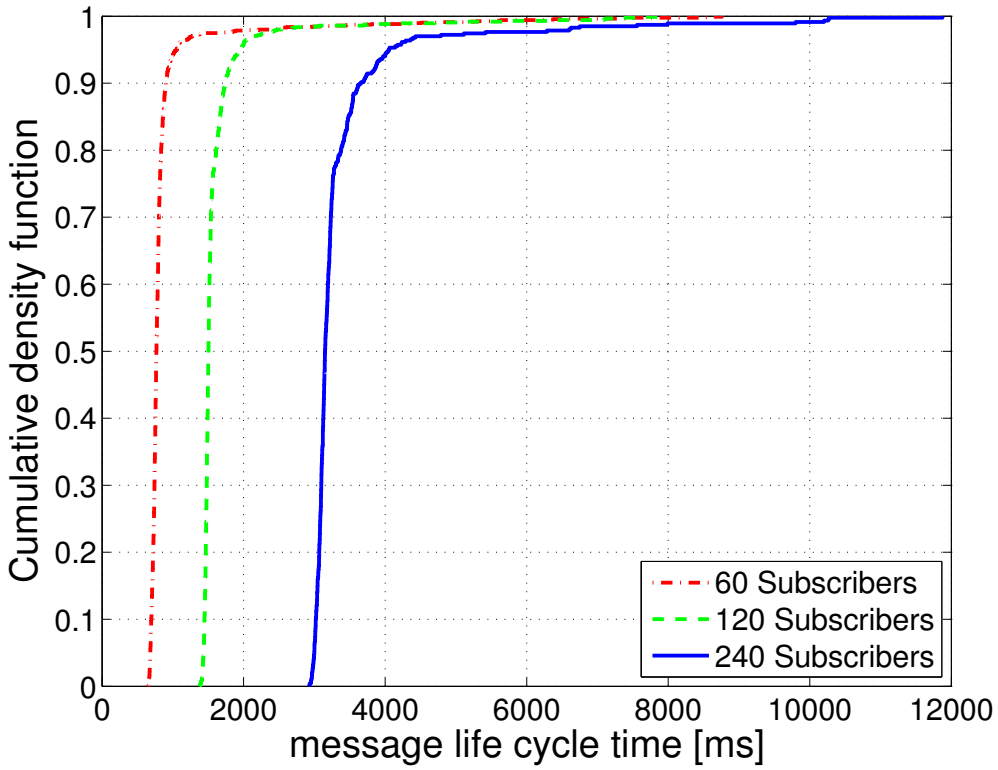



**Figure 6.11: Many subscribers (WMS):** Message Life Cycle Time for various subscribers, respectively 60, 120, and 240.

| | 60 Subscribers | 120 Subscribers | 240 Subscribers |
|---|---|---|---|
| Min | 653 ms | 1382 ms | 2930 ms |
| Max | 8760 ms | 7965 ms | 11876 ms |
| Mean | 873 ms | 1617 ms | 3365 ms |
| q80% | 835 ms | 1613 ms | 3378 ms |
| q95% | 1054 ms | 1943 ms | 4056 ms |
| q99% | 4500 ms | 4715 ms | 9475 ms |

**Table 6.4: Many subscribers (WMS):** Various statistics for delivery times from an event to many subscribers.

**Many Publishers.** Another important aspect is the behavior of the system when many devices (publishers) are connected. This second test evaluates the effect of the number of concurrent devices (publishers) on the overall performance of the gateway. The setup is the same, with three gateways containing each respectively 10, 25, and 50 virtual entities (simulated devices), each generating a 1KB message with at a random time defined with a Poisson distribution (500 ms mean). The sink registers to each entity, therefore receives all the messages emitted by respectively 30, 75, and 150 simulated devices.

Statistics about the MLCT are given in Table 6.5, and the CDF of the MLCT is shown in Figure 6.12. Even with 150 devices each generating a message every 0.5 seconds in average, the

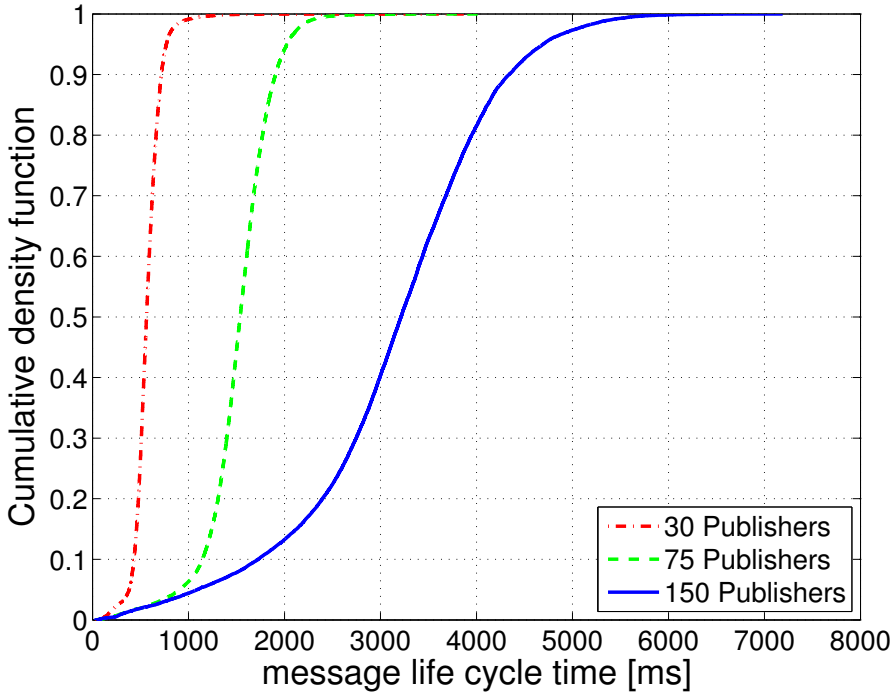



**Figure 6.12: Many publishers (WMS):**Message life cycle time for various messaging load, resp. 30, 75, and 150 publishers.

total delivery time remains reasonable (99% of messages are delivered under 5.4 seconds).

| | 30 Publishers | 75 Publishers | 150 Publishers |
|---|---|---|---|
| Min | 29 ms | 39 ms | 31 ms |
| Max | 3970 ms | 4043 ms | 7189 ms |
| Mean | 570 ms | 1520 ms | 3133 ms |
| q80% | 663 ms | 1771 ms | 3950 ms |
| q95% | 770 ms | 2027 ms | 4695 ms |
| q99% | 972 ms | 2284 ms | 5388 ms |

**Table 6.5: Many publishers (WMS):** Various statistics for delivery times from many publishers to a single client.

**Message Payload Size.** An important factor to take into consideration is the amount of data transmitted with each message. While tiny payloads are suited for battery-powered sensor nodes, large packet sizes are usual for less constrained sensors, as for example Wi-Fi or Ethernet Webcams. In this experiment, we evaluate the influence of the payload upon end-to-end latency.

In this experiment, three publishers send data to a sink at random times, and a single client subscribes to every message received by the sink. Three runs are performed for various payload size, where each message has a payload of respectively 30, 60 and 120 Kilobytes. Key statistics are given in Table 6.6, and the cumulative density function of the message life-cycle time is shown in Figure 6.13

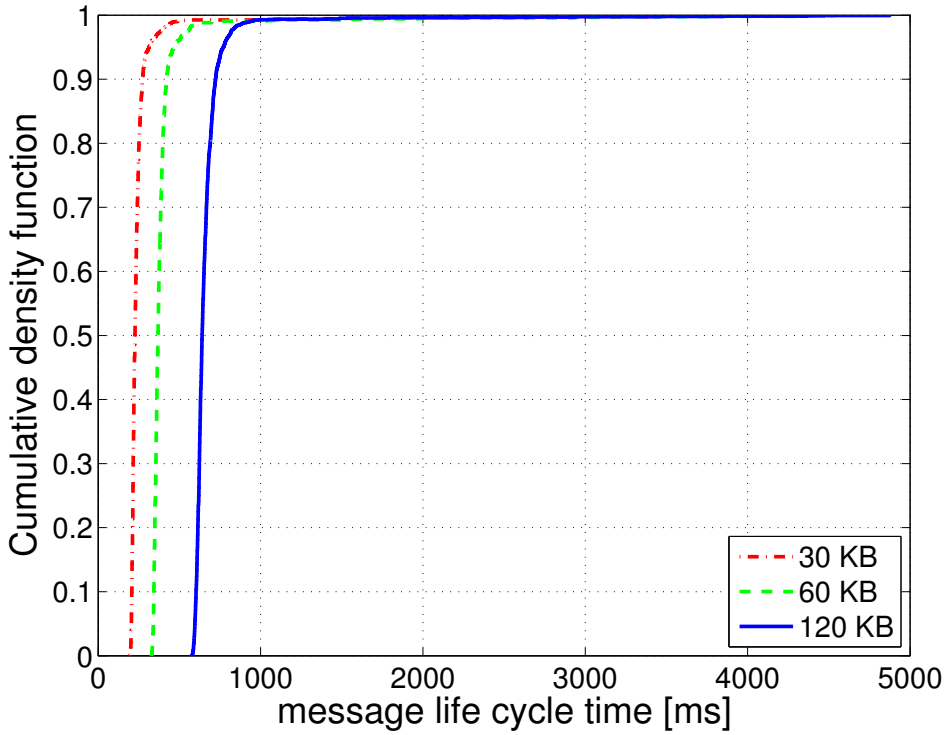



**Figure 6.13:** Message life cycle time for various payload sizes, resp. 30, 60, and 120 KB in each message.

| | 30 KB | 60 KB | 120 KB |
|---|---|---|---|
| Min | 193 ms | 328 ms | 575 ms |
| Max | 4420 ms | 4537 ms | 4875 ms |
| Mean | 253 ms | 399 ms | 668 ms |
| q80% | 252 ms | 398 ms | 690 ms |
| q95% | 314 ms | 471 ms | 772 ms |
| q99% | 477 ms | 752 ms | 933 ms |

**Table 6.6:** Message life cycle time for three publishers to a single client, and various message

With a payload of 30 KB, the message life cycle is short (under 477 ms for 99% of all messages and less than 260 ms in average). Even for large messages of 120 KB over 99% of messages are sent within a second.

### 6.2.2 Discussion: Web Messaging

In this section, we have measured the performance of HTTP push when used as messaging protocol to implement a low-latency notification and streaming mechanism. We do not explicitly address the low-power radio communication used in NEDs as it is out of scope for this thesis. We are only concerned with how to collect sensor data *once it is on the Web* – that is a gateway has received the data from device using various radio protocols and then needs

to transmit it over the Web. It is important to note that we tested the system with a load a few degrees of magnitude higher than usually required by long-lived WSN deployments for environmental monitoring described in litterature, as we tested our system with a few hundreds of messages per second, while WSN nodes report data usually less than once per minute (one message every 5 minutes per device is common).

When a few clients subscribe to the events generated by a gateway, the subscription process is very fast and the time required to notify all client is reasonably low (95% of messages are sent to 200 concurrent subscribers in less than 0,5 s). When 600 events had to be delivered to the same subscriber over 95% of them arrived within 2 seconds. Even with large messages, the delivery times remains less than a second for a few messages per second. In most cases, the performance scales linearly with the number of subscribers, publishers or events, but tends to become unstable and large variations in delivery time occur as the load increases. However, the load remains largely acceptable for a few hundreds messages per second delivered to a few hundreds of subscribers.

Higher throughput could be easily obtained by optimizing the implementation of the messaging broker. One of the major bottlenecks in our implementation of a WMS broker on gateways is that all messages are stored locally into an embedded database (in our implementation we used db4o [10]). Via measurements performed in [101, 184] we have shown that the latency of delivery for all messages is 3-4 times higher when messages are stored in a database, instead of simply being forwarded as they come without any persistence. Other results show that more subscriptions have a smaller impact on the scalability than publishers, and the internal total load (msg/s) that can be handled by the broker is the bottleneck of the system. Besides, thanks to the modular approach used in our gateway software, the actual database implementation is decoupled from the actual application, as it is implemented inside an OSGi bundle. It could be easily replaced by an optimized storage solution or completely outsourced to a dedicated message broker solution that will only handle distribution of many messages to many users, as will be shown in the next chapter.

Also, in other experiments not shown here we have shown that the performance of Restlet degrades significantly when handling more than 500 requests per second. Using a faster NIO-based connector (Grizzly or Jetty), much higher throughput could be attained. Nevertheless, the performance we obtained is sufficient for many small scale NED applications where a few hundreds of devices posting one message per second each with a few parameters (that is a total payload of 20-30 bytes), given that our tests were performed with messages that have 1KB of payload. In particular, the performance we obtained is certainly sufficient for many home automation scenarios where all the devices in a house or a building could be connected together in a unique application.

According to the increasing interest in real-time communication and messaging over the Web, future Web standards are planned to support elaborate and efficient push-based Web communication. This is already the case with HTML 5 Web Sockets API [151], but it is very

likely that efficient libraries, servers, and brokers that support scalable messaging will flourish. The combination of standard Web-based bidirectional messaging and their implementations for embedded devices that rely heavily on cross-layer optimization [115], will make it possible the develop more open and efficient NED applications entirely built upon Web standards. As more efficient implementations and solutions, along with optimized protocols for the Web (e.g., SPDY [51]) will likely increase the performance of Web messaging in large-scale application with thousands of concurrent users and messages per second to be handled.

## 6.3 Part II: Stream Processing Engines

*Database management systems (DBMS)* have become a central component in almost any Web application today to help managing huge amounts of persistent data. Complex queries expressed in query languages such as SQL are used to retrieve and process stored data. These types of queries are executed once and a result is returned, and work well for rather static data (read queries occur much more often than write/update queries). *Data Stream Management Systems (DSMS)* emerged a decade ago from the increasing need to process large amounts of continuous streams of data in many disciplines from remote sensing, to finance, to network security monitoring [74, 95]. A *data stream* is a potentially infinite sequence of data items (called *tuples*) from any source, be it sensors or other applications.

Aurora [65] and Borealis [64] were among the earliest stream processing applications and defined a variation of relational algebra extended with various operators for processing continuous data streams (e.g., aggregation and grouping). Although, these systems did not define a query language, they used the *boxes and arrows* paradigm known from workflow systems to specify the processing operations over continuous data. Another early system was TelegraphCQ [97] which explored distributed queries over data streams and also handled data loss. TelegraphCQ already provided a basic query language, but queries were only a definition of the data flow using basic building blocks (i.e., the query operators). The STREAM system [187] was among the first ones that provided a query language that abstracted more from the basic query operators. Early research lacked a common ground where the stream semantics could be defined without ambiguity, and most of them did not specify the semantics of the query language formally. The Continuous Query Language (CQL) [72] formally specified the semantics of windowing operations (i.e., only process "recent" chunks of incoming data as an approximation), using a SQL-like query language which supports both relations and streams. It supports selection, projection, join and aggregation operations over streams using windowing functionality (sliding windows). Furthermore, stream-to-relation and relation-to-stream operators were defined. Using CQL, simple declarative queries can be used, and the following example shows a query to report the light reading of several sensors every second:

```
1 SELECT Rstream (id, sample1(light) as light)
```

```
FROM Sensors [RANGE 5 SECONDS SLIDE 5 SECONDS]
GROUP BY id
```

Originally, WSN applications were based on a sense-store-process workflow (see Figure 2.2). This behavior was usually hardcoded at the node level, therefore it was difficult to reconfigure the application once a WSN has been deployed [247]. This rigidity led to the development of a new paradigm that enable interaction with sensor networks in a data-centric way, rather than device-centric. Among them, Cougar [249] and TinyDB [178] were early distributed query processing systems that enable data retrieval from a sensor network via simple queries and provided powerful runtime optimizations to make such operations more efficient. With TinyDB, a declarative SQL-like query interface could be used to specify the data collected by a sensor network deployment and various parameters could be configured, such as sampling rates or windowing operations. Furthermore, it was also possible to define event-based queries that are useful for scientific sensor networks where various physical properties are to be monitored (e.g., temperature, light or vibration thresholds). We illustrate this process with the example used in the original paper [177], which is a query to report the average light and temperature level at sensors nearby a bird nest where a bird has been detected:

```
ON EVENT bird-detect(loc):
SELECT AVG(light), AVG(temp), event.loc
FROM sensors AS s
WHERE dist(s.loc, event.loc) < 10m
SAMPLE INTERVAL 2 s FOR 30 s
```

Although TinyDB provided a useful, higher-level interface for querying sensor networks, it focused mainly on processing raw sensor data. Even when using TinyDB, it remained difficult for non-technical users to develop and deploy, analyze and interpret the data collected, and share it with the world.

Semantic Streams is a framework to facilitate the semantic interpretation of sensor data, which is modeled as asynchronous flows of events [242]. Queries are defined using logical predicates and represent first-order logic descriptions of event streams. A simpler declarative language based on Prolog is used to describe and compose inferences over sensor data, but also to specify various quality of service parameters (e.g., confidence interval, total latency or energy consumption) for the result. The benefits of the framework are the abstraction it provides from low-level distributed programming and that it makes the output of sensor networks more understandable to non-experts. Semantic Streams go one level higher than *TinyDB* because they provide a semantic interpretation of the sensor data. However, the framework lacks flexibility, because all sensor data must be collected on a central server where all the processing took place. A unique sink is a limitation and is very different from the approach taken by *TinyDB* where each sensor node can gather sensor data (also from other nodes) and execute queries on it.

### 6.3.1 Query Languages and Data Models

With the multiplication of commercial and open source stream processing engines such as Streambase [54] or Esper [12], it was expected that a standard query language for streams would be agreed upon. Although many of the languages used by these engines are similar to CQL, no single language stands out as a standard language for stream processing. Various applications had very different requirements and no single model that works for all these applications has been found.

Another approach to implement a data stream processing application is to extend the classic messaging system to allow event-detection or continuous queries over data streams. The simplest way to do this is to specify a filter over messages for each subscription. Tian et al. [220] presented such a publish/subscribe system based on XML, which allows to specify an XPath expression along with the subscription, to filter out message that do not match the expression. This mechanism provides only little more functionality than traditional messaging systems as it lacks support for complex events. A system with slightly more expressive power is Cayuga [104] because it supports not only conditions on subscriptions, but also the union of such conditional streams. PADRES [173] proposed a mechanism to detect events where users could register subscriptions described using predicates, and complex patterns could be detected by combining various subscriptions.

In summary, there are three main categories one can consider. First, classic stream processing application that support various query operators over data streams, that support windowing or aggregation operations (among many others) defined using predicated or declarative, SQL-like languages. Second, there are complex event processing system, which are more focused on detecting specific event types or temporal patterns. Third, pub/sub systems allow various users to subscribe to raw or processed data using various parameters.

### 6.3.2 Web-based Querying

Although the various query processing models and languages briefly presented in the previous sections allow to implement elaborate data stream processing applications, they remain overly complex to use for the average technical user. Besides, the integration between various data sources and applications remains a tedious procedure. The Web of Things will benefit greatly from stream processing capabilities, which will require the Web to support various stream processing primitives that can be expressed in a RESTful manner. This will significantly lower the access barrier for users to work with real-time stream directly using HTTP and easily integrate streaming data and operations into mashups, using simply Javascript. We survey in this section existing methods for exposing query primitives on the Web.

The Google Data Protocol (GData)[7] is a RESTful protocol proposed by Google to read and write information on the Web, in particular is used to access the various online services offered by Google (such as Blogger, Google Docs, Analytics, etc.). Data is stored as a collection of items represented using the Atom syndication format and HTTP is used to handle communication. GData extends AtomPub for processing queries, authentication, and batch requests. In addition to categories, GData supports full-text queries, author searches, range queries on the *updated* and *published* timestamps. This is sufficient to retrieve stored data using logical expressions to filter the results, however, it is not meant for processing complex queries over streaming data. Yahoo Pipes [61] is a visually oriented tool that allows users to connect various feed-processing modules and aggregate or filter feed data (or other data providers which are turned into feeds) to generate one result feed. Unfortunately, Pipes only work at the feed level and not on the collections behind them. *Yahoo Query Language (YQL)* also supports various data inputs (including feeds), but the output is YQL-specific XML or JSON, therefore does not provide access to various data sources through a uniform data model (such as ATOM/feeds).

OpenSearch [33] is a widely recognized approach that offers a standardized API for search engines. The interesting fact is that OpenSearch uses feeds with extensions as the result format. For the query format, OpenSearch defines XML formats (one for describing the interface, and one for the query itself), as well as URI templates. The main limitation of OpenSearch is its focus on full-text search only. As discussed in Section 5.3 the utility of only keyword-based search is limited as more and more real-world objects will also become part of the Web. Feeds are a popular way to distribute timely information on the Web and they can be considered as services that represent a view on a collection as they expose only parts of the collection in a specific way. This metaphor is useful as it can serve as the fundamental meta-model for a variety of services offered by sensors, as proposed in [246]. Feed Item Query Language (FIQL) [191] also specifies a query syntax for feeds, but defines a more general set of operators, datatypes, and values which can be used for queries. FIQL also specifies how a feed provider can advertise the query capabilities of the underlying collection in the feed itself using a *query interface*. Although FIQL is an interesting approach, the initial and expired draft has never been updated, and no known implementations exist.

Feeds are a powerful and highly scalable mechanism to distribute efficiently timely data, as a simple document that is updated a few times a day needs to be distributed. Things become more complex on the server side when complex feeds have to be created on the fly for every specific user (for example subscribing to all the new bookmarks tagged with a specific keyword on the social service del.icio.us[8]). According to [107], the Web model with feeds does not fit the sensor data streams abstraction, nor do existing streaming protocols such as multimedia streaming protocols. In many scenarios, users want to access historical data in both a pull and push fashion. The paper proposes *Stream Feeds* which represent sensor streams by an

[7] See: http://code.google.com/apis/gdata/
[8] Online: http://del.icio.us

URL. Streams can be accessed using a RESTful (or SOAP) interface over HTTP. A HTTP `GET` request can be used to get historical data from a stream. To subscribe to streams a HTTP `PUT` request is used (which can contain filters on the stream data).

### 6.3.3 Stream Processing for the Web of Things

How to expose sensor data streams on the Web remains an open research question– should one use a feed model or rather a pub/sub one? Or a combination of both? Besides, how to advertise the Web querying constructs supported over each streams using standard stream processing constructions (filtering, aggregation, etc.)?

Early approaches in data stream processing were mainly considering homogeneous data sources, therefore interoperability among different devices and deployments has been only marginally addressed. As the typical size of NED applications will increase, so will the need to build scalable infrastructures to integrate different systems (see Section 2.3). Indeed, as more and more sensors will be interconnected, tremendous network effects could be obtained if aggregating, filtering, analyzing, and republishing heterogeneous sensor data in real-time could be done in a simple and universal way. To meet the scalability requirements for future Internet-scale sensor networks, loose coupling between actors is essential to ensure sufficient robustness and flexibility. When applied to the Web of Things vision, where heterogeneous devices come and go, it becomes essential to integrate NEDs into stream processing applications using Web standards.

A clean standard to define the connection between devices, the data and notification they produce, and the semantics of the connection between users and devices is clearly missing. This makes interpreting the data a client receives more difficult which results in more complexity on the client side. Extending current solutions for stream processing in such a way that data streams can be manipulated using the well-known and widely supported Web standards such as HTTP, XML, or Atom without compromising the expressive power of the query capabilities of these applications would allow to lower the barrier of entry to collect, use, share, and store sensor data. The ATOM feed model – along with its extensions and basic data types – is an efficient metaphor to access collections of time-ordered (sensor) data. Although the basic feed model is limited to retrieval and filtering of stored data, it could be combined with a pub/sub system over the Web and mapped to various query mechanisms, from general-purpose query languages such as SQL, to more specialized ones for special types of data streams, such as sensor network query languages [127] or more general stream processing languages such as CQL [72].

One can observe that the typical tradeoff between scalability and query expressiveness present in stream processing application remains in the Web world. However, as the recent developments in Web technologies have enabled to build efficient and scalable publish/subscribe systems, we suggest that a Web-based pub/sub model could be used to connect sensor networks with applications. On top of this *raw* pub/sub substrate, one needs to an expressive

query environment suited for embedded devices and exposed over the Web. This leads us to formulate the following requirements for a Web-based query model that can handle both the static data used in relational databases and real-time streaming data used in stream processing applications:

- **Windowing support.** The continuous queries used in stream processing heavily operate on time windows (see more in [73]), therefore a native Web pattern to model and manipulate time windows is essential. Windows are the very basic mechanism that allows to introduce more advanced query functionality (aggregation, grouping) afterwards.

- **Opaque queries.** Declarative queries describe *what* has to be done, not *how*. This allows to decouple the query from its execution, which can take place remotely on an external, specialized stream processing engine optimized for specific data formats and processing operations. Using a Web-level construct for queries, rather than specific queries bound to the implementation of the processing engine, allows a looser coupling in the systems which improves the flexibility and scalability.

- **Static data support.** Because many Web of Things applications might use both real-time data and stored data, a common model to access both will be necessary to simplify the design process. Because query languages used for relational databases are not compatible with the time windowing constructs used in stream processing, a common ground is needed for combining static data with streaming data using the same operators.

- **Query Capability Advertisement.** Another desirable feature is the ability to describe the various functions and primitives supported by each node (similarly to OpenSearch [33]). Some modules support only simple filtering over tuples, while other can support more complex window-based processing and a method to describe *in the stream itself* what it supports would facilitate ad-hoc use of feeds and their functions.

In a data-centric Web of Things, typical applications are concerned with collecting and processing real-time sensor data streams. However, such applications often also have to handle historical data and metadata about the formats, therefore it is important that the language allows to combine streams with relational data stored in a database. How to RESTify a query language in a way that combines both streaming and historical data remains an open problem. Unlike typical HTTP requests, a continuous query does not return a result, but is kept running in memory. Because various stream processing engines have been designed for various applications in mind, the question of how much should be "standardized" as a query language on top of HTTP becomes essential. In the following section, we build upon the discussions in this section to design a framework to develop end-to-end applications for the Web of Things that supports both static and real-time data.

## 6.4 WISSPR – Web-based Stream Processing

Processing the huge amounts of data generated on the Web of Things will require to easily integrate heterogeneous data sources into stream processing applications. While research in data streams processing has considerably progressed in the last decade, a standard method to simply share and integrate data streams from various sources still remains a significant challenge [219]. Recent solutions that tackled this problem in the context of sensor data do not cope with the scalability requirements of a participatory infrastructure, as these solutions are based on tightly-coupled distributed architectures. Given the lack of flexible standards to describe and transmit streaming data over the Web, integration of data from heterogenous sensors remains a complex process that requires much attention. This limits the development and consumer adoption of stream processing applications outside of industrial or business contexts. A lower access barrier to use stream processing systems and integrate sensor data streams on the Web is a necessary component for building interactive and event driven applications on the Web of Things.

In this section, we address the limitations of existing stream processing solutions by exposing the functions they offer over the Web. Little is known about which data model and query language would be optimal to expose streaming sensor data over the Web, therefore we discuss Web standards can be used to model real-time data streams and build data processing services upon them. This implies both *what* has to be exposed (actual raw data streams, only high level aggregates, or both) and *how* to expose it (query languages, push/pull, etc.).

In collaboration with Oliver Senn during his master's thesis [208] we laid the basis for *Wisspr* (Web Infrastructure for Sensor Streams PRocessing), which is a Web-based framework built upon a pub/sub system to facilitate the development of event-driven and real-time processing applications for the Web of Things. Our contribution is to bring sensor data into the Web in its true form – as real-time data streams – and make it easy to process and share that data with many users in a timely manner. Even though the Web has not been designed to cope with sensor streams, we show that HTTP is sufficient for building applications that can collect, process, share, and store hundreds of sensor readings per second, with sub-second end-to-end latency.

The novelty of our approach resides in our usage of HTTP as application protocol, which makes ad-hoc integration with the Web straightforward. This contrasts with existing solutions that reduced the role of HTTP to a mere data transport protocol to implement a custom application layer on top. By using HTTP as an application protocol, our solution minimizes the coupling between components which makes it easy to develop more flexible and scalable distributed applications.

### 6.4.1 General Design Principles

Programming sensor networks is challenging because devices are small and crash-prone, and their limited resources must be managed carefully. In particular, energy consumption and radio transmission must be optimized to cope with the transient nature of sensor networks where connectivity is often unpredictable. Despite the different application-specific features and hardware platforms, sensor network applications share a common goal: periodic delivery of data collected by different sensors towards a common place to be processed, stored, and acted upon.

Pub/sub systems have gained in popularity in many domains as they allow more scalable and flexible distributed applications by decoupling data producers and consumers through intermediary message brokers. Industrial and open source implementations can scale to millions of registered subscriptions and very high event rates, however they have limited expressive power beyond filtering content using simple logical expressions. In contrast, full-fledged stream and event processing systems have powerful query languages that allow to express more complex filtering than pub/sub systems; however, this limits their scalability with the number of subscriptions and number of queries that can be evaluated simultaneously. Because of this fundamental tradeoff, the integration of heterogenous data sources with powerful and expressive stream processing is difficult to do in a sufficiently loosely coupled manner to cope with the dynamic and ad-hoc nature of future Internet-scale distributed sensing applications.

Efficient messaging system can be implemented on low-power sensor/actuator devices and operate over bandwidth-constrained radio communication. In this case, the pub/sub paradigm is leveraged to simplify the integration of sensor networks with other distributed applications [155]. Comet provides low traffic overheads and great reactivity, while polling scales better on the client side [91]. Classic Web servers allocate a thread per incoming connection TCP connection, which is fine for request/response interactions, but does not scale with concurrent clients when using Comet. Embedded Web servers such as SMEWS [115] can serve Comet directly from common sensor nodes as it uses a pool of events instead of threads. As software for embedded devices are usually based on event-driven architectures, this model fits well with event notification mechanisms which allows efficient support for Web push techniques directly on sensor nodes.

A Web-based pub/sub system with an elaborate subscription mechanism that allows complex streaming data processing would provide an efficient abstraction for sensor streams as a first class citizen on the Web. Although HTTP was not designed for real-time stream processing, we have shown promising results when using Web standards to interact with distributed sensors and actuators [139, 227]. The loss in raw performance and latency due to verbose HTTP requests is compensated by allowing sensor networks to be exposed in an easily accessible and universal way. Besides, thanks to the many advantages offered by Web standards such as transparent proxies, declarative Web-based queries can be mapped to the specialized processing features of sensor networks, therefore one can still take advantage of the optimizations and advanced processing implemented within sensor networks and other stream processing systems. We

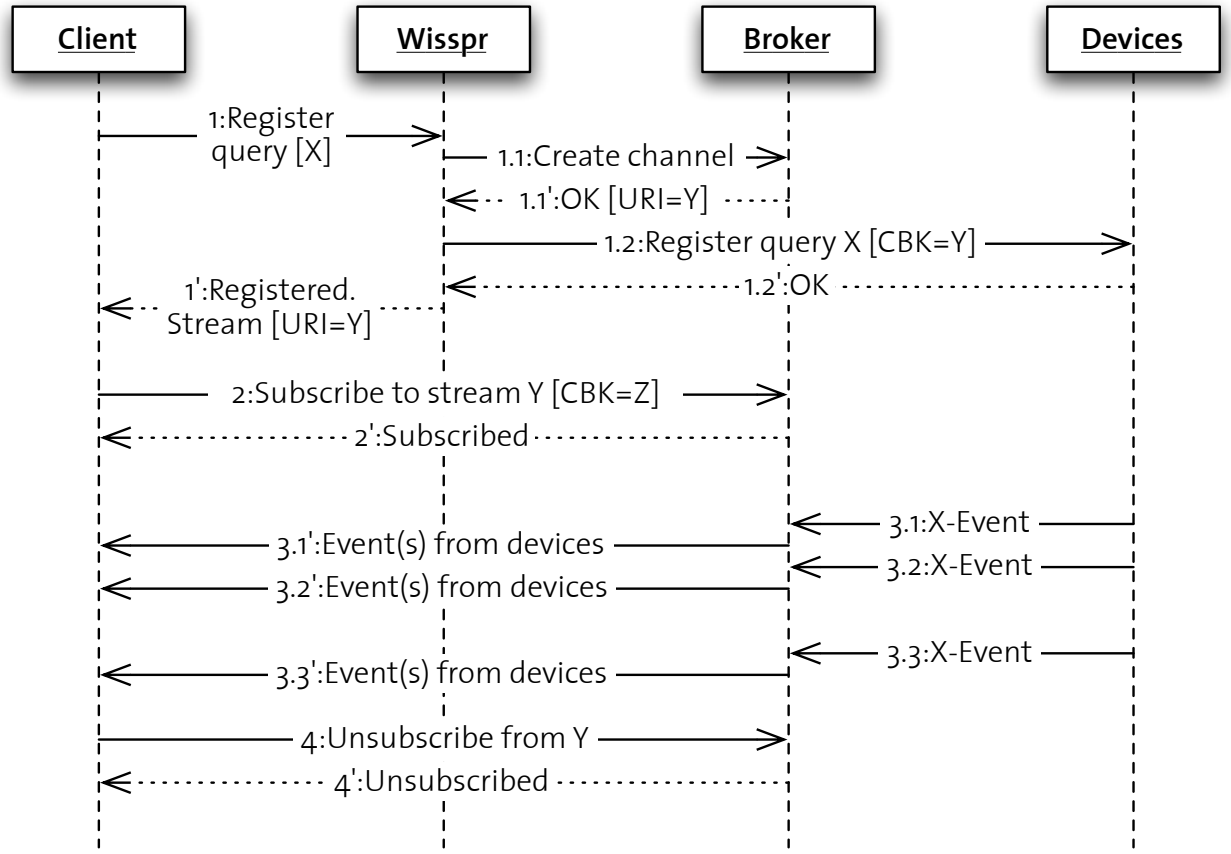



**Figure 6.14: Complete lifecycle of a Web stream query.** A Client sends a query registration request with the query [X] as parameter to a Wisspr query processor node (1). The node creates a channel on pub/sub broker where the stream will be available (1.1) and receives the URI of the stream (Y) in response (1.1'). The node then forwards the query to the concerned devices (1.2) with the channel URI (Y) as callback [CBK] URI (where the devices will post events that match the query using Web hooks), and finally the URI (Y) of the resulting stream is returned to the client (1'). The client then registers to channel Y on the broker (2), and all the events posted from devices that match query X (called *X-events*, 3.1, 3.2, 3.3), will be forwarded to the client by the pub/sub broker using the callback URI (Z) he registered with (3.1',3.2', 3.3'), until the client unsubscribes from the stream (4).

present how full-featured stream processing engines and scalable messaging systems can be integrated into Wisspr, to leverage both the expressiveness and performance of such systems. First, in Section 6.4.2 we describe how the stream model can be adapted using Web paradigms by describing how data can be exposed and encoded. Second, in Section 6.4.3 we describe the overall system architecture of Wisspr built upon the Web stream model we propose.

### 6.4.2 Web Streams

To simplify the development of sensor network applications, data-centric approaches to retrieve specific data using declarative SQL-like queries have been proposed. Considering a

network of sensors as a distributed database allowing queries over the data offers various advantages, as seen in TinyDB [178]. Sensor data is inherently stream-oriented, as it is a potentially unbounded sequence of events or sensor readings. Although streaming abstraction has not been much used in sensor networks, it has become increasingly popular in other domains such as finance. We now describe the *Web stream* abstraction, which is a first class Web citizen, therefore can directly benefit from security, authentication, or caching mechanism of the Web. Moreover, Web streams can be crawled and indexed by search engines, shared with friends or social networks and bookmarked just like any other Web resource.

### Data Stream Abstraction

Sensor data on the Web of Things can be transferred as a sequence of Web messages. Each message contains is a *tuple* of data sent from the device, that can be either sensor readings or other internal properties. Each device has therefore a certain number of data fields, each having a name and a type (string, integer, float, etc.). Devices manufacturers and programmers know in advance the nature of data measured by the device (units, type, range, etc.), therefore a schema could be used to define the structure of data sent by devices. This way, devices can send just a subset of all their data (as the most recent sensor reading every time a message is sent), which can be matched to the schema for interpreting the data. For example, a sensor node (called D1) could send its light and temperature readings as the following JSON object:

```
1 {"Device ID":"D1", "data":{"light":4,"temp":22}}
```

The semantic interpretation of this message requires a schema to understand the data fields. JSON allows to send data without necessary a schema (the XSD equivalent for JSON) to specify the structure of the message, therefore the client needs a mechanism to "understand" what `light` and `temp` refers to. CSV or binary encoding could also be used in case the schema is predefined and does not change. The device identifier (D1) could also be the device URI, which would allow any client receiving the message above to retrieve the schema from the root page of that device.

A more complex example is shown in Figure 6.15, where a stream is created and contains the aggregate of all light and temperature readings from three devices. The same stream could also consist of the individual messages as they come from the three devices, but the stream should accommodate both.

Data streams are created on demand, that is a user will request specific data and will receive it. First of all, one specifies the device one wants to use (D1-3) using their URI:

```
1 POST /streams
2 Host: wisspr.net
3 devices=D1,D2,D3
```

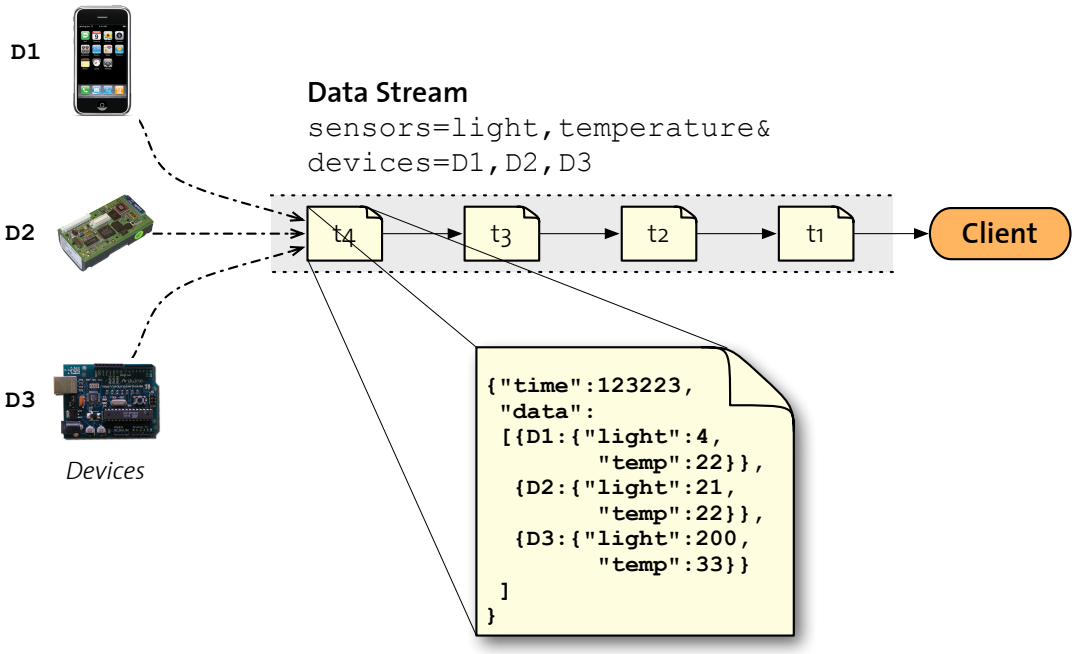



**Figure 6.15:** Data stream which contains light and temperature measurements from three devices (`D1, D2, D3`).

or parametrically, by using some keywords or other attributes that identify a set of devices:

```
POST /streams
Host: wisspr.net
devices={spots}
```

To simplify the design and usage of Wisspr, we suggest to use typed streams. This means that each tuple in a given stream can be described by the same schema, as every tuple has the same data fields. In turn, typed streams facilitate the automatic parsing and understanding of data with little effort, which would be necessary for querying and storing high volumes of data. Therefore, each stream should have a basic description available in a machine-readable format that allows to attach semantics to data fields of a tuple.

### Sampling Frequency

Different applications might require to access the same data source with different granularity in the temporal domain. For example, an application that needs to detect a fire in a building might sample a temperature sensor every second. An application that automatically regulates heating based on current temperature will only need an update once per minute, and an application that only needs to log the temperature in a potato field over a year for statistical purpose will only require an update per hour. The update frequency can be specified using as a parameter

of the subscription request, and the sampling frequency of the stream is independent from the actual sampling frequency of the sensor.

Using the example shown in Figure 6.15, to limit the messages received to one message every five seconds, one can use the following request:

```
POST /streams
Host: wisspr.net
data=light,temperature&devices=D1,D2,D3&frequency=5s
```

or its equivalent in Hertz (`Hz`), which is 0.2 Hz:

```
POST /streams
Host: wisspr.net
data=light,temperature&devices=D1,D2,D3&frequency=0.2Hz
```

These commands will create a stream that contains all the data matching the query, and users can simply subscribe to this stream using WMS or PuSH. The URI of the stream will be returned in the response using the `201 Created` response code using the `Location` header.

### Filtering

More elaborate properties can be specified at subscription time using a filter that processes incoming raw data and only forwards messages that match a specific criteria. The filter is specified as a predicate that can operate over the various data fields, keywords, sensor data or frequency of various data sources, and evaluates as a boolean value. As an example, imagine a rule to select messages where the temperature in the house is above 20 degrees only if the window is closed and somebody is in the house. Using the data fields `temperature`, `light`, and `window` from a sunspot (`window` is a high-level composite function assumed to be provided by the sunspot using the accelerometer and gyroscope sensors) would be: $(temperature >= 20 \;\&\&\; light < 15) \;||\; window = CLOSED$. This request is expressed with the following HTTP request:

```
POST /streams
Host: wisspr.net
data=light,temperature,window&device=sunspot&frequency=2Hz&filter=(temperature
    >=20&&light<15)||window=CLOSED
```

Note that the last line is encoded by browsers as follows:

```
data=light%2Ctemperature%2Cwindow&device=sunspot&frequency=2Hz&filter=(
    temperature>%3D20%26%26light<15)||window%3DCLOSED
```

Sampling frequency and filters are both optional, and if these parameters are not specified in the subscription, by default all the messages from selected devices will be included in the stream, using the default sampling frequency of each device. If both a frequency and a filter are specified, then one cannot guarantee that a message will arrive at every tick, but only that the specified frequency will not be exceeded. This also means that regardless of how many messages match the filter, at most one per sampling period (the most recent or an aggregated version, depending on the aggregation request) will be included in the stream. The procedure is illustrated in Figure 6.16.

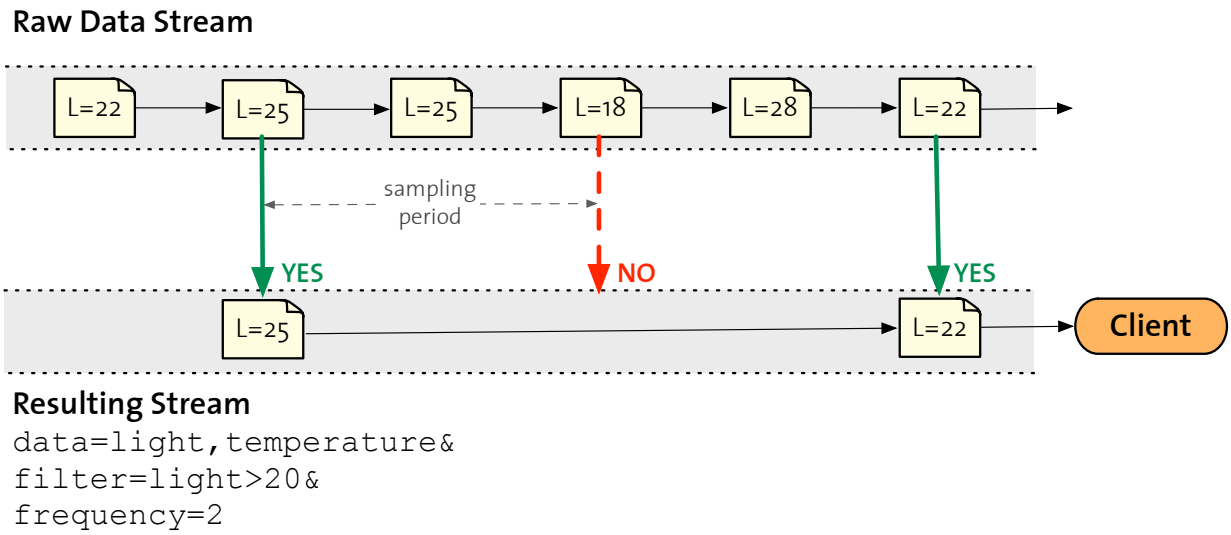



**Figure 6.16:** Messages contained in a data stream with specified frequency and filter: the most recent message is published if several messages match the filter ($L$ is the light sensor reading, which should be $L > 20$); no message published otherwise.

The combination of filter and frequency is especially relevant for applications where sensors are sampling at high frequencies to detect anomalies (for example an accelerometer sampled at 50Hz to detect a sharp onset), but only a fraction of those needs to be stored (for example once every 5 minutes). This is particularly useful to limit data rate when many messages match the filter.

### 6.4.3 Wisspr System Architecture

Different applications that use data from physical sensors present a wide variety of requirements. For simple use cases, easy configurability, installation and usage are vital. In more complex use cases where many devices and users interact, scalability, message latency, and robustness are often more important in order to ensure a certain level of performance. On one hand, a simple processing node might have only limited query processing capabilities or no local storage and are designed to handle only a few simultaneous connections. On the other hand, some processing nodes might be full-featured stream processing engines designed to process thousands of data points from many heterogeneous sources. The fabric of the Web of Things, must be able to accommodate both extremes and allow seamless and transparent interaction between them. The layered design of HTTP makes it easy to abstract the nature and physical location of the nodes behind a unified paradigm and we leverage this property to implement a scalable framework for distributed stream processing applications.

The core idea of Wissprs' architecture is to separate the overall functionality of applications into different modules that exchange data with each other using a common, Web-based format. Applications are then built by *wiring* (or *piping*) the different modules together to form complex processing chains that link raw data from devices to various processing, storage, or notification elements as shown in Figure 6.17. We have identified four classes of modules that are sufficient for building most types of sensor network applications:

- **Devices.** This module allows physical devices to be connected to our system. It offers information about and data from devices that are currently connected. Furthermore, it allows the messaging module to access the data produced by devices.
- **Messaging.** Client applications as well as the storage and query processing modules can compose data streams from the device sensor data. The management and the distribution of those streams is the responsibility of the messaging module.
- **Query processing.** This module allows to process the sensor data streams by running continuous queries over the streams. The query output streams can be then fed back into Wisspr and made available to client applications and other modules via the messaging module.
- **Storage.** Many use cases need to store sensor data, which is handled by the storage module that allows to persist data streams.

Modules are in fact simply connectors to different specialized processing units, and their role is mainly to serve as Web proxies that abstract the different functionalities of the actual units behind a common RESTful Web API. This way modules can interact transparently with each other over the Web, regardless of the specific implementation of each module. Besides, modules can be replaced at run time with little effect on the overall functionality, and no need to

reprogram the application. The connector nature of modules is important both for flexibility (e.g., one can use different storage engines) as well as for scalability (because the modules will often not be the bottleneck, but rather the engines behind them, therefor the implementation can run on a cluster or in the cloud). Moreover, the functionality of each module can be scaled from embedded computers to server farms, depending on the requirements at hand with little or no change to the overall application because the API remains identical (the actual performance of the module will of course depend on its implementation).

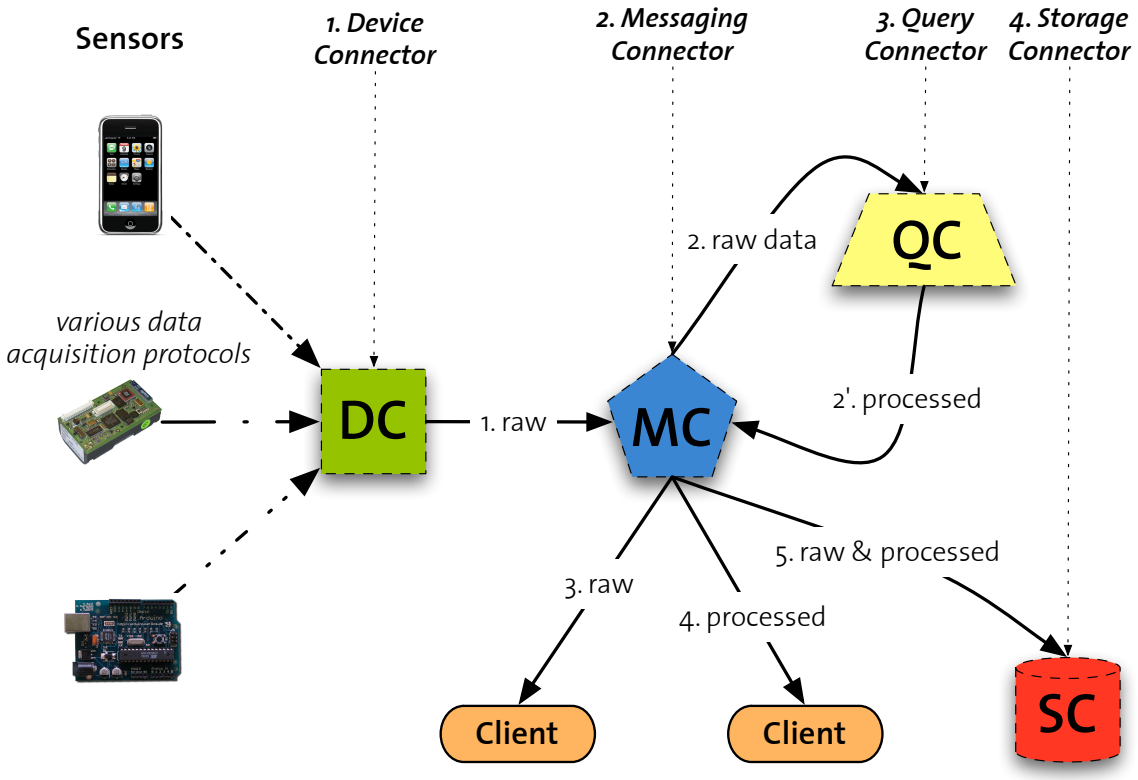



**Figure 6.17: Overall Wisspr Architecture.** Four types of modules used in Wisspr.

The possibility of distributing and accessing modules over the Web is a significant advantage offered by our architecture over traditional sensor sharing protocols. A common Web API would allow different organizations to implement, run, and control different parts of the system just the way they want. Companies that offer cloud storage (such as Amazon's SimpleDB) could run highly scalable and efficient storage modules, whereas companies with more modest requirement (or budget) might only use simpler, less performant storage solution that would be sufficient for their needs such as a Networked Attached storage (NAS).

Although, we propose HTTP as universal protocol to connect modules from different actors, other protocols can be used when all modules involved into a particular process are owned by the same actor or run in the same local network, or even on the same machine. As modules have been implemented in OSGi (also called *bundles* in OSGi terminology) when they run on

the same machine, OSGi declarative services – rather than local REST calls – can be used for inter-module communication which leads to significantly higher performance.

### Underlying Messaging System

A messaging system provides the *glue* to bind different the different nodes (processing, storage, clients, etc.) together via a common asynchronous communication infrastructure. The idea is that several message brokers are distributed all over the Web (in a similar fashion to DNS, where many organizations would operate their own message brokers for their devices). To ensure adaptability and reuse, we have used pubsubhubbub (PuSH) instead of WMS, because it is an existing protocol for Web messaging available off-the-shelf. Because WMS is a subset of PuSH, switching between these protocols is straightforward.

All device data and query result streams are available from message brokers via a uniform RESTful interface, which allows to easily publish and consume device data. Devices capable of sending HTTP requests can directly publish their data to the message broker, either to a queue that already exists and is predefined or by creating a new queue (or exchange) on one of the message brokers. The device has to be configured beforehand to know which message broker to use. The device can then simply send a `POST` message with the new data to the message broker which then forwards the data to all the subscribers of the queue. Devices that do not directly have HTTP support can connect to a gateway which acts as a small scale minimal broker that serves as proxy for an actual broker.

Because Wisspr relies on third-party stream/event processing engines, connectors are used to subscribe to streams and transform the common stream data format used throughout Wisspr into the native format used by the processing engines. Additionally, the connector is also responsible to re-publish the resulting output stream of the SPE to a message broker so that the query result stream can then be re-injected into Wisspr to be used further. The location where the resulting stream should be send to (which queue on which message broker) can be specified at the query registration time.

### Query Registration

Clients should be able to leverage the uniform interface of a RESTful API to register their queries in Wisspr. We consider two types of queries in Wisspr, first are simple queries that only perform a few simple operations (filtering via simple rules, e.g., JSR-94 [20]) that are implemented in the different connectors, and second are more complex rules that are run and processed by external stream processing applications (usually expressed in CQL or other engine-specific language). Minimal rules support the input streams, the query in the language of the engine, and the location where the output stream should be `POST`ed. Figure 6.18 shows how

a simple query is processed by a device connector. All applications that want to access the resulting stream, need to subscribe to the location of the output stream. The following cURL[9] subscribes to a device called `spot11` to receive its temperature and light sensor readings only if the temperature is above 20 (degrees Celsius, if unit not specified uses SI units by default):

```
user@host~$ curl -d "devices=http://devices.wisspr.net/spot11&data=temperature,
    light&filter=temperature>10" http://wisspr.net/streams
```

In practice, the following HTTP excerpt illustrates the request-response cycle generated by the previous command:

```
1 POST streams/
2 Host: wisspr.net
3 Content-Type: application/x-www-form-urlencoded
4 devices=http://devices.wisspr.net/spot11&data=temperature,light&filter=
      temperature>10
5
6 201 CREATED
7 Location: http://wisspr.net/streams/239048
8 Connection: close
```

The `Location` header is used the return the URI of the stream that has been created and will contain the query results. Afterwards, many clients only need to subscribe to the stream URI using a supported mechanism. For example, this command allows a pubsubhubbub client to subscribe to the stream:

```
user@host~$ curl -d "hub.mode=subscribe&hub.callback=http://example.org/
    callback_handler&hub.topic=&hub.verify=sync" http://wisspr.net/streams
    /239048
```

Afterwards, the stream will simply push JSON data to the client handler:

```
1 POST http://example.org/callback_handler
2 {"timestamp":1267370567, "name":"spot11", "temperature":25.5,"light":120}
3 ...
4 POST http://example.org/callback_handler
5 {"timestamp":1267371293, "name":"spot11", "temperature":10.2,"light":50}
```

Ideally, the connector can automatically determine where the resulting stream can be posted to and in which format (the sink endpoint URL, which can be a broker or directly an application endpoint if no other users need to subscribe to the stream) and return it to the query registration request using the `Location` HTTP header. Note that the format of the subscription is not given

[9] cURL is a command-line tool for transferring data using URL-centric protocols. Online: http://curl.haxx.se/

as a parameter, but is deducted by Wisspr which recognizes the `hub.*` as being the syntax used by the PuSH protocol. Alternatively, one could use a 1-step stream creation request, by giving a call-back URI as well as query parameter as follows, with optionally specifying the format (WMS by default defined using the `wms.*` syntax), as follows:

```
user@host~$ curl -d "devices=http://devices.wisspr.net/spot11&data=temperature,
    light&filter=temperature>10&wms.callback=http://example.org/callback_handler
    " http://wisspr.net/streams
```

This allows more flexibility for creating and using streams, but it also increases coupling. To minimize coupling clients should not need to know where the query is actually processed and in what underlying language the specific implementation requires because it uses a higher-level processing language. The actual federation of stream processing engines is an active research topic [90, 89], but it is out of scope of this project and will not be discussed further in this thesis.

Since all streams are identified by an URL and are published over HTTP, one can reuse existing mechanisms such as SSL and HTTP authentication mechanisms. Also, popular streams can be cached by intermediaries or distribution networks (see more about about messaging scalability in Section 6.4.5).

### 6.4.4 Module 1: Device Connector (DC)

The *device connector (DC)* module is essentially a simple gateway that uses various device drivers to connect sensors with the Web. In this respect, it is nothing more than the proxy described in Section 4.3 with a minimal messaging broker that supports a simple query-based subscription mechanism. It allows a few clients to subscribe to sensor data streams from the devices associated. In our implementation, users can specify a combination of data fields from the devices, a frequency and simple filter using rule, as defined in Section 6.4.2. A thread is created for each stream and it notifies subscribers using Web hooks. The scalability of this approach is very limited, but is already sufficient for a dozen of rules with hundreds of subscribers per rule with a standard desktop computer. Obviously, the scalability of this module depends on the machine it is actually implemented on.

As shown in Figure 6.18, the device connector is an augmented gateway that allows to subscribe to queries that will be pushed to a callback URI provided by the client at registration time (`[CBK-URI]`). Obviously, the callback URI must point to any Web server than can receive, understand, and act upon Web hook notifications posted by Wisspr.

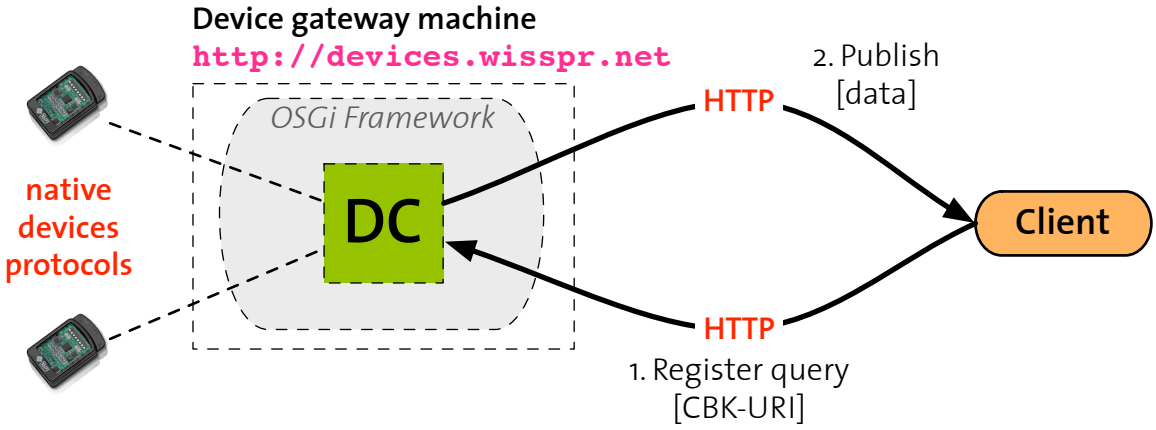



**Figure 6.18: Device connector (DC).** A gateway running on a single machine allows users to register simple queries on devices (1). The client callback URI `[CBK-URI]` where the resulting stream will be posted to (2), must be provided as parameter.

### 6.4.5 Module 2: Messaging Connector (MC)

The *messaging connector (MC)* module enables device data to be made available as Web streams in a more flexible and especially scalable manner. Using an uniform Web messaging system, any module can subscribe to the output data produced by any other module. Our implementation currently supports the RabbitMQ message broker with RabbitHub (a plugin for enabling PuSH support in RabbitMQ). Since the actual message broker is an external component of our system, other brokers could easily be substituted. External brokers could either use PuSH to enable Web-based access, or could support more optimized protocols such as AMQP or 0MQ for improved performance.

We illustrate our system in Figure 6.19. Users who want to receive certain data can request the creation of a data stream from the messaging connector module. The module then creates the data stream and will filter the data received from the device connector and will route the data matching the request to the message broker, who will then notify the user on the callback URL (with a HTTP `POST` request).

#### Messaging Scalability

In an open, Internet-scale Web of Things, many users could need to access various real-time data generated by many sensors, therefore massive output scalability is needed. In order to handle many data streams and many users, the messaging connector has to be scalable well beyond current requirements. More machines and processing power should be easily added to support higher throughput and more users with only slight increase in latency, which is acceptable since latency is not the critical bottleneck for the NED applications considered in this thesis.

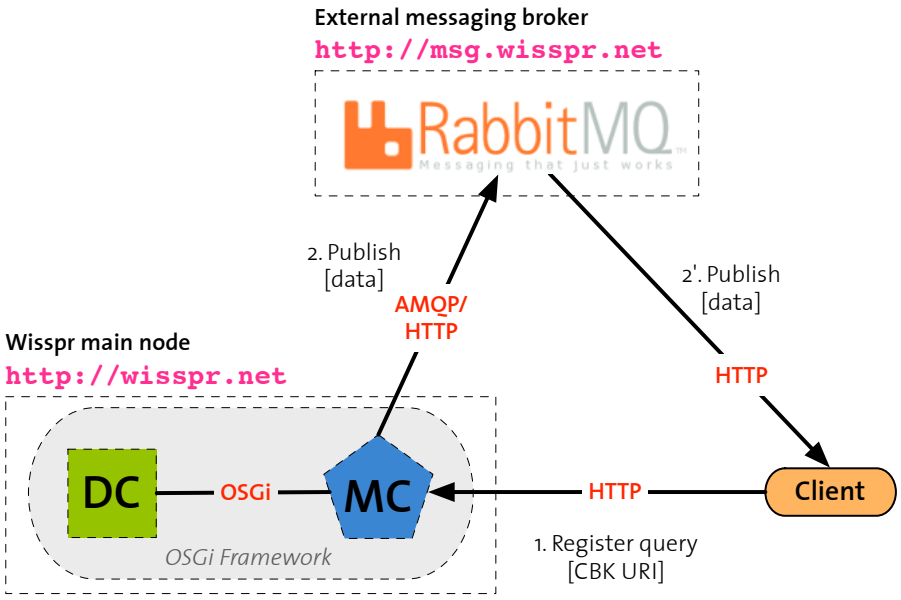



**Figure 6.19: Messaging connector (MC).** Clients can register a query with an HTTP `POST` request on a Wisspr machine with a messaging connector (1). The request is then forwarded to the DCs of the requested devices and all the messages corresponding to the request will be published to a pub/sub broker (2), which will forward them to the stream subscribers (2').

Even if processing and filtering can reduce the number of messages that will have to be sent to clients, situations where many clients will require customized content streaming will be commonplace, thus a mechanism to distribute lots of message quickly to many consumers is essential.

It is essential that both the components of our architecture and the communication mechanism between them scales well, and the loose coupling of a RESTful architecture can help in this respect. For obvious performance reasons, the message broker is running on a separate machine, but could easily also be running on a cluster of machines or in the cloud, as illustrated in Figure 6.20. The messaging broker implementation we have chosen (RabbitMQ) supports this off-the-shelf [41], and it can be easily replaced with alternatives as long as its Web API does not change. Obviously, the implementation of the various processing modules (for example filtering on data streams) in the whole system must also scale similarly.

### Example

To illustrate how requests and messaging works, Listing 6.4.7 shows a typical subscription HTTP request a user can issue on the messaging bundle to receive particular device data. The messaging connector module processes the request by creating a data stream with the temperature and light data from a sensor, with a frequency of two samples per second. After that, the user will get notifications (HTTP `POST` requests to the callback URI specified when

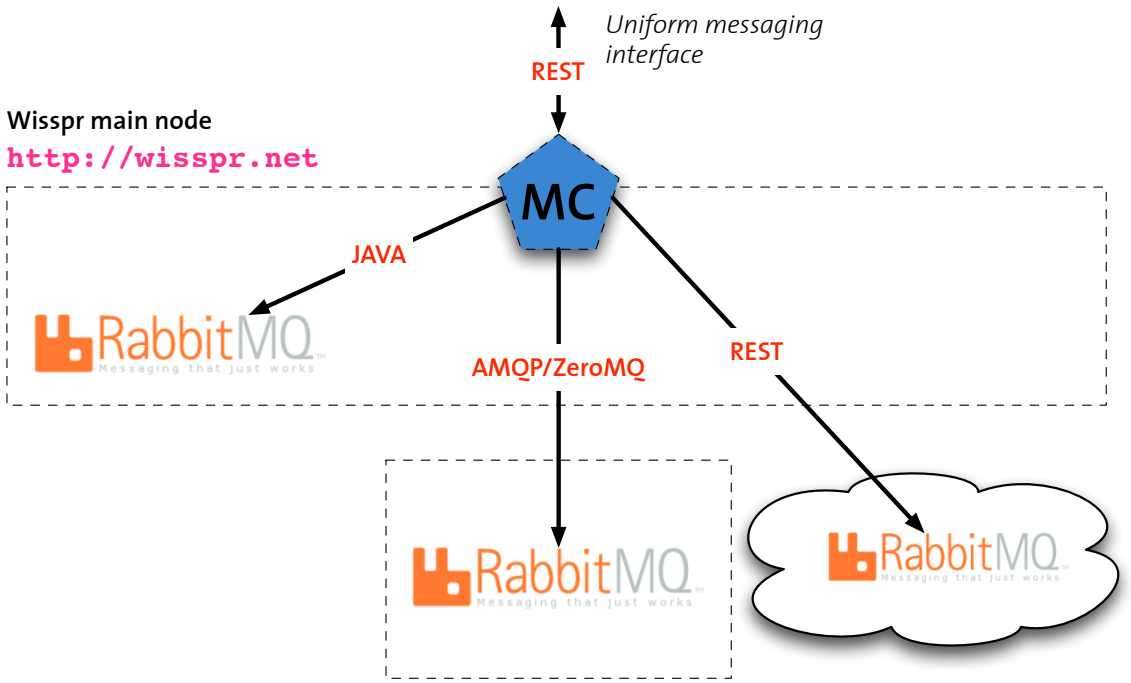



**Figure 6.20:** The *messaging connector (MC)* is a Web-level abstraction of any messaging broker. The messaging connector enables Web-based messaging to glue Wisspr components together regardless of the implementation of the pub/sub broker, which can range from a minimal broker running on a router to an industrial-scale solution running on a cluster or in the cloud.

registering the query) with the most recent data from the devices, at the frequency he specified (2 msg/s.). The filtering criteria specifies that data will be sent to the broker only if the temperature reading is above 19 and light lower than 200.

```
1 POST streams/ HTTP/1.1
2 Host: wisspr.net/
3 Content-Type: application/x-www-form-urlencoded
4   devices=http://devices.wisspr.net/spot11 &
5   data=temperature,light &
6   frequency=2 &
7   filter=light < 200 && temperature > 19 &
8   hub.callback=http://example.com/client_callback_handler
```

**Listing 6.1:** A complete subscription request with data filtering

### 6.4.6 Module 3: Query Processing Connector (QC)

The messaging connector is suited for small scale scenarios that require limited processing, that is simple filtering of raw data from a few devices. Many emerging applications will require much more elaborate and complex stream processing and data analysis capabilities over huge volumes of real-time data streams. As mentioned in Section 6.3, although commercial stream processing engines have been flourishing in the last decade, most of them operate in closed systems, therefore integrating heterogenous data sources is a tedious process.

To benefit from the ease of integration offered by our architecture, so that Web data streams can be easily used within different SPE, we developed a *query processing connector (QC)* module which enables the integration of such elaborate processing engines within Wisspr. As a proof of concept we have integrated the query connector module with Esper, which is a widely used open source SPE. Queries are written in CQL, which is the query language supported by Esper. The query will work on the data streams available in Wisspr using an internal binding that matches the variables used by Esper with the URI of the data streams used in Wisspr. One could send an aliasing command to the query connector module:

```
http://wisspr.net/streams/225 as streamA, http://wisspr.net/streams/228 as
   streamB
```

This creates the variable streamA and streamB that actually refer to two streams available in wisspr and that can be accessed in the query connector module (and the request sent to it). Then, we consider that each stream contains messages from devices which are all encoded using the following XML format:

```
<DEVICE>
        <ID>..</ID>
        <time>..</time>
        <sensors>
                <temperature>22</temperature>
                <light>200</light>
                ...
        </sensors>
</DEVICE>
```

One could then simply use the following query to ask for the average of all light sensors readings from devices streaming to channel `streamA` every 5 seconds, and the average temperature from `streamB` every 30 seconds:

```
SELECT avg(streamA.sensors.light), avg(streamB.sensors.temperature)
        FROM streamA.win:time(1 seconds), streamB.win:time(30 seconds)
```

This will result in a new stream where each message contains two fields (float) that can be pushed back in Wisspr to be further reused by other modules. This can be done as long as the schema of the messages are known in order to be able to create queries, and for this reason we suggested for each stream to adhere to a strict schema. Future work will include how to loosen this constraint to support schema-less streams (although a schema will be necessary to understand the semantics and content of those messages). Furthermore, an easy mechanism that does not require the aliasing

Unfortunately, no query language has been standardized so far for querying continuous data streams, therefore we cannot build an unified RESTful interface that would work with any SPE. For this reason, queries must be written in the language supported by the SPE, not in a generic Web-based query language, which would require the development of specific connectors for each engine and an elaborate, generic, and Web-based stream processing language with support for many common constructs (windowing, etc.).

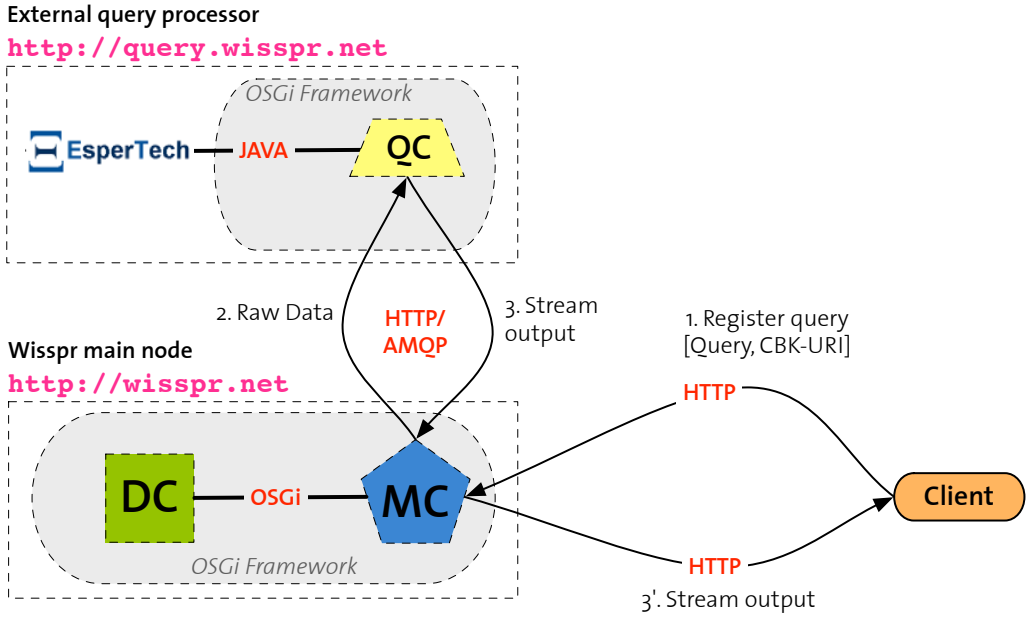



**Figure 6.21: Query Processing Connector (QC).** For elaborate applications, a dedicated stream processing engine provides much better performance and specific functions, therefore easy integration of Web stream with external engines is essential. Clients register a query and provide a call-back URI where the resulting stream should be posted to (1). The query is then registered into the SPE, and the required data streams are pushed into the SPE as well.

The process for registering a continuous query is very similar to the one to persist a certain data stream. What the user is conceptually doing is creating the query output data stream on the messaging connector. The process is depicted in Figure 6.21 (everything on one machine) and described here in more detail.

### 6.4.7 Module 4: Storage Connector (SC)

Many use cases require raw and/or processed data in Wisspr to be persisted for archival or analysis purposes. The *storage connector (SC) module* offers a Web-level abstraction for storing arbitrary Web streams with minimal effort. As illustrated in Figure 6.22, storage of an existing stream is done simply by subscribing the storage connector to it. This can be done either by `POST`ing a storage request directly to any given SC module (1a), along with the stream URI to store as parameter. Additional parameters can also be given to specify how and where the stream should be stored, along with access credentials and other properties. Because our stream data model has a well-defined structure, it can be easily mapped to a relational database, object or document database. The response to the request contains the URI to access the stored data.

Users create a stream as follows:

```
user@host:~$ curl -d "devices=http://devices.wisspr.net/spot31&data=temperature,
    light&filter=temperature>20" http://wisspr.net/streams/34
```

Users can then subscribe the storage connector to any stream in the system using the following request:

```
# Request 1a. to subscribe the storage connector to a stream:
user@host:~$ curl -d "datastreamURL=http://wisspr.net/streams/34" http://store.
    wisspr.net/
```

The storage connector also has a list of stores, each one associated with a particular storage configuration and having its own URI, that can be used as call-backs when manually subscribing the SC to a stream. The following command allows to create a new store on the storage connector:

```
user@host:~$ curl -d "database=mysql&username=john&pwd=doe&tableName=store31"
    http://store.wisspr.net/stores/31
```

The `database` parameter refers to an internal connector (using exactly the same principle as the drivers used in the DC as seen in Section 6.4.4), which handles the actual interaction with the storage solution (e.g., JDBC connector or Amazon Web Service client). Obviously, as with any Web application, developers of every SC can implement detailed APIs for creating and configuring elaborate storage scenarios that specify a specific storage strategy for each stream (for example how the database should store the data internally) or simply hardcode specific data pre-processing routines for each connector. However, as more coupling is introduced between a stream and the way it is stored, the less RESTful it becomes, therefore this topic is not elaborated here any further and left for future work. The only constraint is that each store must be able to handle any Web stream in Wisspr by acting as a callback for the subscription. That means, a storage request can also be sent to a stream directly (1b. in Figure 6.22), as follows:

```
user@host:~$ curl -i -d "callbackURI=http://store.wisspr.net/stores/31" http://
   wisspr.net/streams/34
```

This way, all the messages in the stream will be forwarded to the SC, which will persist them using the store configuration.

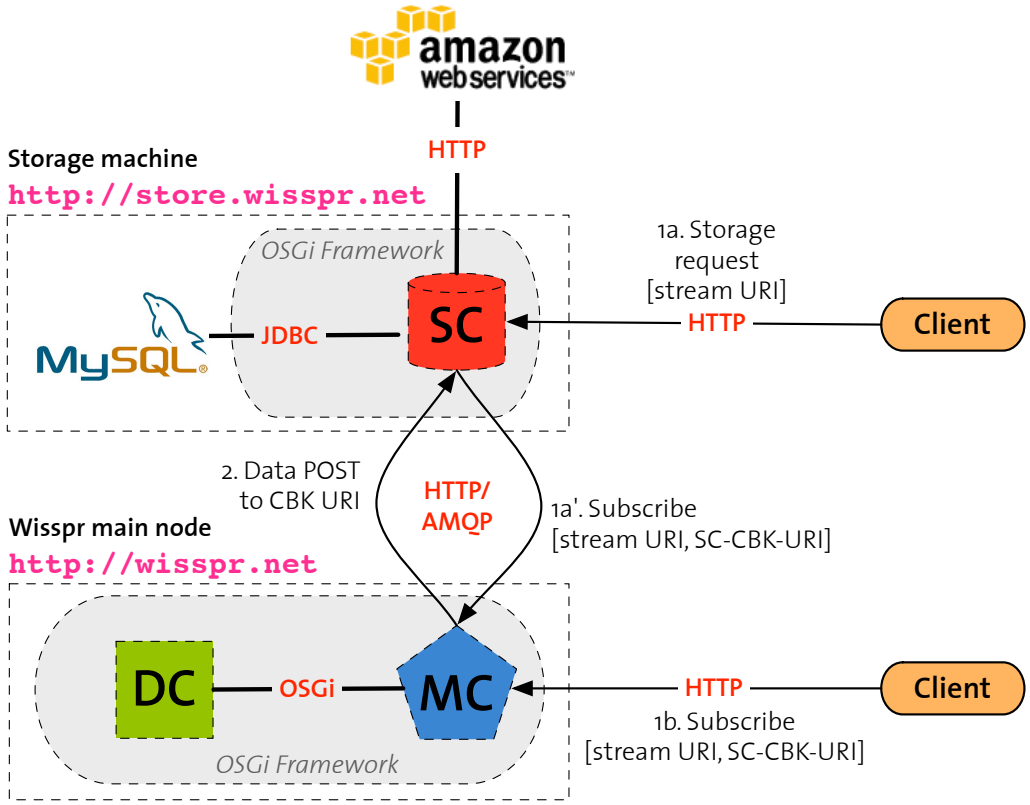



**Figure 6.22: Persisting a data stream**. A client issues a request to the storage connector (SC), instructing it to store a given data stream (1a). The connector will then subscribe to the stream on the messaging connector (MC) giving its own callback URI as parameter (1a'). Alternatively, the client can also directly subscribe the SC to the MC (1b), but then it needs to also specify the call-back URI of the storage connector. Data is then send to SC (2), who will store it locally in a database, or on cloud storage services such as Amazon SimpleDB.

For security purposes, it is very likely that the messaging connector will require credentials to access and use the storage, however we don't want the user to disclose his credentials to the messaging connector. To avoid disclosing any credential, OAuth can be used for delegated authentication [31]. The user authenticates himself against the storage connector and authorizes its use by the messaging connector, this way the credentials are not transferred to the messaging connector. Given that all interactions are HTTP-based, one can use use HTTPS to secure requests.

**Exposing Stored Data over the Web** Making data available over the Web is a central aspect of our project with a particular emphasis on dynamic data rather than stored, static data. Cloud

databases such as Amazon SimpleDB already offers data access over the Web [4] by providing a language similar to SQL to query a database. If data was stored using a storage system that already provides a Web interface then the storage connector only forwards the user to that interface.

For traditional relational databases, Web access can be provided by publishing data in RDF or HTML formats, and by allowing to query the data using the SPARQL query language. There has been a recent boom in Web interfaces on top of relational DBs and document oriented data stores (Persevere [37], CouchDB [6]), where RESTful APIs can be used to read and write data in stores. As our architecture emphasizes the use of REST for interacting with the storage connector, future research in providing efficient storage could be directly integrated with our system.

So far, only relational database management systems (MySQL [30], PortgreSQL [39], SimpleDB [4]) are supported in Wisspr, but additional storage engines could be added easily. Cloud databases and other Web-enabled databases are particularly easy to integrate with Wisspr, because they support a RESTful interface directly.

## 6.5 Prototypes and Evaluation

In this section, we describe two test applications we have developed to evaluate the ability of Wisspr to match various requirements. These evaluations have been undertaken in collaboration with Oliver Senn to be considered for publication[10], and have been proposed in [208]. We reproduce the results of these evaluations here for informational purpose. In the first scenario (which was inspired by [241]), a couple of devices monitor the seismic activity and must send an event to a base station as soon as possible when an anomaly has been detected, therefore low end-to-end latency (in the order of a second) at a moderate sampling frequency (50 Hz) is essential, while scalability is less of a concern. In the second scenario, many smart meters in a building or neighborhood have to send energy consumption data periodically to the energy company to be logged for billing. Assuming secure and encrypted interactions, sampling period can be relatively low (in the order of a minute), however the ability to support a large amount of devices is essential.

### 6.5.1 Case Study I: Detection Applications

Our first case study address *detection* applications. This class of applications covers sensor deployments for detecting abnormal activity and notify as quickly as possible external

[10] Unpublished work.

applications or human operators. These applications are concerned with low-latency and high reliability, and we arbitrarily limit the maximal latency to 5 seconds. For example, any solution to detect incidents or security fall in this category (fire detectors, burglar alarms, etc.).

In our experiment, we replicate the famous volcano monitoring experiment by Werner-Allen et al. [241] where a sensor network was deployed to monitor seismic[11] activity. We simulate 20 devices with acceleration sensors shared in 5 streams (each stream will contain an event every time a sensor in the group detects motion above a threshold). We want to be informed as soon as any device in a group detects an acceleration above threshold in any dimensions. Machine A (Core 2 Duo processor with 2GB Ram) runs a device and messaging connector modules and the message broker. Machine B (connected via a 100MBit/s LAN with the first one) acts as the client and registers a data stream for each of group of devices.

The client is then notified over HTTP each time a seismic event is detected. We are interested in the end-to-end latency, that is the time between a device detects abnormal activity and the time when the client application has received the notification. Because we are interested by the infrastructure behavior and not the particularities of wireless communication and the issues faced by a real application (external and environmental factors) we do not address here the details of the timing and other issues involved at the WSN level, which can be significant, but are out of scope of this thesis.

To measure the performance of the filtering mechanism on the messaging connector, we simulate 20 sensor nodes that sample the environment at a fixed rate and create four data streams that get notifications (each stream contains the aggregated data from a group of five devices) when acceleration in one dimension exceeds a predefined threshold (for this experiment the filter is true, so all messages are forwarded and maximal load reached).

#### Results

The sampling frequency (SF) of the sensors is varied from 5 to 100 Hz, and Table 6.7 shows the sampling rate in Hz for each device and the corresponding overall load in messages per second (msg/s) and then the internal *processing time* (time data from sensor is queued in Wisspr before it is send to the broker as a message) and the *fetching time* (delay between the message is sent to the broker and arrives at the client).

With our implementation and setup, we have observed small delays (below 100 ms) for up to 1000 msgs/s (sampling frequency of 50Hz), and the delay starts degrading pretty quickly with more messages. One can see that the internal processing time for each message is much smaller than the sending and fetching over HTTP (Figure 6.23). Besides, one can see that the fetching time increases much faster with more messages that the internal processing time. This means

[11] Obviously, the nodes are deployed in our lab and not on a real volcano like in the original deployment.

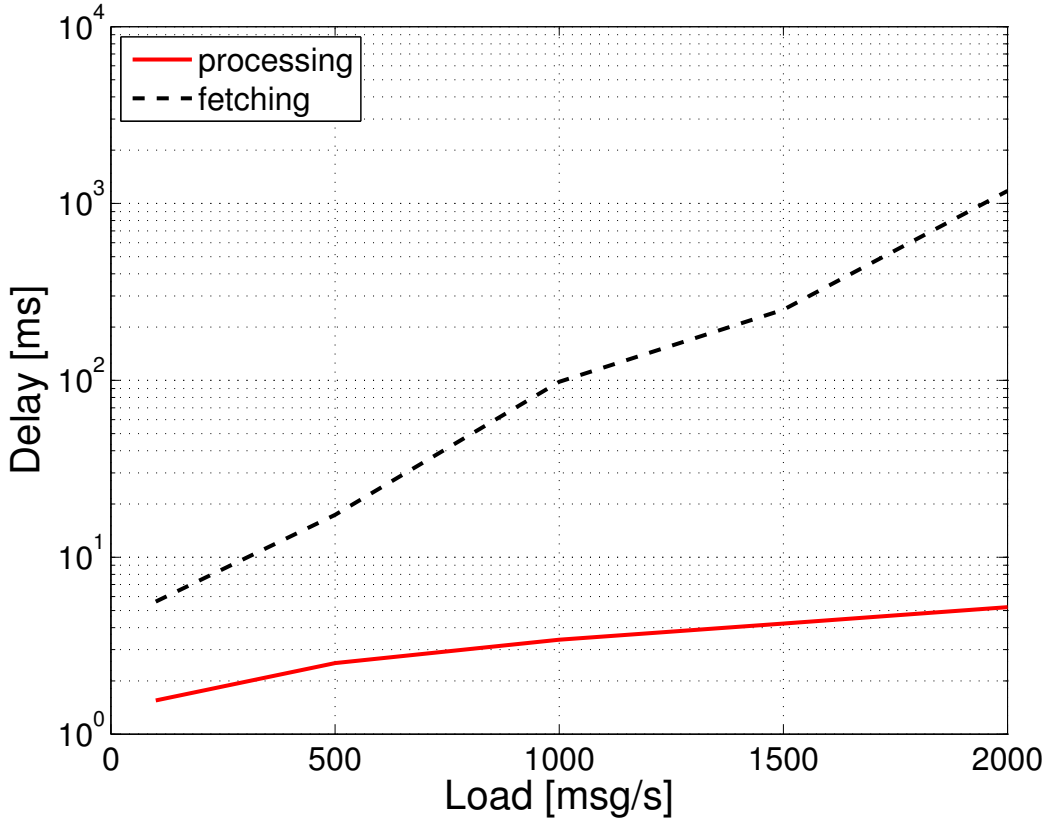



**Figure 6.23:** Average time required by the internal processing in Wisspr and for fetching and processing the message on the client for each message. As seen by the logarithmic scale, the processing time scales well with higher load (msg/s), while the fetching times degrades much faster.

that fetching the messages via RabbitHub takes most of the time and that the limitation comes from HTTP messaging itself and not from our implementation of Wisspr. Another cause of the performance degradation was that the client side could not handle the reception of more than 500 messages per second (which means receiving more than 500 HTTP `POST`s per second) as the default Web server used on the client to receive notifications (grizzly) was not tuned for such high loads, as we found out during our tests. With optimized versions of RabbitHub and of the Web server used by clients, superior performance could be easily obtained.

### 6.5.2 Case Study II: Data Collection Applications

In this second case study, we consider the scenario of an energy company that wants to monitor the electricity consumption of their customers. Each apartment in a residential building (see Figure 6.24) is equipped with several energy monitors that measure periodically the energy consumed. All the measurements are collected by a local device connector in each apartment that processes the data locally and forwards it to the energy company through a common

| SF [Hz] | Msgs/s | Processing [ms] | Fetching [ms] | Total [ms] |
|---|---|---|---|---|
| 5 | 100 | 1.5 | 5.6 | 7.3 |
| 25 | 500 | 2.5 | 17.3 | 20.0 |
| 50 | 1000 | 3.4 | 98.0 | 101.5 |
| 75 | 1500 | 4.2 | 250.8 | 255.1 |
| 100 | 2000 | 5.2 | 1177.7 | 1183.0 |

**Table 6.7:** Scenario 1 results: average end-to-end delay for event detection with various sampling frequencies.

message broker shared by the whole building. The business application of the energy supplier company (for example an ERP system) will then subscribe to the energy consumption in each apartment via the messaging connector of the building (whose URI is `http://buildingA.ch`).

The energy supplier would also like to store the generated data for later analysis. For that, the company sets up an internal storage connector on their own cluster. The storage connector is then configured to receive the energy data and then forwards it to the storage engine. The supplier set up a MySQL server to store the data. Furthermore, the data is also stored in an Amazon SimpleDB so that it can be used by a university which is conducting a study regarding energy consumption. The supplier can then integrate a query processing engine to filter the data that is stored on the SimpleDB instance, for example to anonymize the data before being publicized.

### Results

As in this scenario, throughput and reliability is the most important aspect, so we only focus on the storage of data from sensors, with very little processing on the messages and simply put all the data collected into the database. To evaluate this this scenario, we use two desktop machines and using RabbitHub as message broker (network setup is the same as in Figure 6.22). Machine A runs the device connector with simulated energy meters and the RabbitHub message broker, while machine B runs a storage connector and MySQL to store the data, and subscribes to all the messages from the simulated devices on machine A. We ran three tests with respectively 50, 100, and 200 simulated devices and varied the sampling frequency of each simulated device from 1 to 16 Hz (50 to 800 msgs/s in total).

| #D | 1 Hz | 2 Hz | 4 Hz | 10 Hz | 14 Hz | 16 Hz |
|---|---|---|---|---|---|---|
| 50 | 51.8 | 53.3 | 82.66 | 163.74 | 906.63 | 8269.2 |
| 100 | 69.0 | 169.4 | 246.51 | 23477 | 44226 | 57095 |
| 200 | 218.1 | 294.4 | 3585.5 | 66905 | 89510 | 90561 |

**Table 6.8:** Scenario 2 results: Average end-to-end delay (milliseconds) for different numbers of devices (#D) and sampling rates.

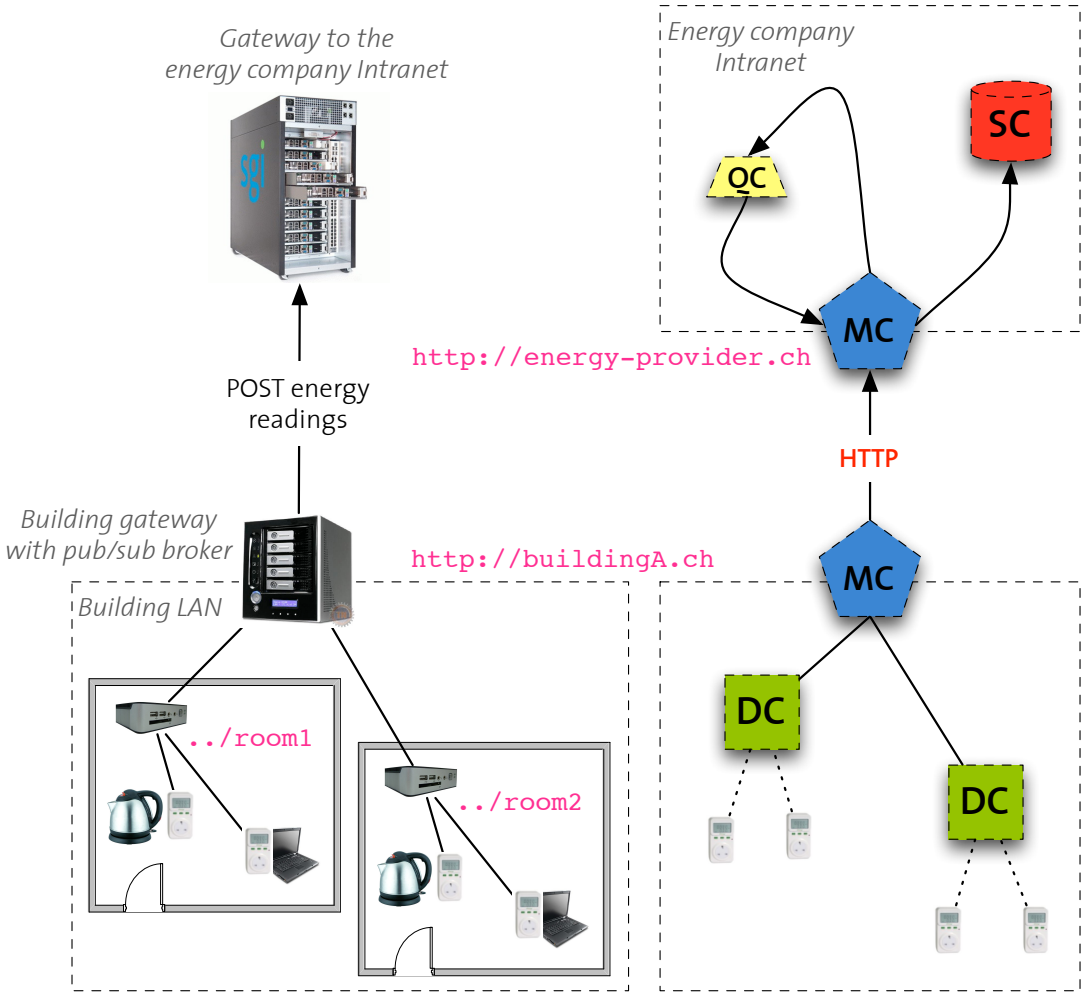



**Figure 6.24: Data Collection Application Architecture.** Each room of a building has a local device connector (for example running on a wireless router) that collects the energy readings of the smart meters in the apartment. A central broker (implemented within a Messaging Connector) in the building subscribes to the gateways of each apartment, processes the data (aggregates) and transmits it in an encrypted format to the subscribers (among which the energy company).

The experiment uses three data fields and to store each message we need 473 bytes (averaged). The data throughput can simply be calculated by multiplying the messages/second rate with the size of each message. From our results shown in Figure 6.25 and Table 6.8, the delay is very small (160 ms) for up to 500 msgs/s (sampling frequency of 10Hz). The problem is the huge variation incurred by more messages (i.e., when the message fetching is too slow). Because this test is very similar to the end-to-end test in the previous section (only with an additional database insert operation), the additional delay comes from the storage connector which handles the transformation of the JSON object and the insertion of its content into the database.

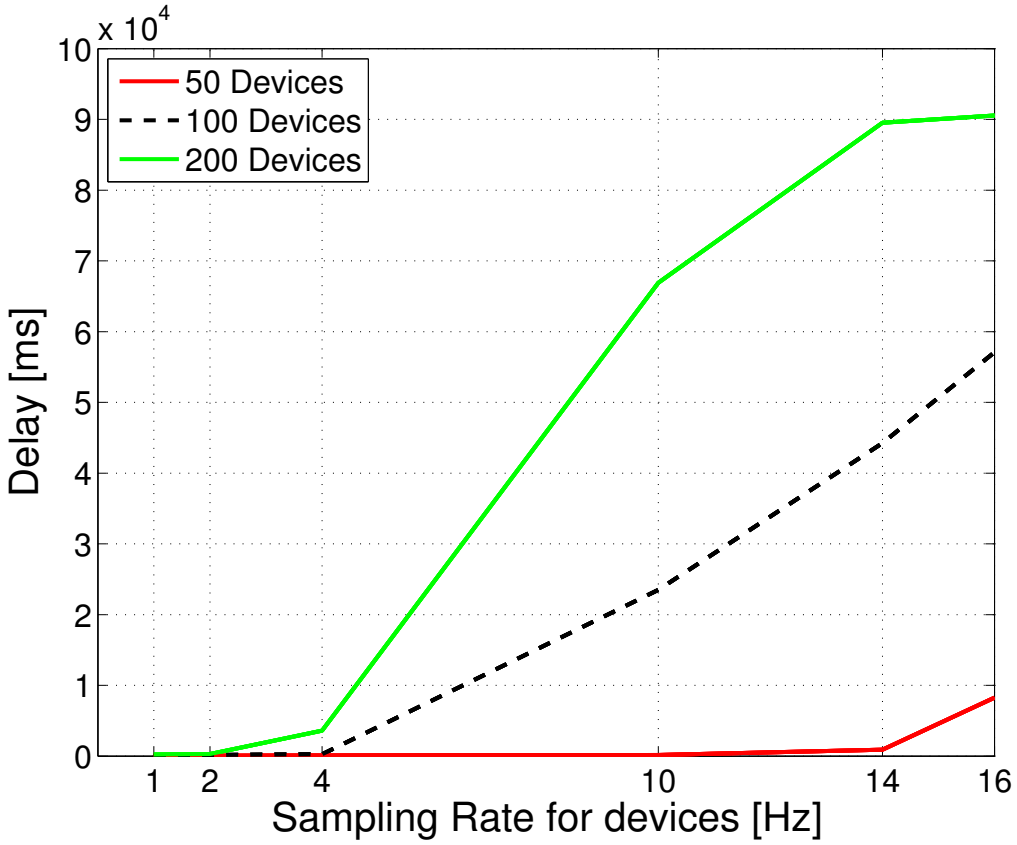



**Figure 6.25: Data Collection Application** End-to-end latency in milliseconds. With high data load, the storage connector significantly increases the latency because persisting data in a database requires much time.

## 6.6 Discussion

In this chapter, we have proposed two essential components for a programmable Web of Things. First, we have designed and evaluated WMS, a simple protocol for Web-based messaging, and showed that it can handle hundreds of messages per second with sub-second latency. Second, on top of this messaging system, we have built Wisspr, a complete framework for collecting, processing, sharing, and storing sensor data using Web standards in near real-time.

We proposed a model that allows to conveniently expose sensor on the Web in its true form: real-time data streams. The flexibility offered by the combination of a messaging system and a stream processing engine, allows to easily integrate data streams into Web applications and allow the community of Web developers to access and use real-time sensor data easily. The advantages of REST for decoupling components, facilitates the implementation of a modular architecture. Each component (connector) can offer high-level services directly exposed over Web APIs, while the implementation of the module can implement various third party applications, either general-purpose open source engines or specific ones designed for particular applications and requirements.

Certainly, higher data rates will be easily attainable with more powerful hardware and software optimizations. Wisspr can process several hundreds of messages per second, while exhibiting sub-second event delivery latency. Although this is inferior to the raw performance of optimized stream processing engines, it is certainly sufficient for most sensor-driven applications that can tolerate a few seconds of delay with hundreds of devices and thousands of concurrent users.

Large deployments used for environmental monitoring reported in sensor network literature (e.g., [170, 221]), rarely had more than 200 devices. Besides, the sampling frequency used in these deployment was rarely faster than one sample per minute. Low sampling rates in the order of a few samples per hour are common for the battery-powered devices used in sensor network deployments. Our experiments with Wisspr have shown that it can support thousands of messages per second with less than a second of end-to-end delay, which is sufficient to support the data throughput of most monitoring applications reported in sensor network literature.

No previous study has combined the three domains of Web programming, sensor networks, and stream processing. Even though much research will be needed for a flexible and exhaustive manner to expose and process streaming data over the Web, our research enables already a flexible manner to handle both raw sensor data and high-level, processed information using the same paradigm (Web messaging).

Our results show that the Web represents an excellent tradeoff between performance and features. To our knowledge, Wisspr is significantly simpler to use than existing solutions. Once installed, a complete application that collects, processes, and stores real-time sensor data can be developed with only three HTTP requests. In the meanwhile, the applications developed will scale well thanks to the loosely coupled nature of the Web. In addition to a functional prototype scheduled for release as open-source project, our results show that Web technologies are an excellent choice for building more flexible and open infrastructures for handling real-time sensor data.

# CHAPTER 7

# Conclusion

Traditionally, research in networked embedded sensing has mainly addressed particular problems within isolated deployments. Little effort has been addressing the issue of interoperability and integration of heterogenous devices to form large-scale participatory applications. As the actual rewards of networked sensors will stem from the integration of data from various sources, new models for building more scalable and open distributed sensing applications are required. In the meanwhile, the World Wide Web has been continuously evolving and today allows much richer and more complex interaction patterns, and modern Web applications offer to users the look-and-feel that was once reserved to desktop applications. The Web can be considered as one of the most successful distributed information infrastructure, and one of the main reason is that developing simple Web applications is easier and faster than with classical software projects, especially when it comes to data and services integration. As a consequence of the growing trend of Web 2.0 applications, the World Wide Web is experiencing two fundamental transformations. First, an expansion into the physical world as more and more appliances and other objects are being connected to the Internet (e.g., GPRS or WiFi). Second, the emergence of a real-time Web, where timely information is made available just after being produced using real-time messaging such as Twitter and RSS feeds. This real-time trend is significantly popularized by mobile phones with messaging applications.

In this thesis, we have addressed the limitations of existing solutions for building large-scale sensing applications, in particular the tight coupling of these solution that prevent the development of an open participatory infrastructure. By building upon the state-of-the-art developments in Web technologies, we have designed the building blocks for an infrastructure that

allows to develop complete sensing applications solely using Web standards. Our solution allows heterogenous devices to interact with each other using the same design patterns and standards that made the success of the Web, but with an additional layer of services (search, discovery, or caching) and interaction types (eventing and streaming) that are particularly suited to the requirements of future applications for embedded devices. The rationale behind using Web standards to connect devices is to benefit from the experience and tools gathered in the last two decades around building highly scalable and efficient Web applications for millions of concurrent requests. The openness and simplicity of REST facilitates the integration of real-time data from the physical world into any Web application with much less effort that required by existing solutions.

We have shown that the loss in performance caused by using Web standards in place of custom, optimized protocols commonly used for embedded devices does not prevent to develop complex and highly scalable stream processing applications with sub-second latency requirements. These results support the hypothesis that fully Web-based solutions are a promising solution for a new generation of open and scalable sensor data processing platforms that leverage all the advantages of Web technologies for an acceptable loss of performance. In a recent publication [175], it was mentioned *"The Web of Things is still just an architectural conceptualization and not a design for an integration architecture."*. In this thesis, we have proposed, designed, implemented, and evaluated the core components to remedy to the situation and laid concrete basis of an entirely Web-based integration architecture.

## 7.1 Future Work

The research and the solutions developed in this thesis are only an initial exploration of the future Web of Things. Much research and prototyping is still required to realize an efficient and global open sensing infrastructure, and we briefly highlight the future challenges that need to be tackled. First of all, an open-source project that further develops the gateway-based infrastructure we proposed in Chapters 4 and 5 will provide an initial software basis developers can build upon. In particular, drivers that allow Web-based access to the atomic services offered by many existing and future networked embedded devices will be required. Further exploration of the caching and scalable access to these services will need to be investigated to design a generic gateway that allows powerful, optimized, and transparent access to devices. In the meanwhile, an additional layer of cross-device services that augment the atomic functions offered by individual devices will be necessary. In particular, modules that handle automated analysis to offer data-centric access at different levels of granularity to the information collected by aggregating and processing information from heterogenous devices will be required. Similarly to an *app store*, various processing routines that are tailored for various domains from environmental monitoring, to home automation, to supply chain optimization will greatly augment the value offered by the Web of Things.

The obvious challenge this implies is how to efficiently and unambiguously describe the services and meta-information about devices and services offered by applications and devices. In particular, further exploration of how semantic technologies can encode this information to be read and understood by machines will greatly benefit automatic integration of new devices and their interaction with the other resources on the Web. Wisspr will need to further developed and better integrated with existing stream processing applications and messaging solutions to allow more scalable and heterogenous applications. In particular, much more research will be needed to understand how real-time data streams can be efficiently exposed, processed, queried, and searched using Web standards.

Security and privacy issues have only been marginally addressed in this thesis, yet they will be a fundamental challenge for future open sensing applications and need to be tackled. Fortunately, because the Web of Things builds upon the existing Web infrastructure, one can directly leverage all the tools and techniques for building scalable and secure Web applications developed over the last two decades. As we have shown in [137], one can for example use HTTPS and OAuth to enable authenticated and secure communication between mobile clients and gateways. Besides, one can use social networks to facilitate the sharing process with various circles of relatives, colleagues, and business partners.

Needless to say, a methodical and thorough performance and scalability evaluation with large deployments of machines and real sensors is necessary to carefully understand and quantify the behavior and limitations of a Web-based data collection infrastructure. Furthermore, techniques to improve the robustness and flexibility of such heterogenous distributed applications, particularly for business and industrial applications will be required. Nevertheless, our initial results support the idea that a fully Web-based system is able to process several hundred messages per second, while allowing concurrent users to pose continuous queries for monitoring heterogenous sensor data streams and operators to be notified in a timely manner about anomalies. Our experimental results suggests that Wisspr is sufficient for most sensor-driven application where a few seconds of delay are acceptable.

This thesis offers a window on what the future Web might look like, and we hope our results will inspire Web developers and sensor network researchers to think about new possibilities that arise when combining a truly location-aware infrastructure with the Web. The work presented here shall not be taken as a finite solution, but a mere prototypical draft to foster the exploration of a future Web of Things. Much applied research and prototypes will be required before device-oriented standards for the Web become widely adopted. However, we hope our initial results and positive experiences with REST on embedded devices will stimulate further efforts to construct the Web of Things.

# APPENDIX A

## Microformat Example

```
<h1>Sample Device: Light Controller</h1>
<span class="hentity">
  <h2>Device information:</h2>
    <h3>Device name</h3>
      <p><span class="fn n">light-controller</span></p>
    <h3>Device description:</h3>
      <p><span class="description">Smart home light controller.</span></p>
    <h3>Device tags:</h3>
    <ul>
      <li><span class="tag">light</span></li>
      <li><span class="tag">lamp</span></li>
    </ul>
    <h3>Device UUIDs:</h3>
    <ul>
      <li><span class="uuid"><span class="type">epc</span>:<span class="value"
          >5465464554646354164</span></span></li>
    </ul>
    <h3>Available operations:</h3>
    <ul>
      <li><span class="hoperation"><span class="fn n">mode</span>[<span class="
          acceptable-value">on</span>, <span class="acceptable-value">off</span
          >]</span></li>
```

```
      <li><span class="hoperation"><span class="fn n">brightness</span>[<span
          class="acceptable-value"><span class="range">0-100</span></span>]</span
          ></li>
    </ul>
    <h3>Web hook calllback URLs:</h3>
    <ul>
      <li><span class="rms"><a class="url" href="/rms/callback">/rms/callback</a
          >, filtering values: <span class="tag">entrance</span><span class="tag"
          >alarm</span></span></li>
    </ul>

    <span class="hproduct">
      <h2>Product information:</h2>
      <ul>
        <li>Brand: <span class="brand"><span class="fn n"><span class="given-
            name">Philips</span></span></span></li>
        <li>Category: <span class="category">HVAC devices</span></li>
        <li>Price: <span class="price">CHF 1995</span></li>
        <li>Description: <span class="description">Web enabled light control
            system.</span></li>
        <li>Product name: <span class="fn">PH10654654.</span></li>
      </ul>
    </span>

    <div class="geo">
      <h2>Geographical location</h2>
        <ul>
          <li>Latitude: <span class="latitude">37.386013</span></li>
          <li>Longitude: <span class="longitude">-122.082932</span></li>
          <li>Altitude: <span class="altitude">750</span></li>
        </ul>
  </div>

  <div class="indoor-location">
      <h2>Indoor location</h2>
        <p><span class="hierarchy">/ETHZ/IFW/4/49.2</span></p>
  </div>
</span>
```

**Listing A.1:** Microformats annotations used to describe a device and its operations, in this case a photosensor of a sensor node.

# Bibliography


[1] Adobe Flex. Online at http://www.adobe.com/products/flex/. Accessed 1. June 2011.

[2] Aginova. Online at http://aginova.com. Accessed 1. June 2011.

[3] Ajax Push Engine. Online at http://www.ape-project.org/. Accessed 1. June 2011.

[4] Amazon SimpleDB.
Online at http://aws.amazon.com/simpledb/. Accessed on 1. June 2011.

[5] AMQP Specification - Advanced Message Queueing Protocol. Online at http://www.amqp.org/. Accessed on 1. June 2011.

[6] Apache CouchDB. Online at http://couchdb.apache.org/. Accessed 1. June 2011.

[7] Apple bonjour. Online at http://en.wikipedia.org/wiki/Bonjour_%28software%29. Accessed on 1. June 2011.

[8] Cabspotting. Online at http://cabspotting.org/. Accessed 1. June 2011.

[9] Citysense. Online at http://www.sensenetworks.com/citysense.php. Accessed 1. June 2011.

[10] db4o database. Online at http://www.db4o.com/. Accessed 1. June 2011.

[11] Distributed Component Object Model (DCOM) Remote Protocol Specification.
Online at http://msdn.microsoft.com/library/cc201989.aspx. Accessed 1. June 2011.

[12] Esper, complex event processing. Online at http://esper.codehaus.org/. Accessed 1. June 2011.

[13] Evrythng. Online at http://www.evrythng.net. Accessed 1. June 2011.

[14] Gainspan. Online at http://www.gainspan.com/. Accessed 1. June 2011.

[15] How to GET a cup of coffee. Online at http://www.infoq.com/articles/webber-rest-workflow. Accessed on 1. June 2011.

[16] Java Messaging System (JMS). Online at http://java.sun.com/products/jms. Accessed 1. June 2011.

[17] Java remote method invocation (rmi). Online at http://download.oracle.com/javase/tutorial/rmi/index.html. Accessed 1. June 2011.

[18] Jini specification. Online at http://www.jini.org. Accessed on 1. June 2011.

[19] jQTouch. Online at http://jqtouch.com/. Accessed 1. June 2011.

[20] JSR 94: Java Rule Engine API. Online at http://jcp.org/en/jsr/detail?id=94. Accessed 1. June 2011.

[21] KML Documentation. Online at http://www.opengeospatial.org/standards/kml/. Accessed 1. June 2010.

[22] Lantronix. Online at http://www.lantronix.com/device-networking/embedded-device-servers/matchport.html. Accessed 1. June 2011.

[23] Lightstreamer. Online at http://www.lightstreamer.com/. Accessed 1. june 2011.

[24] lighttpd Web server. Online at http://www.lighttpd.net/. Accessed 1. June 2011.

[25] Microchip. Online at http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2884. Accessed 1. June 2011.

[26] Microsoft healhvault. Online at http://www.healthvault.com/personal/index.aspx. Accessed 1. June 2011.

[27] Microsoft silverlight. Online at http://www.silverlight.net/. Accessed 1. June 2011.

[28] MoteIV TMote Sky. Online at http://www.snm.ethz.ch/Projects/TmoteSky. Accessed 1. June 2011.

[29] MQ Telemetry Transport. Online at http://mqtt.org. Accessed 1. June 2011.

[30] MySQL database. Online at http://mysql.com/. Accessed on 1. June 2011.

[31] OAuth Community. Online at http://oauth.net/. Accessed 1. June 2011.

[32] Open sound control. Online at http://opensoundcontrol.org/. Accessed 1. June 2011.

[33] Opensearch. Online at http://www.opensearch.org/. Accessed 1. June 2011.

[34] Open.sen.se. Online at http://open.sen.se/. Accessed 1. june 2011.

[35] OSGi service platform, release 4, Enterprise Specification. Online at http://www.osgi.org/Release4/. Accessed 1. June 2011.

[36] Pachube. Online at www.pachube.com. Accessed: 1. June 2011.

[37] Persevere framework. Online at http://code.google.com/p/persevere-framework/. Accessed 1. June 2011.

[38] PhoneGap. Online at http://www.phonegap.com/. Accessed 1. June 2011.

[39] PostgreSQL Database.
Online at http://www.postgresql.org/. Accessed on 1. June 2011.

[40] Programmable web. Online at http://www.programmableweb.com/. Accessed 1. June 2011.

[41] RabbitMQ, Clustering Guide.
Online at http://www.rabbitmq.com/clustering.html. Accessed on 1. June 2011.

[42] RESTduino. Online at https://github.com/jjg/RESTduino. Accessed 1. June 2011.

[43] Restlet - RESTful Web framework for Java. Online at http://www.restlet.org/. Accessed on 1. June 2011.

[44] RestMS. Online at http://www.restms.org/. Accessed on 1. June 2011.

[45] ReverseHttp. Online at http://reversehttp.net/. Accessed on 1. June 2011.

[46] Round solutions. Online at http://www.roundsolutions.com/shop/en/WiFi-Module. Accessed 1. June 2011.

[47] Seeclickfix. Online at http://www.seeclickfix.com/citizens. Accessed 1. June 2011.

[48] Sencha Touch. Online at http://www.sencha.com/products/touch/. Accessed 1. June 2011.

[49] Sensorpedia. Online at http://www.sensorpedia.com/. Accessed 1. June 2011.

[50] Simple Object Access Protocol (SOAP). Online at http://www.w3.org/TR/soap/. Accessed 1. June 2011.

[51] SPDY, A chromium project. Online at http://www.chromium.org/spdy. Accessed 1. June 2011.

[52] A special report on smart systems: It's a smart world. The Economist Nov. 4th 2010, print edition. Online: http://www.economist.com/node/17388368. Accessed 1. June 2011.

[53] Sproxil. Online at http://www.sproxil.com/. Accessed 1. June 2011.

[54] StreamBase Streaming Platform. Online at http://www.streambase.com/. Accessed 1. June 2011.

[55] The Friend of a Friend (FOAF) project. Online at http://www.foaf-project.org/. Accessed 1. June 2011.

[56] Thingworx. Online at http://www.thingworx.com/. Accessed 1. June 2011.

[57] Uddi version 3.0.2 technical committee draft. Online at http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm. Accessed 1. June 2011.

[58] UPnP Forum. Online at http://www.upnp.org. Accessed 1. June 2011.

[59] Vitality glow caps. Online at http://www.vitality.net/. Accessed 1. June 2011.

[60] XEP-xxxx: Sensor-Over-XMPP. Online at http://xmpp.org/extensions/inbox/sensors.html. Accessed 1. June 2011.

[61] Yahoo pipes. Online at http://pipes.yahoo.com/pipes/. Accessed 1. June 2011.

[62] ZeroMQ. Online at http://www.zeromq.org. Accessed 1. June 2011.

[63] An Introduction to GeoRSS: A Standards Based Approach for Geo-enabling RSS feeds. OGC 06-050r3, July 2006.

[64] Abadi, D., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. The Design of the Borealis Stream Processing Engine. In *Conference on Innovative Data Systems Research (CIDR'05)* (Asilomar, CA, January 2005).

[65] Abadi, D., Carney, D., U., Ç., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. Aurora: a new model and architecture for data stream management. *The VLDB Journal 12*, 2 (2003), 120–139.

[66] Aberer, K., Hauswirth, M., and Salehi, A. Infrastructure for Data Processing in Large-Scale Interconnected Sensor Networks. In *International Conference on Mobile Data Management* (2007), pp. 198–205.

[67] Abowd, G. D., and Mynatt, E. D. Charting past, present, and future research in ubiquitous computing. *ACM Trans. Comput.-Hum. Interact. 7*, 1 (2000), 29–58.

[68] Adida, B., and Birbeck, M. RDFa primer. W3C working draft, W3C, Mar. 2008. http://www.w3.org/TR/2008/WD-xhtml-rdfa-primer-20080317/.

[69] Adida, B., and Birbeck, M. RDFa Primer – Bridging the Human and Data Webs. World Wide Web Consortium, Note NOTE-xhtml-rdfa-primer-20081014, October 2008.

[70] Akyildiz, I., Su, W., Sankarasubramaniam, Y., and Cayirci, E. Wireless sensor networks: a survey. *Computer Networks 38*, 4 (2002), 393–422.

[71] Alonso, G., and Casati, F. *Web Services*, 1 ed. Springer, 2003.

[72] Arasu, A., Babu, S., and Widom, J. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal 15*, 2 (2006), 121–142.

[73] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (New York, NY, USA, 2002), ACM, pp. 1–16.

[74] Babu, S., Subramanian, L., and Widom, J. A data stream management system for network traffic management. In *Workshop on Network-Related Data Management (NRDM 2001)* (May 2001).

[75] Bagnasco, A., Cipolla, D., Occhipinti, D., Preziosi, A., and Scapolla, A. Application of web services to heterogeneous networks of small devices. *WSEAS Transactions on Information Science and Application 3*, 5 (2006), 1790–0832.

[76] Bai, L., Dick, R., and Dinda, P. Archetype-based design: Sensor network programming for application experts, not just programming experts. In *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN 2009)* (2009).

[77] Barrenetxea, G., Ingelrest, F., Schaefer, G., and Vetterli, M. The hitchhiker's guide to successful wireless sensor network deployments. In *Proceedings of the 6th ACM conference on embedded network sensor systems (SenSys08)* (New York, NY, USA, 2008), ACM, pp. 43–56.

[78] Bauer, M., Becker, C., and Rothermel, K. Location models from the perspective of context-aware applications and mobile ad hoc networks. *Personal Ubiquitous Computing 6*, 5-6 (December 2002), 322–328.

[79] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and Norrman, K. The Secure Real-time Transport Protocol (SRTP). RFC 3711 (Proposed Standard), Mar. 2004. Updated by RFC 5506.

[80] Beigl, M., Zimmer, T., and Decker, C. A location model for communicating and processing of context. *Personal Ubiquitous Computing 6*, 5-6 (December 2002), 341–357.

[81] Benford, S., Crabtree, A., Flintham, M., Drozd, A., Anastasi, R., Paxton, M., Tandavanitj, N., Adams, M., and Row-Farr, J. Can you see me now? *ACM Trans. Comput.-Hum. Interact. 13*, 1 (2006), 100–133.

[82] Berners-Lee, T. Information management: A proposal. Online at http://www.w3.org/History/1989/proposal.html. Accessed 1. June 2011, May 1990.

[83] Berners-Lee, T., Fielding, R. T., and Masinter, L. Uniform resource identifier (uri): Generic syntax. Internet RFC 3986, Jan. 2005.

[84] Bernstein, P. A. Middleware: a model for distributed system services. *Communications of the ACM 39* (February 1996), 86–98. ACM ID: 230809.

[85] Bizer, C., Heath, T., Berners-Lee, T., Heath, T., Hepp, M., and Bizer, C. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems (IJSWIS)* (2009).

[86] Bolliger, P. Redpin - adaptive, zero-configuration indoor localization through user collaboration. *Workshop on Mobile Entity Localization and Tracking in GPS-less Environment Computing and Communication Systems (MELT), San Francisco* (2008).

[87] Boonma, P., and Suzuki, J. Middleware support for pluggable non-functional properties in wireless sensor networks. In *SERVICES '08: Proceedings of the 2008 IEEE Congress on Services - Part I* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 360–367.

[88] Borriello, G., and Want, R. Embedded computation meets the World Wide Web. *Commun. ACM 43*, 5 (2000), 59–66.

[89] Botan, I., Cho, Y., Derakhshan, R., Dindar, N., Haas, L., Kim, K., Lee, C., Mundada, G., Shan, M., Tatbul, N., Yan, Y., Yun, B., and Zhang, J. Design and Implementation of the MaxStream Federated Stream Processing Architecture. Tech. Rep. TR-632, ETH Zurich Department of Computer Science, June 2009.

[90] Botan, I., Cho, Y., Derakhshan, R., Dindar, N., Haas, L., Kim, K., and Tatbul, N. Federated Stream Processing Support for Real-Time Business Intelligence Applications. In *VLDB International Workshop on Enabling Real-Time for Business Intelligence (BIRTE'09)* (Lyon, France, August 2009).

[91] Bozdag, E., Mesbah, A., and van Deursen, A. Performance testing of data delivery techniques for ajax applications. *J. Web Eng. 8*, 4 (2009), 287–315.

[92] Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., and Klemmer, S. R. Opportunistic programming: Writing code to prototype, ideate, and discover. *IEEE Software 26* (2009), 18–24.

[93] Brumitt, B., Meyers, B., Krumm, J., Kern, A., and Shafer, S. Easyliving: Technologies for intelligent environments. In *Handheld and Ubiquitous Computing: Second International Symposium (HUC 2000)* (Bristol, UK, September 2000), vol. 1927 of *LNCS*, Springer, pp. 12–27.

[94] Campbell, A. T., Eisenman, S. B., Lane, N. D., Miluzzo, E., and Peterson, R. A. People-centric urban sensing. In *Proceedings of the 2nd annual international workshop on Wireless internet* (Boston, Massachusetts, 2006), WICON '06, ACM. ACM ID: 1234179.

[95] Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. Monitoring Streams - A New Class of Data Management Applications. In *International Conference on Very Large Data Bases (VLDB'02)* (Hong Kong, China, August 2002).

[96] Chalmers, M., and MacColl, I. Seamful and seamless design in ubiquitous computing. Tech. Rep. Equator-03-005, Equator, 2003.

[97] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. A. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceeding of the Biennial Conference on Innovative Data Systems Research (CIDR)* (2003).

[98] Chang, K., Yau, N., Hansen, M., and Estrin, D. SensorBase.org - A centralized repository to slog sensor network data, 2006.

[99] Chu, X., Kobialka, Durnota, B., and Buyya, R. Open sensor web architecture: Core services. In *Fourth International Conference on Intelligent Sensing and Information Processing, 2006. ICISIP 2006* (December 2006), IEEE, pp. 98–103.

[100] Cuff, D., M., H., and Kang, J. Urban sensing: out of the woods. *Communications of the ACM 51*, 3 (2008), 24–33.

[101] Davidovski, V. A web-oriented infrastructure for interacting with digitally augmented environments. Master's thesis, Departement of Computer Science, ETH Zurich, 2010.

[102] de Souza, L. M. S., Spiess, P., Guinard, D., Koehler, M., Karnouskos, S., and Savio, D. Socrades: A web service based shop floor integration infrastructure. In *Proceedings of the IEEE International Conference on the Internet of Things (IoT 2008)* (Zurich, Switzerland, 2008), Springer.

[103] Delicato, F., Pires, P., Pirmez, L., and da Costa Carmo, L. A flexible Web service based architecture for wireless sensor networks. In *Distributed Computing Systems Workshops* (2003), pp. 730–735.

[104] Demers, A., Gehrke, J., Hong, M., Riedewald, M., and White, W. Towards expressive publish/subscribe systems. *Advances in Database Technology-EDBT 2006* (2006), 627–644.

[105] Demirkol, I., Ersoy, C., and Alagoz, F. MAC protocols for wireless sensor networks: a survey. *Communications Magazine, IEEE 44*, 4 (2006), 115–121.

[106] Dey, A. K. *Providing Architectural Support for Building Context-Aware Applications.* PhD thesis, College of Computing, Georgia Institute of Technology, 2000.

[107] Dickerson, R., Lu, J., Lu, J., and Whitehouse, K. Stream Feeds: an Abstraction for the World Wide Sensor Web. In *Proceedings of the IEEE International Conference on the Internet of Things (IoT 2008)* (Zurich, Switzerland, 2008).

[108] Drytkiewicz, W., Radusch, I., Arbanowski, S., and Popescu-Zeletin, R. pREST: a REST-based protocol for pervasive systems. In *Proc. of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems* (Oct. 2004), pp. 340–348.

[109] Dunkels, A. Full TCP/IP for 8-bit architectures. In *Proceedings of the 1st international conference on mobile systems, applications and services* (2003), ACM, p. 98.

[110] Dunkels, A., Gronvall, B., and Voigt, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)* (Tampa, Florida, USA, 2004).

[111] Dunkels, A., and Vasseur, J. IP for Smart Objects Alliance. Internet Protocol for Smart Objects (IPSO) Alliance White paper, September 2008.

[112] Dunkels, A., Voigt, T., and Alonso, J. Making TCP/IP Viable for Wireless Sensor Networks. In *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN 2004), work-in-progress session* (Berlin, Germany, Jan. 2004).

[113] Duquennoy, S., Grimaud, G., and Vandewalle, J.-J. Consistency and scalability in event notification for embedded web applications. In *11th IEEE International Symposium on Web Systems Evolution (WSE'09)* (Edmonton, Canada, September 2009).

[114] Duquennoy, S., Grimaud, G., and Vandewalle, J.-J. Smews: Smart and mobile embedded web server. In *3rd International Workshop on Intelligent, Mobile and Internet Services in Ubiquitous Computing (IMIS'09)* (Fukuoka, Japan, March 2009).

[115] Duquennoy, S., Grimaud, G., and Vandewalle, J.-J. The Web of Things: interconnecting devices with high usability and performance. In *6th International Conference on Embedded Software and Systems (ICESS'09)* (Hangzhou, Zhejiang, China, May 2009).

[116] Durvy, M., Abeillé, J., Wetterwald, P., O'Flynn, C., Leverett, B., Gnoske, E., Vidales, M., Mulligan, G., Tsiftes, N., Finne, N., and Dunkels, A. Making Sensor Networks IPv6 Ready. In *Proceedings of the Sixth ACM Conference on Networked Embedded Sensor Systems (ACM SenSys 2008), poster session* (Raleigh, North Carolina, USA, Nov. 2008).

[117] Eagle, N. Behavioral Inference Across Cultures: Using Telephones as a Cultural Lens. *IEEE Intelligent Systems 23* (July 2008), 62–64. ACM ID: 1449437.

[118] Edwards, W. K., Newman, M. W., Sedivy, J., Smith, T., and Izadi, S. Challenge: recombinant computing and the speakeasy approach. In *MobiCom '02: Proceedings of the 8th annual international conference on mobile computing and networking* (New York, NY, USA, 2002), ACM, pp. 279–286.

[119] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. The many faces of publish/subscribe. *ACM Comput. Surv. 35*, 2 (June 2003), 114–131.

[120] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[121] Fielding, R. T., Gettys, J., Mogul, J. C., Frystyk Nielsen, H., Masinter, L., Leach, P. J., and Berners-Lee, T. Hypertext transfer protocol — http/1.1. Internet RFC 2616, June 1999.

[122] Fitzpatrick, B., Slatkin, B., and Atkins, M. PubSubHubbub Core 0.3 – Working Draft. Online at http://pubsubhubbub.googlecode.com/svn/trunk/pubsubhubbub-core-0.3.html. Accessed on 1. June 2011.

[123] Fowler, M. Richardson Maturity Model: steps toward the glory of REST. Online at http://martinfowler.com/articles/richardsonMaturityModel.html, March 2010.

[124] Franklin, M. J., Jeffery, S. R., Krishnamurthy, S., and Reiss, F. Design considerations for high fan-in systems: The hifi approach. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research* (2005), pp. 290–304.

[125] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L. Http authentication: Basic and digest access authentication. Internet

RFC 2616. Online at http://www.apps.ietf.org/rfc/rfc2617.html. Accessed 1. June 2011, 1999.

[126] Friday, A., Davies, N., Wallbank, N., Catterall, E., and Pink, S. Supporting service discovery, querying and interaction in ubiquitous computing environments. *Wirel. Netw. 10*, 6 (2004), 631–641.

[127] Galpin, I., Brenninkmeijer, C. Y., Jabeen, F., Fernandes, A. A., and Paton, N. W. Comprehensive optimization of declarative sensor network queries. In *SSDBM 2009: Proceedings of the 21st International Conference on Scientific and Statistical Database Management* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 339–360.

[128] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Nov. 1994.

[129] Gershenfeld, N. *FAB: The Coming Revolution on Your Desktop - From Personal Computers to Personal Fabrication*. Basic Books, 2005.

[130] Gibbons, P., Karp, B., Ke, Y., Nath, S., and Seshan, S. IrisNet: an architecture for a worldwide sensor Web. *Pervasive Computing, IEEE 2*, 4 (2003), 22–33.

[131] Girod, L., Elson, J., Cerpa, A., Stathopoulos, T., Ramanathan, N., and Estrin, D. Emstar: a software environment for developing and deploying wireless sensor networks. *Proceedings of the 2004 USENIX Technical Conference* (2004).

[132] Girod, L., Lukac, M., PARKER, A., STATHOPOULOS, T., TSENG, J., WANG, H., ESTRIN, D., GUY, R., and KOHLER, E. A reliable multicast mechanism for sensor network applications. CENS Tech. rep. 48, University of California, Los Angeles, 2005.

[133] Gnawali, O., Jang, K., Paek, J., Vieira, M., Govindan, R., Greenstein, B., Joki, A., Estrin, D., and Kohler, E. The tenet architecture for tiered sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems (SENSYS)* (Boulder, Colorado, USA, 2006), ACM, pp. 153–166.

[134] Gomadam, K., Ranabahu, A., and Sheth, A. SA-REST: Semantic Annotation of Web Resources. Online at http://www.w3.org/Submission/SA-REST/. Accessed 1. June 2011.

[135] Gregory, H., Chien-Liang, F., Gruia-Catalin, R., and Lu, C. Agimone: Middleware support for seamless integration of sensor and IP networks. In *Procedings of the second IEEE international conference (DCOSS 2006)* (San Francisco, CA, USA, 2006).

[136] Guinard, D. *A Web of Things Application Architecture - Integrating the Real-World into the Web*. PhD thesis, ETH Zurich, 2011.

[137] Guinard, D., Fischer, M., and Trifa, V. Sharing using social networks in a composable web of things. In *Proc. of the First IEEE International Workshop on the Web of Things (WOT2010)*. (Mannheim, Germany, March 2010).

[138] Guinard, D., Mueller, M., and Trifa, V. *In: Erik Wilde and Cesare Pautasso (Ed.): REST: From Research to Practice*. No. 16. Springer, August 2011, ch. RESTifying Real-World Systems: a Practical Case Study in RFID.

[139] Guinard, D., and Trifa, V. 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), Madrid, Spain* (2009).

[140] Guinard, D., Trifa, V., Karnouskos, S., Spiess, P., and Savio, D. Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *IEEE Transactions on Services Computing 3*, 3 (2010), 223–235.

[141] Guinard, D., Trifa, V., Mattern, F., and Wilde, E. *Architecting the Internet of Things*. No. 5. Springer, May 2011, ch. From the Internet of Things to the Web of Things: Resource Oriented Architecture and Best Practices.

[142] Guinard, D., Trifa, V., Pham, T., and Liechti, O. Towards Physical Mashups in the Web of Things. In *Proceedings of the Sixth International Conference on Networked Sensing Systems (INSS)* (2009), pp. 1–4.

[143] Guinard, D., Trifa, V., Spiess, P., Dober, B., and Karnouskos, S. Discovery and on-demand provisioning of real-world web services. In *Proceedings of the 2009 IEEE International Conference on Web Services (ICWS 09)* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 583–590.

[144] Guinard, D., Trifa, V., and Wilde, E. A resource oriented architecture for the web of things. In *Proceedings of the IEEE International Conference on the Internet of Things (IoT 2010)* (Tokyo, Japan, November 2010).

[145] Guinard, D., Weiss, M., and Trifa, V. Are you energy-efficient? sense it on the web! In *Adjunct Proceedings of Pervasive 2009 (International Conference on Pervasive Computing)* (Nara, Japan, May 2009).

[146] Gupta, V., Poursohi, A., and Udupi, P. Sensor.Network: An Open Data Exchange for the Web of Things. In *Proc. of the First IEEE International Workshop on the Web of Things (WOT2010), PerCom Workshops* (2010), pp. 753–755.

[147] Hadley, M. Web Application Description Language. Online at http://www.w3.org/Submission/wadl/. Accessed 1. June 2011.

[148] Hartmann, B., Doorley, S., and Klemmer, S. R. Hacking, Mashing, Gluing: Understanding Opportunistic Design. *IEEE Pervasive Computing 7* (July 2008), 46–54. ACM ID: 1449424.

[149] Hatler, M., Gurganious, D., Chi, C., and Ritter, M. WSN for Smart Industries. OnWorld Study, 2007.

[150] Heinzelman, W. B., Murphy, A. L., Carvalho, H. S., and Perillo, M. A. Middleware to support sensor network applications. *IEEE Network 18*, 1 (2004), 6–14.

[151] Hickson, I. "6.2 Server-sent DOM events". HTML 5 - Call For Comments. WHATWG. Online at http://www.whatwg.org/specs/web-apps/2007-10-26/multipage/section-server-sent-events.html. Accessed 05.06.2010.

[152] Hohpe, G., Woolf, B., and Brown, K. *Enterprise integration patterns*. Addison-Wesley Professional, 2004.

[153] Hornsby, A., Belimpasakis, P., and Defee, I. XMPP-based wireless sensor network and its integration into the extended home environment. In *IEEE 13th International Symposium on Consumer Electronics, 2009. ISCE '09* (May 2009), IEEE, pp. 794–797.

[154] Hui, J. W., and Culler, D. E. IP is dead, long live IP for wireless sensor networks. In *SenSys '08: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems* (New York, NY, USA, 2008), ACM, pp. 15–28.

[155] Hunkeler, U., Truong, H. L., and Stanford-Clark, A. MQTT-S – A publish/subscribe protocol for Wireless Sensor Networks. In *Proceedings of the Third International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE 2008)* (Bangalore, India, January 2008), pp. 791–798.

[156] Ishii, H., and Ullmer, B. Tangible bits: Towards seamless interfaces between people, bits and atoms. In *CHI* (1997), pp. 234–241.

[157] Jammes, F., Mensch, A., and Smit, H. Service-oriented device comunications using the devices profile for web services. In *Proc. of 3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC05) at the 6th International Middleware Conference* (2005).

[158] Jiang, C., and Steenkiste, P. A hybrid location model with a computable location identifier for ubiquitous computing. In *Proceedings of the 4th international conference on Ubiquitous Computing* (Göteborg, Sweden, 2002), Springer-Verlag, pp. 246–263.

[159] Johansson, I., and Westerlund, M. Support for Reduced-Size Real-Time Transport Control Protocol (RTCP): Opportunities and Consequences. RFC 5506 (Proposed Standard), Apr. 2009.

[160] Juntunen, J. K., Kuorilehto, M., Kohvakka, M., Kaseva, V. A., Hannikainen, M., and Hamalainen, T. D. WSN API: application programming interface for wireless sensor networks. In *2006 IEEE 17th International Symposium on Personal, Indoor and Mobile Radio Communications* (September 2006), IEEE, pp. 1–5.

[161] Kamilaris, A. A lightweight resource-oriented application framework for wireless sensor networks. Master's thesis, ETH Zurich, 2009.

[162] Kamilaris, A., Trifa, V., and Guinard, D. Building web-based infrastructures for smart meters. In *Workshop on Energy Awareness and Conservation through Pervasive Applications at Pervasive 2010* (Helsinki, Finland, May 2010).

[163] Kamilaris, A., Trifa, V., and Pitsillides, A. The Smart Home Meets the Web of Things. *Int. J. of Ad Hoc and Ubiquitous Computing 7*, 3 (2011).

[164] Kansal, A., Nath, S., Liu, J., and Zhao, F. SenseWeb: an infrastructure for shared sensing. *IEEE Multimedia 14*, 4 (2007), 8–13.

[165] Khare, R., and Çelik, T. Microformats: A pragmatic path to the semantic web. In *Proceedings of the 15th International World Wide Web Conference (WWW2006)* (Edinburgh, Scotland, 2006).

[166] Kim, J.-H., Kim, D.-H., Kwak, H.-Y., and Byun, Y.-C. Address internetworking between wsns and internet supporting web services. In *MUE '07: Proceedings of the 2007 International Conference on Multimedia and Ubiquitous Engineering* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 232–240.

[167] Kindberg, T., Barton, J., Morgan, J., Becker, G., Caswell, D., Debaty, P., Gopal, G., Frid, M., Krishnan, V., Morris, H., Schettino, J., Serra, B., and Spasojevic, M. People, places, things: web presence for the real world. *Mob. Netw. Appl. 7*, 5 (2002), 365–376.

[168] King, T., Kopf, S., Haenselmann, T., Lubberger, C., and Effelsberg, W. Compass: A probabilistic indoor positioning system based on 802.11 and digital compasses. *Proceedings of the First ACM International Workshop on Wireless Network Testbeds, Experimental evaluation and CHaracterization (WiNTECH)* (Aug 2006).

[169] Lane, G. Urban tapestries: Wireless networking, public authoring and social knowledge. *Personal Ubiquitous Computing*, 7 (April 2003), 169–175.

[170] Langendoen, K., Baggio, A., and Visser, O. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *14th Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)* (apr 2006).

[171] Langheinrich, M., Mattern, F., Römer, K., and Vogt, H. First steps towards an event-based infrastructure for smart things. In *In Ubiquitous Computing Workshop, PACT 2000* (2000).

[172] Lei, S., Xiaoling, W., Hui, X., Jie, Y., Cho, J., and Lee, S. Connecting Heterogeneous Sensor Networks with IP Based Wire/Wireless Networks. In *SEUS-WCCIA '06: Proceedings of the Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06)* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 127–132.

[173] Li, G., and Jacobsen, H.-A. Composite subscriptions in content-based publish/subscribe systems. In *Middleware '05: Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware* (New York, NY, USA, 2005), Springer-Verlag, pp. 249–269.

[174] Ljungstrand, P., Redström, J., and Holmquist, L. E. Webstickers: using physical tokens to access, manage and share bookmarks to the web. In *DARE '00: Proceedings of DARE 2000 on Designing augmented reality environments* (New York, NY, USA, 2000), ACM, pp. 23–31.

[175] López, T. S., Ranasinghe, D., Harrison, M., and McFarlane, D. Adding sense to the Internet of Things - an architecture framework for Smart Object systems. *Personal and Ubiquitous Computing* (2011). (to be published in print - published online 03 Jun 2011.

[176] Luckenbach, T., Gober, P., Arbanowski, S., Kotsopoulos, A., and Kim, K. TinyREST - a protocol for integrating sensor networks into the internet. *Proc. of REALWSN* (2005).

[177] Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. The design of an acquisitional query processor for sensor networks. In *ACM SIGMOD* (2003), ACM Press, pp. 491–502.

[178] Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst. 30*, 1 (2005), 122–173.

[179] Mainland, G., Morrisett, G., and Welsh, M. Flask: staged functional programming for sensor networks. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on functional programming* (New York, NY, USA, 2008), ACM, pp. 335–346.

[180] Mark, W. Turning pervasive computing into mediated spaces. *IBM Systems Journal 38*, 4 (1999).

[181] Markowetz, A., yu Chen, Y., and Suel, T. Design and implementation of a geographic search engine. In *Eigth Int. Workshop on the Web and Databases (WebDB)* (2005).

[182] Mattern, F., and Floerkemeier, C. *From the Internet of Computers to the Internet of Things*, vol. 6462 of *LNCS*. Springer, 2010, pp. 242–259.

[183] Mattern, F., and Sturm, P. From distributed systems to ubiquitous computing - the state of the art, trends, and prospects of future networked systems. In *K. Irmscher and K.-P. Fähnrich, editors, Proc. KIVS 2003* (2003), Springer-Verlag, pp. 3–25.

[184] Mayer, S. Deployment support for an infrastructure for web-enabled devices. Master's thesis, Departement of Computer Science, ETH Zurich, 2010.

[185] Mayer, S., and Guinard, D. An extensible discovery service for smart things. In *Proceedings of the Second International Workshop on Web of Things* (San Francisco, CA, USA, June 2011), ACM, p. 7.

[186] Mayer, S., Guinard, D., and Trifa, V. Facilitating the integration and interaction of real-world services for the web of things. In *Proceedings of the First Urban Internet of Things workshop (Urban-IOT 2010)* (Tokyo, Japan, November 2010).

[187] Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., and Varma, R. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)* (2003).

[188] Mueller, F. F., Stevens, G., Thorogood, A., O'Brien, S., and Wulf, V. Sports over a distance. *Personal Ubiquitous Comput. 11* (December 2007), 633–645.

[189] Mühl, G., Fiege, L., and Pietzuch, P. *Distributed Event-Based Systems*. Springer-Verlag, 2006.

[190] Nelson, T. H. *Literary Machines*, 93 ed. Mindful Press, September 1994.

[191] Nottingham, M. FIQL: The Feed Item Query Language. Internet Draft online at http://tools.ietf.org/html/draft-nottingham-atompub-fiql-00. Accessed 1. June 2011.

[192] OASIS. Devices Profile for Web Services (DPWS) specification. Online at http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01. Accessed 1. June 2011.

[193] O'Reilly, T. What is web 2.0. Online at http://oreilly.com/web2/archive/what-is-web-20.html. Accessed 1. June 2011.

[194] Pardo-Castellote, G. OMG data-distribution service: Architectural overview. In *Proceedings. 23rd International Conference on Distributed Computing Systems Workshops* (2003), pp. 200–206.

[195] Pastrone, C., Spirito, M. A., Tomasi, R., and Rizzo, F. A Jabber-Based Management Framework for Heterogeneous Sensor Network Applications. *International Journal of Software Engineering and Its Applications 2*, 3 (2008), 9–24.

[196] Paterson, I., Smith, D., Saint-Andre, P., and Moffitt, J. XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH). Online at http://www.xmpp.org/. Accessed on 1. June 2011.

[197] Pautasso, C., and Wilde, E. Why is the web loosely coupled? a multi-faceted metric for service design. In *Proceedings of the 18th International World Wide Web Conference* (2009), pp. 911–920.

[198] Pautasso, C., Zimmermann, O., and Leymann, F. RESTful Web Services vs. "big" web services: making the right architectural decision. In *Proceedings of the 17th International World Wide Web Conference* (Beijing, China, April 2008), pp. 805–814.

[199] Pham, T. Resource oriented architecture in wireless sensor network: Smartlogger in the context of a supply chain. Master's thesis, University of Applied Sciences of Western Switzerland, 2008.

[200] Prehofer, C., van Gurp, J., and di Flora, C. Towards the Web as a Platform for Ubiquitous Applications in Smart Spaces. In *Second Workshop on Requirements and Solutions for Pervasive Software Infrastructures (RSPSI), at Ubicomp 2007* (2007).

[201] Priyantha, N. B., Kansal, A., Goraczko, M., and Zhao, F. Tiny web services: design and implementation of interoperable and evolvable sensor networks. In *SenSys '08: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems* (New York, NY, USA, 2008), ACM, pp. 253–266.

[202] Raggett, D. Towards the web of things. Slides online at www.w3.org/2007/Talks/0926-dsr-WDC/slides.pdf.

[203] Resch, B., Calabrese, F., Biderman, A., and Ratti, C. An Approach Towards Real-Time Data Exchange Platform System Architecture (concise contribution). In *IEEE International Conference on Pervasive Computing and Communications* (Los Alamitos, CA, USA, 2008), vol. 0, IEEE Computer Society, pp. 153–159.

[204] Römer, K., Kasten, O., and Mattern, F. Middleware challenges for wireless sensor networks. *SIGMOBILE Mob. Comput. Commun. Rev. 6*, 4 (2002), 59–61.

[205] Römer, K., Ostermaier, B., Mattern, F., Fahrmair, M., and Kellerer, W. Real-Time Search for Real-World Entities: A Survey. *Proceedings of the IEEE 98*, 11 (2010), 1887–1902.

[206] Rooney, S., and Garces-Erice, L. Messo & Preso Practical Sensor-Network Messaging Protocols. *Proceedings of ECUMN'2007* (2007), 364–376.

[207] Russel, A., Wilkins, G., Davis, D., and M., N. Bayeux Protocol. Online at http://svn.cometd.com/trunk/bayeux/bayeux.html. Accessed on 1. June 2011.

[208] Senn, O. WISSPR: A Web-based Infrastructure for Sensor Data Streams Sharing, Processing and Storage. Master's thesis, ETH Zurich, 2010.

[209] Sgroi, M., Wolisz, A., Sangiovanni-Vincentelli, A., and Rabaey, J. M. A service-based universal application interface for ad hoc wireless sensor and actuator networks. *Ambient Intelligence* (2005), 149–172.

[210] Shaw, G. M., and David, G. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[211] Shelby, Hartke, Bormann, and Frank. Constrained application protocol (coap). Current draft online at http://tools.ietf.org/html/draft-ietf-core-coap-06, May 2011.

[212] Shilton, K. Four billion little brothers? *Communications of the ACM 52*, 11 (2009), 48.

[213] Shneidman, J., Pietzuch, P., Ledlie, J., Roussopoulos, M., Seltzer, M., and Welsh, M. Hourglass: An infrastructure for connecting sensor networks and applications. Tech. Rep. TR-21-04, Harvard University, 2004.

[214] Srivastava, M., Hansen, M., Burke, J., Parker, A., Reddy, S., Saurabh, G., Allman, M., Paxson, V., and Estrin, D. Wireless urban sensing systems. CENS Technical Report 65, University of California Los Angeles, April 2006.

[215] Stahl, C., and Heckmann, D. Using semantic web technology for ubiquitous hybrid location modelling. In *1st Workshop on Ubiquitous GIS, in conjunction with 12th International Conference on Geoinformatics* (2004).

[216] Stirbu, V. Towards a RESTful Plug and Play Experience in the Web of Things. In *IEEE International Conference on Semantic Computing* (Aug. 2008), pp. 512–517.

[217] Suba, F., Prehofer, C., and van Gurp, J. Towards a common sensor network API: practical experiences. In *International Symposium on Applications and the Internet (SAINT)* (2008), pp. 185–188.

[218] Szewczyk, R., Mainwaring, A., Polastre, J., Anderson, J., and Culler, D. An analysis of a large scale habitat monitoring application. In *SenSys '04: Proceedings of the 2nd international conference on embedded networked sensor systems* (New York, NY, USA, 2004), ACM, pp. 214–226.

[219] Tatbul, N. Streaming Data Integration: Challenges and Opportunities. In *IEEE ICDE International Workshop on New Trends in Information Integration (NTII'10)* (Long Beach, CA, March 2010).

[220] Tian, F., Reinwald, B., Hamid, P., Mayr, T., and Myllymaki, J. Implementing a scalable XML publish/subscribe system using relational database systems. In *Proceedings of the international conference on management of data (ACM SIGMOD)* (New York, NY, USA, 2004), ACM, pp. 479–490.

[221] Tolle, G., Polastre, J., Szewczyk, R., Culler, D., Turner, N., Tu, K., S., B., Dawson, T., Buonadonna, P., Gay, D., et al. A macroscope in the redwoods. In *Proceedings of the 3rd international conference on Embedded networked sensor systems* (2005), ACM New York, NY, USA, pp. 51–63.

[222] Traversat, B., Abdelaziz, M., Doolin, D., Duigou, M., Hugly, J., and Pouyoul, E. Project JXTA-C: Enabling a Web of Things. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences* (2003), pp. 282–290.

[223] Trifa, V. A framework for bird songs detection, recognition and localization using acoustic sensor networks. Master's thesis, Ecole Polytechnique Federale de Lausanne, 2006.

[224] Trifa, V. Content creation on the web: Mashing up the real world with the internet. In *Proceedings of The First International Workshop on Contents Creation Activity Support by Networked Sensing (CCASNS)* (Kanazawa, Japan, 2008).

[225] Trifa, V., Cianci, C., and Guinard, D. Dynamic control of a robotic swarm using a service-oriented architecture. In *Proceedings of the International Symposium on Artificial Life and Robotics (AROB 13th 2008)* (2008), M. Sugisaka and H. Tanaka, Eds., no. ISBN: 978-4-9902880-2-0.

[226] Trifa, V., Guinard, D., Bolliger, P., and Wieland, S. Design of a Web-based Distributed Location-Aware Infrastructure for Mobile Devices. In *Proc. of the First IEEE International Workshop on the Web of Things (WOT2010)* (Mannheim, Germany, March 2010), pp. 714–719.

[227] Trifa, V., Guinard, D., Davidovski, V., Kamilaris, A., and Delchev, I. Web messaging for open and scalable distributed sensing applications. In *Proc. of the 10th International Conference on Web Engineering (ICWE 2010)* (Vienna, Austria, June 2010).

[228] Trifa, V., Guinard, D., and Mayer, S. *In: Erik Wilde and Cesare Pautasso (Ed.): REST: From Research to Practice.* No. 17. Springer-Verlag, 2011, ch. Leveraging the Web for a Distributed Location-aware Infrastructure for the Real World.

[229] Trifa, V., Wieland, S., Guinard, D., and Bohnert, T. M. Design and implementation of a gateway for web-based interaction and management of embedded devices. In *Proceedings of the 2nd International Workshop on Sensor Network Engineering (IWSNE'09)* (Marina del Rey, CA, USA, June 2009).

[230] Trossen, D., and Pavel, D. Building a ubiquitous platform for remote sensing using smartphones. In *Mobile and Ubiquitous Systems: Networking and Services* (2005).

[231] Valli, A. Natural interaction. White Paper, September 2007.

[232] van Dam, T., and Langendoen, K. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the 1st international conference on embedded networked sensor systems* (Los Angeles, California, USA, 2003), SenSys '03, ACM, pp. 171–180. ACM ID: 958512.

[233] van Gurp, J., Prehofer, C., and di Flora, C. Experiences with realizing smart space web service applications. *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE* (Jan. 2008), 1171–1175.

[234] Vazquez, J. I., de Ipiña, D. L., and Sedano, I. SoaM: A Web-powered Architecture for Designing and Deploying Pervasive Semantic Devices. *IJWIS - International Journal of Web Information Systems 2*, 3-4 (2006).

[235] Vinoski, S. Serendipitous reuse. *IEEE Internet Computing 12*, 1 (2008), 84–87.

[236] W3C. Web Services Description Language (WSDL) 1.1 Specification. Online at http://www.w3.org/TR/wsdl. Accessed 1. June 2011.

[237] Wang, M.-M., Cao, J.-N., Li, J., and Dasi, S. K. Middleware for wireless sensor networks: A survey. *Journal of Computer Science and Technology 23*, 3 (May 2008), 305–326.

[238] Wang, N., Meng, X., and Yu, L. Real-time forest fire detection with wireless sensor networks. In *Proceedings of the International Conference on Wireless Communications Networking and Mobile Computing* (2005), vol. 2, pp. 1214–1217.

[239] Weiser, M. The computer for the 21st century. *Scientific American 265*, 3 (1991), 94–104.

[240] Weiss, M., and Guinard, D. Increasing energy awareness through web-enabled power outlets. In *Proceedings of MUM 2010 (9th ACM SIGMOBILE Conference on Mobile and Ubiquitous Multimedia)* (Limassol, Cyprus, December 2010).

[241] Werner-Allen, G., Lorincz, K., Johnson, J., Lees, J., and Welsh, M. Fidelity and yield in a volcano monitoring sensor network. In *OSDI '06: Proceedings of the 7th symposium on operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 381–396.

[242] Whitehouse, K., Zhao, F., and Liu, J. Semantic streams: A framework for composable semantic interpretation of sensor data. In *Proceedings of the 3rd European Workshop on Wireless Sensor Networks (EWSN 2006)* (Zurich, Switzerland, February 2006), pp. 5–20.

[243] Wieland, S. Design and implementation of a gateway for web-based interaction and management of embedded devices. Master's thesis, ETH Zurich, 2009.

[244] Wilde, E. Putting things to REST. Tech. Rep. UCB iSchool Report 2007-015, School of Information, UC Berkeley, November 2007.

[245] Wilde, E., and Kofahl, M. The locative Web. In *Proceedings of the First International Workshop on Location and the Web* (Beijing, China, 2008), ACM, pp. 1–8.

[246] Wilde, E., and Marinos, A. Feed querying as a proxy for querying the web. In *FQAS '09: Proceedings of the 8th International Conference on Flexible Query Answering Systems* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 663–674.

[247] Woo, A., Seth, S., Olson, T., Liu, J., and Zhao, F. A spreadsheet approach to programming and managing sensor networks. In *Proceedings of the fifth international conference on information processing in sensor networks (IPSN)* (New York, NY, USA, 2006), ACM, pp. 424–431.

[248] Yan, T., He, T., and Stankovic, J. A. Differentiated surveillance for sensor networks. In *Proceedings of the 1st international conference on embedded networked sensor systems* (Los Angeles, California, USA, 2003), SenSys '03, ACM, pp. 51–62. ACM ID: 958498.

[249] Yao, Y., and Gehrke, J. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec. 31*, 3 (2002), 9–18.

[250] Yazar, D. Restful wireless sensor networks. Master's thesis, Uppsala University, 2009.

[251] Yazar, D., and Dunkels, A. Efficient Application Integration in IP-Based Sensor Network. In *Proc. of the First ACM Workshop on embedded sensing systems for energy-efficiency in buildings (BuildSys), at SenSys09* (2009).

[252] Ye, W., Heidemann, J., and Estrin, D. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)* (2002), vol. 3, pp. 1567–1576.

[253] Yick, J., Mukherjee, B., and Ghosal, D. Wireless sensor network survey. *Computer Networks 52*, 12 (2008), 2292–2330.

# Curriculum Vitae: Vlad Trifa

**Personal Data**

| | |
|---|---|
| Date of Birth | 12 March, 1982 |
| Birthplace | Oradea, Romania |
| Citizenship | Swiss |
| Status | Single |

**Education**

| | |
|---|---|
| 2007–2011 | **ETH Zurich, Switzerland**<br>Ph.D. Student at the Department of Computer Science |
| 2010–2011 | **Massachusetts Institute of Technology (MIT), USA**<br>Visiting Researcher at the Senseable City Lab |
| 2005–2006 | **University of California, Los Angeles (UCLA), USA**<br>Visiting Graduate Student in Bioacoustics (Master thesis) |
| 2001–2005 | **École Polytechnique Fédérale de Lausanne (EPFL), Switzerland**<br>M.Sc. in Computer Science (Robotics, AI, Neuroscience) |
| 1998–2001 | **Gymnase de Burier, Switzerland**<br>Baccalauréat in Science (High school), La Tour-de-Peilz, Switzerland |

**Employment**

| | |
|---|---|
| 2010 – 2010 | Scientific Collaborator, *Singapore-MIT Alliance for Research and Technology*, Singapore |
| 2010 – 2010 | Visiting Researcher, *MIT*, Cambridge, MA, USA |
| 2007 – 2010 | Research Associate, *SAP Research*, Zurich, Switzerland |
| 2007 – 2011 | Research Assistant, *Institute for Pervasive Computing, ETH Zurich*, Zurich, Switzerland |
| 2006 – 2007 | Researcher, *Advanced Telecommunications Research Center (ATR)*, Kyoto, Japan |
| 2005 – 2006 | Photojournalist, *Daily Bruin*, Los Angeles, CA, USA |
| 2004 – 2007 | Research Assistant, *Distributed Intelligent Systems and Algorithms Laboratory (DISAL), EPFL, Switzerland* |