Diss. ETH No. 19891

A Web of Things Application Architecture - Integrating the Real-World into the Web

A dissertation submitted to
ETH Zurich

for the degree of
Doctor of Science

Presented by
Dominique Guinard
M.Sc. in Computer Science, University of Fribourg
born February 27, 1981
citizen of Switzerland

accepted on the recommendation of
Prof. Dr. Friedemann Mattern, examiner, ETH Zurich
Prof. Dr. Gustavo Alonso, co-examiner, ETH Zurich
Prof. Dr. Sanjay Sarma, co-examiner, MIT Boston

2011

Abstract

A central concern in the area of pervasive computing has been the integration of digital artifacts with the physical world and vice-versa. Recent developments in the field of embedded devices have led to *smart things* increasingly populating our daily life. We define smart things as digitally enhanced physical objects and devices that have communication capabilities. Application domains are for instance wireless sensor and actuator networks in cities making them more context-aware and thus smarter. New appliances such as smart TVs, alarm clocks, fridges or digital-picture frames make our living-rooms and houses more energy efficient and our lives easier. Industries benefit from increasingly more intelligent machines and robots. Usual objects tagged with radio-tags or barcodes become linked to virtual information sources and offer new business opportunities.

As a consequence, *Internet of Things* research is exploring ways to connect smart things together and build upon these networks. To facilitate these connections, research and industry have come up over the last few years with a number of low-power network protocols. However, while getting increasingly more connected, embedded devices still form multiple, small, incompatible islands at the application layer: developing applications using them is a challenging task that requires expert knowledge of each platform. As a consequence, smart things remain hard to integrate into composite applications. To remedy this fact, several service platforms proposing an integration architecture appeared in recent years. While some of them are successfully implemented on some appliances and machines, they are, for the most part, not compatible with one another. Furthermore, their complexity and lack of well-known tools let them only reach a relatively small community of expert developers and hence their usage in applications has been rather limited.

On the other hand, the Internet is a compelling example of a scalable global network of computers that interoperate across heterogeneous hardware and software platforms. On top of the Internet, the Web illustrates well how a set of relatively simple and open standards can be used to build very flexible systems while preserving efficiency and scalability. The cross-integration and developments of composite applications on the Web, alongside with its ubiquitous availability across a broad range of devices (e.g., desktops, laptops, mobile phones, set-top boxes, gaming devices, etc.), make the Web an outstanding candidate for a universal integration platform. Web sites do not offer only pages anymore, but Application Programming Interfaces that can be used by other Web resources to create new, ad-hoc and composite applications running in the computing cloud and being

accessed by desktops or mobile computers.

In this thesis we use the Web and its emerging technologies as the basis of a smart things application integration platform. In particular, we propose a Web of Things application architecture offering four layers that simplify the development of applications involving smart things. First, we address *device accessibility* and propose implementing, on smart things, the architectural principles that are at the heart of the Web such the Representational State Transfer (REST). We extend the REST architecture by proposing and implementing a number of improvements to fit the special requirements of the physical world such as the need for domain-specific proxies or real-time communication.

In the second layer we study *findability*: In a Web populated by billions of smart things, how can we identify the devices we can interact with, the devices that provide the right service for our application? To address these issues we propose a lightweight metadata format that search engines can understand, together with a Web-oriented discovery and lookup infrastructure that leverages the particular context of smart things.

While the Web of Things fosters a rather open network of physical objects, it is very unlikely that in the future access to smart things will be open to anyone. In the third layer we propose a *sharing* infrastructure that leverages social graphs encapsulated by social networks. We demonstrate how this helps sharing smart things in a straightforward, user-friendly and personal manner, building a Social Web of Things.

Our primary goal in bringing smart things to the Web is to facilitate their integration into composite applications. Just as Web developers and tech-savvies create Web 2.0 mashups (i.e., lightweight, ad-hoc compositions of several services on the Web), they should be able to create applications involving smart things with similar ease. Thus, in the *composition* layer we introduce the *physical mashups* and propose a software platform, built as an extension of an open-source workflow engine, that offers basic constructs which can be used to build mashup editors for the Web of Things.

Finally, to test our architecture and the proposed tools, we apply them to two types of smart things. First we look at wireless sensor networks, in particular at energy and environmental monitoring sensor nodes. We evaluate the benefits of applying the proposed architecture first empirically by means of several prototypes, then quantitatively by running performance evaluations and finally qualitatively with the help several developers who used our frameworks to develop mobile and Web-based applications. Then, to better understand and evaluate how the Web of Things architecture can facilitate the development of real-world aware business applications, we study automatic identification systems and propose a framework for bringing RFID data to the Web and global RFID information systems to the cloud. We evaluate the performance of this framework and illustrate its benefits with several prototypes.

Put together, these contributions materialize into an ecosystem of building-blocks for the Web of Things: a world-wide and interoperable network of smart things on which applications can be easily built, one step closer to bridging the gap between the virtual and physical worlds.

Kurzfassung

Die Integration von digitalen Artefakten mit der physischen Welt ist ein zentrales Anliegen des Pervasive Computing. Jüngste Entwicklungen im Bereich der eingebetteten Systeme haben dazu geführt, dass wir in unserem täglichen Leben immer öfter mit "smarten" Dingen – vernetzten, digital angereicherten Geräten – interagieren. Unsere Städte werden durch drahtlose Sensor- und Aktuatornetze intelligenter und ermöglichen kontextsensitives Verhalten. Neue Geräte wie intelligente Fernseher, Wecker, Kühlschränke oder digitale Bilderrahmen machen unsere Wohnungen und Häuser energieeffizienter und unser Leben angenehmer. Die Industrie profitiert von zunehmend intelligenteren Maschinen und Robotern. Alltägliche Objekte, die mit Funkchips markiert oder mit Strichcodes versehen sind, werden um virtuelle Informationsquellen erweitert und bieten neue Geschäftsmöglichkeiten.

Als Folge dieser Entwicklung wird in den letzten Jahren im Rahmen des *Internet der Dinge* nach Möglichkeiten gesucht, smarte Dinge miteinander zu vernetzen. Um das Verbinden von Geräten zu vereinfachen, haben Forschung und Industrie eine Reihe von Niedrigenergie-Kommunkationsprotokollen konzipiert und standardisiert. Eine Folge dieser Entwicklungen war jedoch, dass sich auf der Anwendungsebene unter den verbundenen Geräten immer mehr kleine, unvereinbare Inseln bildeten. Das Erstellen von Anwendungen für smarte Dinge ist heute eine anspruchsvolle Aufgabe, die Fachwissen über jede einzelne Plattform erfordert. Dies erschwert die Integration von vernetzten Alltagsgegenständen in geräteübergreifenden Anwendungen. Um dieser Entwicklung entgegenzuwirken, erschienen immer mehr Integrationsarchitekturen die zwar zum Teil erfolgreich eingesetzt werden, jedoch meistens nicht miteinander kompatibel sind. Ihre Komplexität und der Mangel an unterstützenden Werkzeugen führen dazu, dass sie nur von einer kleinen Gruppe von Experten verwendet werden können und damit ihr Nutzen für die Erstellung von innovativen Anwendungen bisher eher begrenzt ist.

Das Internet ist ein überzeugendes Beispiel für ein skalierbares weltweites Computernetz, in dem heterogene Hardware- und Softwareplattformen ohne Integrationsprobleme zusammenarbeiten. Des Weiteren zeigt das World Wide Web, wie durch die Nutzung von vergleichbar einfachen und offenen Standards hochflexible Systeme gebaut werden können, während die Effizienz und Skalierbarkeit des Internet weiterhin gewährleistet

sind. Aufgrund seiner breiten Verfügbarkeit auf verschiedenen Geräten (z.B. PCs, Laptops, Mobiltelefone, Set-Top-Boxen, Spielkonsolen etc.) und seiner hohen Flexibilität ist das Web ein hervorragender Kandidat für eine universelle Integrationsplattform. Webseiten bieten neben ihrem klassischen Inhalt auch Programmierschnittstellen, die von anderen Web-Ressourcen verwendet werden können, um innovative Anwendungen in der Cloud zu erstellen, auf die von Desktops oder mobilen Computern aus zugegriffen werden kann.

In dieser Arbeit schlagen wir das Web of Things als eine vierschichtige Applikationsintegrationsarchitektur vor, die die Entwicklung von Anwendungen mit smarten Dingen vereinfacht. Zunächst wenden wir uns dem Problem der *Zugänglichkeit* von Geräten zu und schlagen die Umsetzung der Prinzipien, die das Herzstück des Web bilden (z.B. Representational State Transfer, REST), auf smarten Dingen vor.

Des Weiteren haben wir die REST-Architektur erweitert, um die speziellen Anforderungen der physischen Welt – etwa die Notwendigkeit für domänenspezifische Proxies oder für Echtzeit-Kommunikation – zu berücksichtigen. Ausserdem betrachten wir die Frage der *Auffindbarkeit*: Wie können in einem Netz von Milliarden von smarten Dingen diejenigen Geräte, die für eine bestimmte Anwendung benötigte Dienste anbieten, gefunden werden, und wie kann mit ihnen interagiert werden? Zur Lösung dieser Probleme schlagen wir ein leichtgewichtiges, für Suchmaschinen lesbares, Metadaten-Format vor, das mit einer Web-basierten Auffindungs- und Suchinfrastruktur zusammenarbeitet, die den besonderen Kontext von smarten Dingen berücksichtigt.

Obwohl das Web of Things ein offenes Netz von physischen Objekten unterstützt, ist es unwahrscheinlich, dass der Zugriff auf smarte Dinge in Zukunft für jedermann uneingeschränkt möglich sein wird. Aus diesem Grund bauen wir auf der dritten Ebene eine Infrastruktur, die soziale Netzwerke verwendet, um das gemeinsame Nutzen von smarten Dingen zu ermöglichen. Wir zeigen, wie ein solches *social Web of Things* die *kollektive Verwendung* von physischen Artefakten auf benutzerfreundliche Art und Weise ermöglicht.

Das Hauptziel dieser Arbeit ist die Verwendung des Web, um die Integration von smarten Dingen in kollaborativen Anwendungen zu vereinfachen. Dadurch ermöglichen wir Webentwicklern und anderen Personen mit guten Computerkenntnissen, innovative, auf smarten Alltagsdingen basierende, Anwendungen zu entwickeln – so, wie sie heute Web 2.0-Mashups (leichtgewichtige, ad hoc aus verschiedenen Web-Diensten zusammengesetzte Applikationen) erstellen. Um dieses Ziel zu erreichen, stellen wir auf einer weiteren Ebene (*Composition Layer*) das Konzept von physischen Mashups vor. Wir schlagen ausserdem eine Softwareplattform vor, die als Erweiterung eines quelloffenen Workflow-Systems konzipiert wurde und Grundkonstrukte bereitstellt, um Mashup-Umgebungen für das Web of Things zu erstellen.

Wir verwenden zwei verschiedene Arten von smarten Dingen, um unsere vorgeschlagene Architektur und die dazugehörigen Werkzeuge zu testen: Zuerst betrachten wir drahtlose Sensorknoten, insbesondere solche, die für Umwelt- und Energieüberwachung eingesetzt werden. Wir bewerten die mit dem Einsatz unserer Architektur zusammenhängenden Effekte empirisch in verschiedenen Prototypen, quantitativ über Leistungsmessungen und

qualitativ mithilfe von mehreren Entwicklern, die unsere Systeme bei der Erstellung von mobilen Applikationen und Webanwendungen einsetzen. Um ausserdem zu verstehen, wie das Web of Things die Entwicklung von intelligenten Geschäftsanwendungen vereinfachen kann, betrachten wir Systeme zur automatischen Identifizierung von Gegenständen und schlagen ein System vor, mit dem RFID-Daten in das World Wide Web und globale RFID-Informationssysteme in die Cloud integriert werden können. Wir demonstrieren das Leistungsverhalten dieses Systems anhand von verschiedenen Prototypen.

Zusammengefasst liefern unsere Beiträge ein Ökosystem von Bausteinen für ein globales Netz von interoperablen smarten Dingen, die das Erstellen von geräteübergreifenden Anwendungen vereinfachen, welche die Kluft zwischen der virtuellen und der physischen Welt überbrücken.

Résumé

L'intégration du digital avec le réel reste l'une des préoccupations principales de l'informatique ubiquitaire et pervasive. De plus, les récents développements en informatique embarquée ont pour conséquence un déploiement croissant d'objets intelligents. Nous définissons les objets intelligents (appelés smart things dans cette thèse) comme des objets du monde réel doués d'une capacité de communication. Parmi les domaines d'application de ces objets, on peut citer: les réseaux de capteurs déployés dans nos villes modernes, les rendant plus intelligentes et adaptatives ou la domotique permettant à nos nouveaux téléviseurs, radio-réveils, frigidaires ou cadres à photos de nous rendre la vie plus facile et d'optimiser notre communication ou consommation d'énergie. De façon similaire, l'industrie bénéficie de robots et de machines de plus en plus intelligents et les biens de consommation sont équipés d'étiquettes électroniques ou de code-barres liés à des sources d'information virtuelles permettant de nouveaux cas d'utilisation.

Forte de l'engouement pour ces nouveau systèmes, la recherche dans le domaine de l'Internet des objets explore de nouvelles manières de connecter ces objets ensemble. Afin de faciliter ces connections, la recherche et l'industrie ont proposé plusieurs protocoles de communication. Toutefois, bien que ces protocoles facilitent la communication bas-niveau, les objets intelligents forment encore des îlots isolés les uns des autres au niveau applicatif. Par conséquent, la création d'applications pour des objets intelligents reste presque exclusivement accessible à des spécialistes du domaine de l'embarqué tout comme la création de services composites utilisant des objets intelligents. Pour contrer cette complexité, plusieurs plateformes orientées services proposent une architecture d'intégration. Toutefois, bien qu'implémentées avec succès sur quelques appareils et machines, ces plateformes ne sont souvent pas compatibles entre elles et leur complexité les rend difficiles d'accès aux novices.

En revanche, l'Internet se présente comme un très bon exemple de réseau global d'ordinateurs hétérogènes intégrés avec succès. Au-dessus de l'Internet, le Web quant à lui illustre comment un petit nombre de standards ouverts et relativement simples facilitent la construction d'applications complexes tout en préservant une certaine efficacité. De plus, la capacité du Web à supporter la création d'applications composites et interopérables ainsi que sa disponibilité pour une large palette d'appareils (p.ex., PC, portables, téléphones,

set-top boxes, consoles de jeu, etc.) en font un candidat idéal pour une plateforme d'intégration universelle. En effet, les sites Web ne sont plus de simples pages mais de véritables services qui peuvent être réutilisés en combinaison avec d'autres sites afin de créer dynamiquement de nouvelles applications s'exécutant en ligne et dont les clients sont de natures diverses.

Dans cette thèse nous utilisons le Web et ses technologies émergentes comme base pour une plateforme applicative intégrant les objets intelligents. Au sein de l'architecture applicative du Web des objets, nous proposons quatre couches permettant de simplifier la création d'applications. La première couche traite de *l'accès aux objets intelligents*. Nous y dissertons de l'adaptation et de l'implémentation dans le monde des objets des principes architecturaux du Web. En particulier nous étudions l'utilisation du "Representational State Transfer" (REST). Nous étendons l'architecture REST en proposant certaines adaptations afin de prendre en compte les contraintes des objets intelligents. A titre d'exemple, nous étudions l'utilisation de passerelles et proposons des modèles de communication en temps réel.

Dans la deuxième couche nous étudions la *recherche et la localisation des objets*. Dans un Web peuplé de milliards d'objets, comment pouvons-nous retrouver celui fournissant le service le plus adapté à notre application? Afin de répondre à cette question nous proposons un modèle léger de méta-données que les moteurs de recherche peuvent interpréter. De plus, nous implémentons un système de registre permettant d'effectuer des recherche de services en fonction du contexte particulier des clients et des objets intelligents.

Le Web des objets tel que nous le présentons dans cette thèse promouvoit un réseau global et ouvert d'objets intelligents. Pourtant, il est peu probable que nous souhaitions laisser l'accès libre à tous nos objets au reste du monde. Dans la troisième couche nous adressons ce problème en proposant une infrastructure permettant le *partage* d'objets sur le Web. Cette infrastructure utilise les réseaux sociaux afin de permettre un protocole de partage facilement utilisable et basé sur nos connections personnelles, créant ainsi un Web social des objets.

Notre but principal lorsque nous proposons d'amener les objets au plus proche du Web est de faciliter leur utilisation dans des applications composites. Tout comme les aficionados de la technologie et du Web créent facilement des "mashups" (càd. des applications légères et dynamiques utilisant plusieurs services du Web), ils devraient pouvoir en faire de même avec les objets intelligents. En conséquence, la troisième couche traite de la composition de services et introduit la notion de *mashups physiques*. Nous y proposons une plateforme logicielle construite comme une extension d'un gestionnaire de processus et offrant des éléments de langage permettant de créer des éditeurs de mashup pour les objets intelligents.

Finalement, afin de d'évaluer l'architecture et les outils proposés, nous nous attardons sur deux types d'objets intelligents. Tout d'abord nous considérons les réseaux de capteurs environnementaux. Les bénéfices de l'utilisation du Web des objets y sont testés de façon empirique par le biais de plusieurs prototypes, de façon quantitative à l'aide d'évaluations

de performances, puis de façon qualitative par le biais d'études avec des développeurs utilisant ces approches. Ensuite, nous étudions le cas des systèmes d'identification par ondes radio (RFID) et proposons une structure permettant d'amener les données et appareils RFID sur le Web. Nous évaluons les performances de cette structure et illustrons ses bénéfices par le biais de plusieurs prototypes.

Mises ensemble, les contributions de cette thèse proposent les fondations du futur Web des objets: un réseau d'objets et de services global et interopérable au-dessus duquel des applications peuvent être créées avec aisance. Cette thèse permet donc de diminuer le fossé qui existe encore entre notre monde de tous les jours et le monde virtuel.

Acknowledgements

Earning a Ph.D. degree would be an impossible journey without the support of professors, peers and loved ones. Here, I would like to thank a few of the ones that supported me over the four years of my Ph.D. and beyond. My first and deepest gratitude goes to Prof. Friedemann Mattern. His pioneering work on the Internet of Things inspired me profoundly and his guidance, support and interest in my research were great sources of motivation.

Even if this thesis reflects my views and research, it is the outcome of a collaborative effort. Instrumental to this thesis were the fruitful collaborations with Erik Wilde at UC Berkeley and in particular with Vlad Trifa at ETH Zurich. Vlad and I shared the most creative (and often funniest!) moments. Over the years, he became one of my closest friends and his support was key to my success. Thanks Vlad!

I had the chance to split my Ph.D. time between research at ETH and SAP Research Zurich where further collaborations and friendships emerged. At ETH for example, learning the art of user studies with my office-mate and friend Iulia Ion or working with Simon Mayer and his unlimited creativity for Web of Things prototypes. I also heartily want to thank my other colleagues at ETH: Robert Adelmann, Alexander Bernauer, Christian Beckel, Philipp Bolliger, Steve Hinske, Wilhelm Kleiminger, Matthias Kovatsch, Marc Langheinrich, Benedikt Ostermaier, Matthias Ringwald, Christof Roduner, Kay Römer, Silvia Santini, Gábor Sörös and Markus Weiss. I further want to extend my gratitude to all the students who contributed, piece by piece, to building the Web of Things as we present it in this thesis: Azu Aguilar, Bettina Dober, Mathias Fischer, Soojung Hong, David Karam, Mathias Mueller, Lukas Naef and Thomas Pham.

At SAP, I would first like to thank the SOCRADES and SENSEI team-members for our collaborations: Oliver Bäcker, Markus Eurich, Stefan Haller, Stamatis Karnouskos, Moritz Köhler, Luciana Moreira Sá de Souza, Frederic Montagut, Patrik Spiess, Domnic Savio and Claudia Villalonga. I also enjoyed the very spontaneous collaborations with Ivan Delchev, Felix von Reischach and Andreas Budde. I extend my thank you to all friends and colleagues from SAP Research and especially to my manager Uli Eisert for his valuable support.

I found a significant source of inspiration for my thesis in the world of product identifica-

tion and RFID. Hence, I am thankful to Prof. Elgar Fleich and his team for letting me the chance to work as a research associate of the Auto-ID labs in Zurich for more than a year. Furthermore, I want to express my deepest gratitude to Prof. Sanjay Sarma and his team who welcomed me at the MIT Auto-ID labs. They treated me as one of theirs from day one on and for the following six months of very inspiring discussions and fruitful collaborations.

I was also lucky enough to be able to count on several mentors. In particular, Christian Floerkemeier whose guidance first at MIT and then ETH was truly valuable. His constant motivation, insights in the business of the Internet of Things and pragmatic view of research helped me to a level he probably does not suspect. Furthermore, I need to thank my two long-term mentors Olivier Liechti and Patrik Fuhrer whose guidance, sharp advices and jokes were always a source of pure motivation. I also want to thank Jacques Pasquier, Hans Gellersen and Beat Hirsbrunner for their very valuable advices and support at several important steps of my career.

Building a community while working on this thesis was a tremendous experience and I would like to heartily thank all the contributors, community members and readers of the Web of Things blog[1] as well as all the organizers, PC members and participants of the WoT workshop series[2].

Last but not least I want to express my deepest gratitude to my family: my sister and best friend Véronique, my brother-in-law Steven, and to the rest of the family for their tremendous love and support over the years. To Rachel who, thanks to her love, patience and support, was the best partner a Ph.D. student could dream of. To my mother who always believed in me and motivated me to study. To my father Jean-Pierre whose humbleness, patience, dedication and taste for all things computers were and will always be a model for me. Thank you.

[1] See `www.webofthings.org`

[2] See `www.webofthings.org/wot`

Contents

Chapter 1

Introduction

Contents

1.1 Motivation

Pervasive and ubiquitous computing have a long-lasting tradition of looking into integration of physical objects with the digital world. Recent developments in the field of embedded devices have led to *smart things* increasingly populating our daily life, slowly but steadily forming interconnected networks of physical objects: Sensor nodes are networked together to create environmental monitoring applications, making cities smarter and dynamically adapting to their context [196]. Home appliances such as TVs, alarm clocks, digital picture frames and Hi-Fi systems can communicate with each other to offer integrated services such as cross-devices multimedia experiences, smarter HVAC (Heating, Ventilating, and Air Conditioning) systems or more energy aware and efficient homes [113, 91, 196, 133, 134]. RFID-tagged objects in stores and along supply-chains allow manufacturers, suppliers and service providers to optimizes their operations [16, 55]. Products get digital identities through barcodes or RFID tags and offer unprecedented business opportunities [56, 57, 16, 133].

To facilitate these connections, research and industry have come up with a number of low-power physical and transport network protocols such as Zigbee, Bluetooth, IEEE 802.15.4 or, more recently, low-power WiFi and 6LoWPAN [98, 99] [244]. However, while these

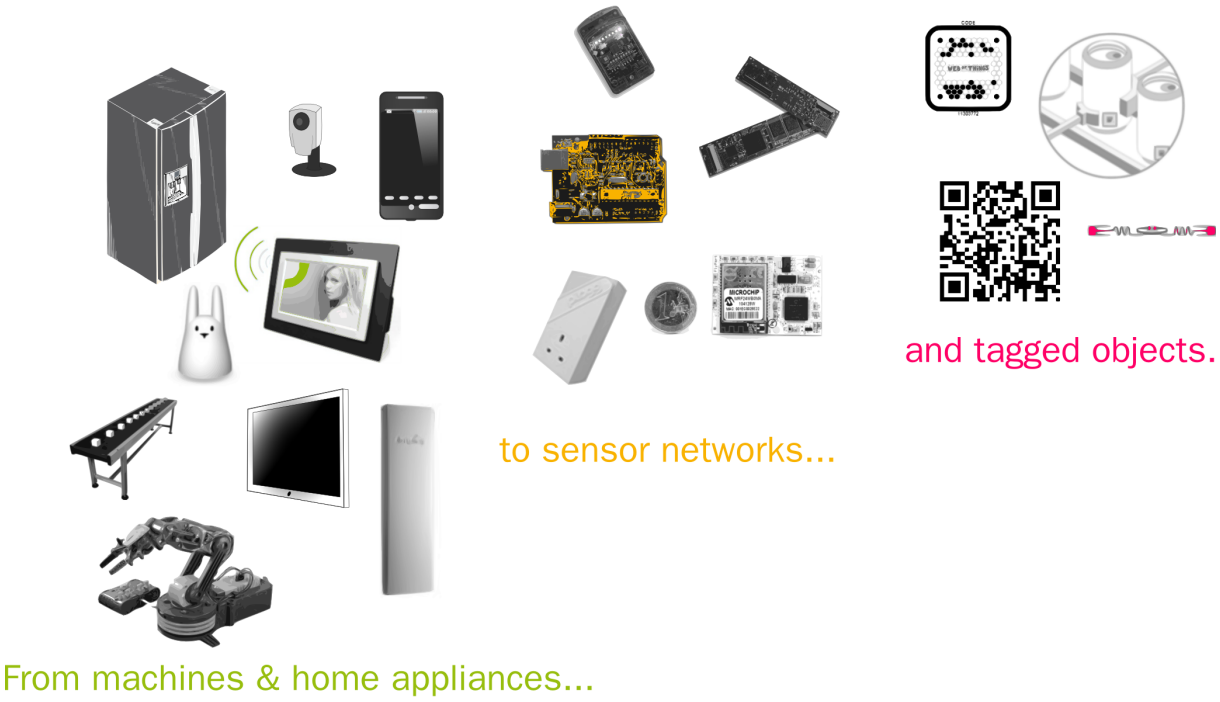


Figure 1.1: Smart things are digitally enhanced objects and devices that have communication capabilities. They range from machines and home appliances to wireless sensor and actuator networks as well as tagged every-day objects.

developments help towards integrating smart things at the network layer, at the application layer, embedded devices still form multiple, small, incompatible islands: Developing applications using them remains a challenging task that requires expert knowledge of each smart things platform [36, 161]. As a consequence, smart things remain hard to integrate into composite applications.

Several service platforms propose a standardized integrated architecture to facilitate the cross-integration of smart things. Standards such as UPnP and DNLA, Jini or OSGi successfully address concerns such as service discovery and registration. However, these systems are not fully compatible with one another and their complexity and lack of well-known tools let them only reach a relatively small community of expert developers [35]. Hence their direct usage for innovative applications (e.g., mobile or Web-based applications) has been rather limited to date.

In the world of enterprise computing, interoperability and loose-coupling at the business application layer is achieved using WS-* Web services [10]. WS-* Web services, sometimes called "Big Web services" [154], are based on two main XML formalisms: WSDL and SOAP as well as a set of additional standards (WS-Addressing, WS-Security, WS-Discovery, etc.). With the goal of facilitating the integration of smart things with applications, several research initiatives look at adapting these services to the real-world [157, 36, 104, 79, 161]. This research led to lighter forms of WS-* services targeted towards real-world applications such as DPWS (Device Profile for Web Services) [104] or DNLA, both direct evolutions of UPnP.

The ultimate goal of these initiatives can be summarized as trying to *create a loosely-coupled ecosystem of services for smart things*. That is, a widely distributed platform in which the services provided by smart things can be easily composed to create new applications and use-cases. As shown in several research projects, the WS-* approach is an improvement over the proprietary protocols traditionally used in this field [36, 104, 35], however the approach also has important shortcomings. First, WS-* standards are rather verbose and heavy in terms of required bandwidth, memory and CPU. This makes them challenging to implement on devices with limited resources [214, 154]. More importantly, despite their name, Web services actually use the Web as a transport layer and not as an application architecture [153] making them harder to use and integrate with the World Wide Web.

The Internet is a compelling example of a scalable global network of computers that interoperate across heterogeneous hardware and software platforms. On top of the Internet, the Web illustrates well how a set of relatively simple and open standards (e.g., HTTP, HTML, XML, JSON, etc.) can be used to build very flexible systems while preserving efficiency and scalability. The cross-integration and developments of composite applications on the Web, alongside with its ubiquitous availability across a broad range of devices (e.g., desktop computers, laptops, mobile phones, set-top boxes, gaming devices, etc.), make the Web an outstanding candidate for a universal integration platform [81].

Hence, as more and more devices are getting connected to the Internet, the next logical step is to use the World Wide Web and its associated technologies as a platform for smart things. In the *Web of Things (WoT)*, we are considering smart things as first-class citizens of the Web and position the WoT as a refinement of the Internet of Things (IoT) [133, 134, 170] by integrating smart things not only into the Internet (the network), but into the Web (the application layer).

To achieve this integration, we propose to reuse and adapt patterns commonly used for the Web. We embed Web servers [99, 42, 76] on smart things and apply the REST (Representational State Transfer) architectural style [50, 160] to the physical world. The essence of REST is to focus on creating loosely coupled services on the Web so that they can be easily reused [153]. REST is actually core to the Web and uses URIs for encapsulating and identifying services on the Web. In its Web implementation it also uses HTTP as a true application protocol. It finally decouples services from their presentation and provides mechanisms for clients to select the best possible formats. This makes REST an ideal candidate architecture to build a *universal API* for smart things. As the "client-pull" interaction model of HTTP does not fully match the needs of event-driven IoT applications, we further suggest the use of syndication techniques such as Atom and some of the recent real-time Web technologies to enable smart things push interactions.

In this thesis, we propose a Web of Things Architecture: a Web-based distributed application platform for smart things. As a consequence of the proposed architecture, smart things and their functionality get transportable URIs that one can exchange, reference on Web sites and bookmark. Smart things are also linked together enabling discovery simply by browsing. The interaction with smart things can also almost entirely happen in

a browser, a tool that is ubiquitously available and that most users are familiar with [110]. Applications can be built upon them using well-known Web languages and technologies. Furthermore, smart things can benefit from the mechanisms that made the Web scalable and successful such as caching, load-balancing, indexing and searching.

In this thesis, rather than looking at one particular challenge, we take a holistic view, looking also at the bigger picture. We propose a number of building-blocks towards creating a distributed deployment of smart things which fosters serendipitous re-use of smart things. Our goal is to create a participatory WoT where communities and users can create opportunistic applications, i.e., composite applications easily created by re-using existing services or devices. Just as people create Web mashups [90] involving Web 2.0 services, they should be able to create *Physical Mashups* [81] mixing services from the real and virtual worlds together.

1.2 Contributions

In this section we outline the main contributions of the thesis towards an Internet of Things supporting opportunistic applications or physical mashups. First, we present the Web of Things architecture. We then apply it systematically to two domains: Wireless Sensor and Actuator Networks and Auto-ID (Automatic Identification) networks.

1.2.1 The Web of Things: A Web-Oriented Service Platform for Smart Things

Our first contribution is the Web of Things Architecture. For this, we build on top of network connectivity and focus on the application layer. We take a systematic approach required to achieve the *mashability* of smart things with the Web. We identify four layers: *accessibility, findability, sharing and composition*. We study how each layer can be designed and implemented as a service on the Web using Web languages and patterns.

We begin by addressing *device accessibility*. We propose to use the REST architectural style [50] and study its applicability to smart things [81]. For this part, we build upon several projects in the field of ubiquitous computing [110, 39, 126] and look at a systematic application of the REST principles and their current Web implementation in HTTP to adapt the architecture to the needs of smart things. We then discuss two ways of integrating smart things to the Internet and the Web. Either directly or through the use of enhanced reverse proxies that we call Smart Gateways [193, 77]. As a result, smart things become easier to build upon. Popular Web languages (e.g., HTML, Python, JavaScript, PHP) can be used to easily build applications involving smart things and users can leverage well-known Web mechanisms (e.g., browsing, searching, bookmarking, caching, linking) to interact and share these devices. We illustrate this by means of a user study.

We then study the *findability* [144] of smart things: once they are connected to Web, how does one find the services they offer to integrate them into composite applications? In particular, we propose a discovery and lookup infrastructure for the Web of Things and describe smart things according to a metadata model that we implement using semantic annotations based on Microformats [254]. Furthermore, we propose, implement and evaluate extending user search queries for smart things services based on a new process [70] using related keywords extracted from services on the Web (e.g., Wikipedia, Yahoo Web Search, etc.) [77].

Using smart things in composite applications requires a scalable *sharing* mechanism that lets owners of smart things manage access control in a convenient and straightforward way. We introduce the Social Access Controller [71], a platform that relies on social networks (e.g., Facebook, LinkedIn, Twitter, etc.) and their open APIs (e.g., OpenSocial) to enable owners to leverage the social structures in place for sharing smart things with others.

Finally, we discuss the *composition* of smart things on the Web. We introduce the notion of *Physical Mashups* where services from the Web are serendipitously composed with services offered by smart things. We discuss a number of requirements towards a Physical Mashups Framework upon which Physical Mashup editors can be built and propose an implementation as an open service on the Web [68].

These four layers form the basis of our Web of Things Architecture. Not every Internet of Things application that is to be ported to the Web will require the four of them, but they present a model that can be used towards a looser-coupling, better and easier integration of the physical and the virtual worlds thanks to Web technologies.

1.2.2 Case Studies

Our second and third contributions are two domains in which we apply the Web of Things Architecture in three different case studies. We first look at the field of Wireless Sensor and Actuator Networks and then at Auto-ID networks.

Bringing Wireless Sensors and Actuators Networks to the Web

In the last decade, important progress in the field of embedded systems has given birth to a myriad of tiny computers to which virtually any type of sensors/actuators can be attached. By inter-connecting these devices using low-power wireless communication, a whole new world of possible applications is unveiled. Networks of physically distributed computers, usually called Wireless Sensor Networks (WSN), are valuable tools for monitoring the physical world [196]. Likewise the same type of devices can also be used for actuating the world, such as controlling security systems, traffic lights, etc. These networks are then called Wireless Sensor and Actuator Networks[1]

[1]We will subsequently use the term Wireless Sensor Network (WSN) for both, pure sensor and sensor/actuator networks.

Unfortunately, most projects using WSNs are based on different – and usually incompatible – software and hardware platforms [36, 104, 35, 6]. Within such an heterogeneous ecosystem of devices, the development of simple applications still requires special skills and a substantial amount of time [146]. Moreover, for each new deployment, a large amount of work must be devoted to re-implement basic functions and application-specific user interfaces, which is a waste of resources that could be used by developers to focus on the application logic. Ideally, developers should be able to quickly build applications only by recombining ready-made building-blocks.

Hence, the world of WSNs is an interesting case-study for the proposed approach as adopting a simple application architecture for these devices contributes to foster their wider usage and applicability. Our contribution here is to look at two WSN platforms and evaluate the validity of our approach by integrating them to the Web based on the Web of Things Architecture.

Facilitating General Purpose Sensing Applications We first look at an all purposes WSN platform called Sun SPOTs [273]. We design, implement and evaluate how each layer of the model can be leveraged to make the developments on top of this platform as accessible as simple Web development is. With this platform we also evaluate the performances when using Web protocols directly on WSNs using embedded Web servers.

Facilitating Energy Monitoring and Control Applications Rising global energy demand and the limitation of natural resources has led to increased thoughts on residential energy consumption. A necessary step towards energy conservation is to provide timely and fine-grained consumption information. This allows for users to identify energy saving opportunities and possibly adjust their behavior to conserve energy [135].

Currently available off-the-shelf products that depict the energy consumption in near real-time are helpful, but do not fully meet the user needs as they have high usage barriers and often require complex installations [60]. Furthermore, they are not able to provide the most compelling feedback [53] since they lack the ability to provide an appliance-specific break down of the energy consumption and are not able to compare the consumption of individual devices in an appealing manner. Finally, they do not meet the needs of software developers as they do not offer open APIs, and developing applications on top of them is rather cumbersome.

Hence, we propose an *easy to use, deploy and develop upon* device-level energy monitoring platform called *Energie Visible* [83, 81, 204]. It is based on an off-the-shelf smart power outlet that we seamlessly integrate into the Web by systematically applying the Web of Things Architecture. We demonstrate how this facilitates, on the one hand-side, the deployment and usage by home users and, on the other hand-side, the development of novel applications for developers. We present the design and implementation of a Web user interface. We further evaluate the suitability of our approach with the help of a pilot deployment and feedback from several developers using our framework in research

projects and prototypes such as a mobile phone energy monitoring application.

Facilitating the Development and Deployment of Distributed Auto-ID Applications

The RFID (Radio Frequency IDentification) [169, 52] standards community has developed a number of communication interfaces and software standards to provide interoperability across different RFID deployments. This extensive standards framework, known as the EPC (Electronic Product Code) Network [170], covers aspects such as reader-to-tag communication, reader configuration and monitoring, tag identifier translation, filtering and aggregation of RFID data, and persistent storage of application events. While there are in total fifteen standards that make up the EPC Network framework, the air interface protocol known as EPCglobal UHF Gen2 has seen the most adoption – both in large scale supply chain applications as well as niche RFID deployments.

The adoption of the software standards within the EPC Network has been significantly slower [172, 72]. The deployment of RFID applications that implement the EPC Network standards often remains complex and cost-intensive mostly because they typically involve the deployment of rather large and heterogeneous distributed systems. As a consequence, these systems are often only suitable for big corporations and large implementations and do not fit the limited resources of small to mid-size businesses and small scale applications both in terms of required skill-set and costs.

While there is most likely no universally available solution to these problems, the success of the Web in bringing complex, distributed and heterogeneous systems together through the use of simple design patterns appears as a viable approach to address these challenges. Thus, our contribution in this context is to study the pain points of RFID applications and systematically apply the Web of Things model [74, 72].

In particular, we show how Cloud Computing, RESTful interfaces and the real-time web as well as Physical Mashups can simplify application development, deployments and maintenance in a common RFID application. Our analysis also illustrates that RFID/EPC Network applications are an interesting fit for WoT technologies and that further research in this field can significantly contribute to making real-world applications in this domain less complex and cost-intensive.

1.3 Thesis Outline

The remainder of this thesis is structured as follow: Chapter 2 presents the Web of Things Architecture. We expose the required components for a successful Web-integration of smart things. We further propose a number of optional components that help to create a Web ecosystem in which Physical Mashups are made possible. In particular, we discuss device accessibility, findability, sharing and composition. In this part of the thesis, we also review an alternative architecture and present a user-study in which we build upon

the developers' experience to provide guidelines on making the right architectural decision for IoT projects.

In Chapter 3 we apply the architecture to the domain of Wireless Sensor Networks. In the first part we discuss the integration of a general purpose sensing platform. We then present the benefits of the Web of Things Architecture when applied to smart energy monitoring and control. Based on our open-sourcing of the platform, we also discuss a pilot deployment and illustrate the ease of use and integration that the Web of Things Architecture provides.

In Chapter 4 we show how the architecture can be applied to the Auto-ID and RFID domain. We apply it to several components of the EPC Network and illustrate, by means of prototypes and studies, how the Web of Things Architecture has the potential to simplify and foster developments in the RFID domain.

Finally, in Chapter 5 we provide an summary of our contributions. Taking a step back, we also discuss some of the open challenges and interesting future directions that we believe the Internet of Things and Web of Things domains will have to take.

Chapter 2

The Web of Things

"Any sufficiently advanced technology is indistinguishable from magic"

Arthur C. Clarke

Contents

In this chapter, we present the Web of Things Architecture. We describe the four layers it is based on: the Device Accessibility Layer, Findability Layer, Sharing Layer and Composition Layer as shown in Figure 2.1.

The overall goal of this architecture is to facilitate the integration of smart things with existing services on the Web and to facilitate the creation of Web applications using smart things. In particular, we formulate the following general requirements [77, 72] we would like for our architecture to fulfill:

1. It should *lower the entry barrier for developers* and foster rapid prototyping. This allows a wider range of developers, tech-savvy users (technologically skilled people) or researchers to develop on top of smart things and contributes to fostering third party (public) innovation using smart things.

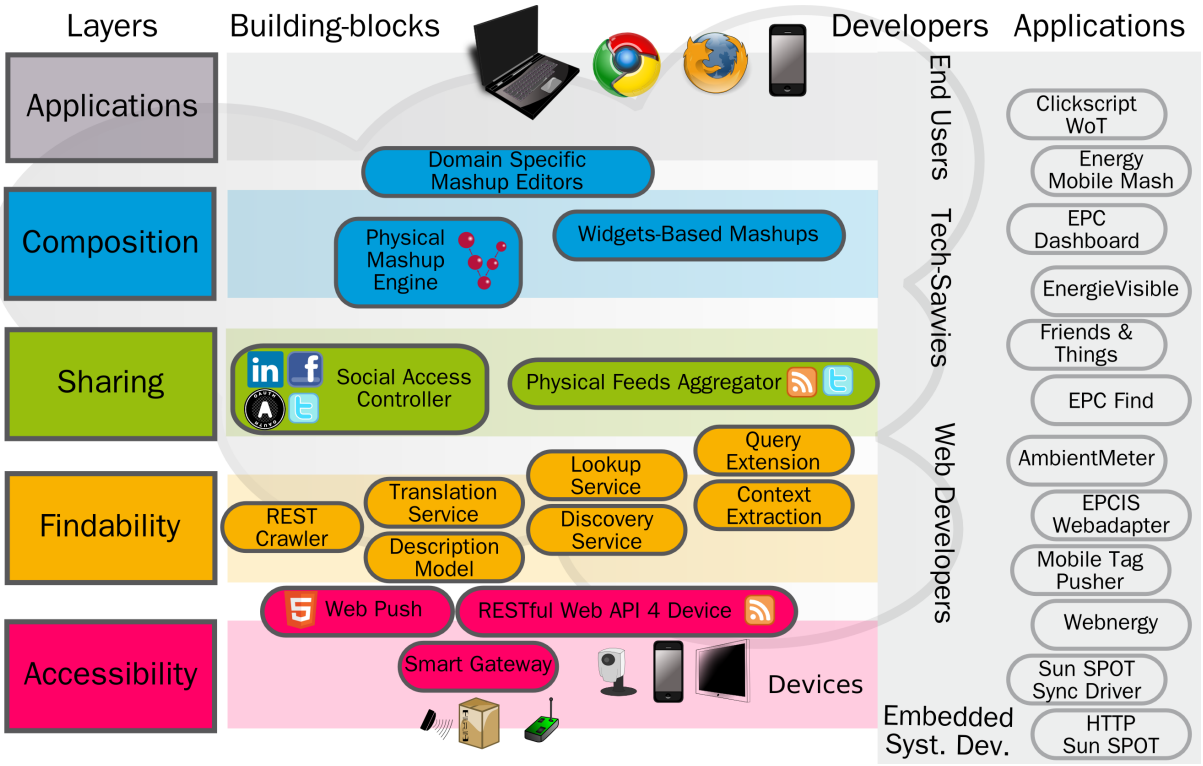


Figure 2.1: The four layers of the Web of Things architecture: Accessibility, Findability, Sharing, Composition. Applications can be built on top of each layer, but as we go up the layers they become more accessible to a broader community of developers and users. The figure provides an overview of the deliverables of this thesis. On the left-side are the architectural building-blocks. On the right-side the applications and prototypes.

2. It should *offer direct access for users*. Users should be able to access and use smart things without the need for installing additional software. From a Web browser (or an HTTP library in the case of a software client) they should further have means to directly extract, save and share smart things data and services. This ensures the usability of the architecture and minimizes the entry barriers for users.

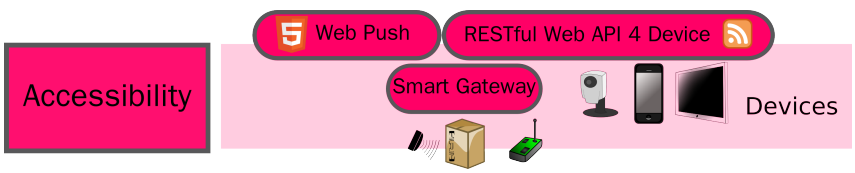
3. It should offer a *lightweight* access to smart things data. This enables creating applications in which real-world data are directly consumed by *resource-constrained devices* such as mobile phones or wireless sensor nodes without requiring dedicated software on these devices.

Unlike traditional layered architectures such as the OSI (Open Systems Interconnection) model [34], the layers in the presented Web of Things Architecture are not strictly defined and do not literally hide the previous layers. Rather, the architecture should be seen as an ecosystem of different services that ease, step by step, the creation of applications using smart things. The architecture proposes services that address each layer required to consider smart things as first-class citizen of the Web. However, applications can be built on top of the services offered by the implementation of each layer or on top of a combination of them depending on the requirements of a particular use-case.

As illustrated on Figure 2.1 the development of applications using smart things on top of their native operating systems, protocols and libraries still requires specific skills and is, for the greater part, only accessible to embedded systems experts [146]. The goal of each layer of the Web of Things Architecture is first to bring this development closer to Web developers and technically skilled hobbyists [90]. Then, it brings the usage and development of Internet of Things applications closer to end-users, enabling them to create simple applications tailored to their needs.

In this chapter, we describe each layer. Focusing first on the architecture of the components in these layers, we then look at the services and APIs they offer and propose implementations of these services. The proposed components are evaluated in a generic way in this chapter. These evaluations are complemented by Chapter 3 and Chapter 4, where we apply the architecture to two specific domains: Wireless Sensor Networks and RFID tagged-objects and evaluate it within these domains.

2.1 Device Accessibility Layer



In the first layer of our Web of Things Architecture (see Figure 2.1), we address the access to smart things: *How can we, from an application point of view, enable a consistent access*

to all kinds of connected objects?

Our proposal is to integrate things to the core of the Web, making them first-class citizens just as Web pages are. For this, we use the REST architectural style and its Web implementation. In the first part of this section, we illustrate how we can model the functionality and services of smart things using the RESTful principles. Then, we discuss the integration of devices that are not capable of connecting to the Internet. Finally, we propose a way for smart things to push data to clients rather than having them constantly polling data. Parts of this section have been published in [78, 81, 79, 76].

2.1.1 A Web API for Smart Things

We begin by briefly summarizing the principles of RESTful architectures. We then focus on how a systematic application of the RESTful principles to smart things leads to an API that can be consumed and understood by a broad number of clients.

REST in a Nutshell

Initially proposed by Roy Fielding in his Ph.D. dissertation [50], REST is an architectural style that was used as a set of guidelines to implement the second wave of Web standards and in particular HTTP 1.1 and URIs (Uniform Resource Identifiers). The goal of this second wave of standards was to move from a Web serving documents as of HTTP 0.9 to a Web as a true application layer wit HTTP 1.1. The REST guidelines where created to make sure that the new architecture would support "scalability of component interactions, generality of interfaces, the independent deployment of components as well as intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems" [50].

As such, REST is independent from the Web and can be implemented in other systems. However, in the remainder of this thesis we focus on the Web implementation of REST.

The central idea of REST revolves around the notion of resource as *any component of an application that needs to be used or addressed*. Resources can include physical objects (e.g., a temperature sensors) abstract concepts such as collections of objects, but also dynamic and transient concepts such as server-side state or transactions.

A system can be basically considered as RESTful if it respects the five following constraints [50]:

C1 *Resource Identification:* the Web relies on *Uniform Resource Identifiers (URI)* to identify resources, thus links to resources (C4) can be established using a well-known identification scheme.

C2 *Uniform Interface:* Resources should be available through a uniform interface with well-defined interaction semantics, as is *Hypertext Transfer Protocol (HTTP)*. HTTP

has a very small set of methods with different semantics (*safe*, *idempotent*, and others), which allows interactions to be effectively optimized. It also allows for a clean decoupling of the interface (RESTful interface) and the actual service implementation. Unlike in WS-*, where methods (also known as service operations) take arbitrary names and semantics, in HTTP, the uniform interface has 5 main methods:

1. `GET` is used to retrieve the representation of a resource.
2. `PUT` is used to update the state of an existing resource or to create a resource by providing its identifier.
3. `DELETE` is used to remove a resource.
4. `POST` creates a new resource.

While these verbs definitely cover very well CRUD (Create Read Update Delete) types of applications, they are also supposed to explicit every action a client can execute on a resource, whatever the type of application is.

C3 *Self-Describing Messages:* Agreed-upon resource representation formats make it much easier for a decentralized system of clients and servers to interact without the need for individual negotiations. On the Web, media type support in HTTP and the *Hypertext Markup Language (HTML)* allow peers to cooperate without individual agreements. For machine-oriented services, media types such as the *Extensible Markup Language (XML)* and *JavaScript Object Notation (JSON)* have gained widespread support across services and client platforms. JSON is a lightweight alternative to XML that is widely used in Web 2.0 applications and directly parsable to JavaScript objects.

C4 *Hypermedia Driving Application State (Connectedness):* Clients of RESTful services are supposed to follow links they find in resources to interact with services. This allows clients to "explore" a service without the need for dedicated discovery formats, and it allows clients to use standardized identifiers (C1) and a well-defined media type discovery process (C3) for their exploration of services. This constraint must be backed by resource representations (C3) having well-defined ways in which they expose links that can be followed.

C5 *Stateless Interactions:* This requires requests from clients to be self-contained, in the sense that all information to serve the request must be part of the request. HTTP implements this constraint because it has no concept beyond the request/response interaction pattern; there is no concept of HTTP sessions or transactions. It is important to point out that there might very well be state involved in an interaction, either in the form of state information embedded in the request (HTTP cookies), or in the form of server-side state that is linked from within the request content (C3). Even though these two patterns introduce state into the service, the interaction itself is completely self-contained (does not depend on the context for interpretation) and thus is stateless.

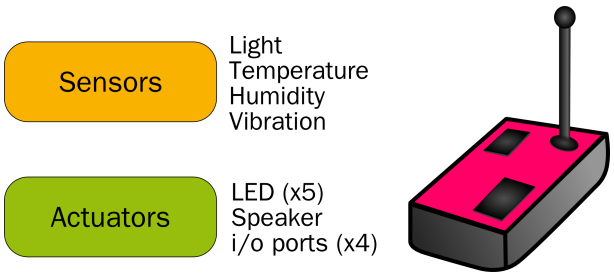


Figure 2.2: An generic sensor node, offering a number of sensors and actuators.

Tying together C2 and C3, HTTP also supports *content negotiation*, allowing both clients and servers to communicate about the requested and provided representations for any given resource. Since content negotiation is built into the uniform interface, clients and servers have agreed-upon ways in which they can exchange information about available resource representations, and the negotiation allows clients and servers to choose the representation that is the best fit for a given scenario.

The seminal work on REST [50] on which these guidelines are based, presents REST as a meta-architecture [160]. This offers the advantage of being able to use the thesis as a set of tools for judging how RESTful systems are but it lacks practical guidelines on how to actually implement a RESTful system on the Web. In [160], Leonard Richardson and Sam Ruby propose the concept of ROA or Resource Oriented Architectures, which is a Web architecture that can be used to create loosely-coupled services on the Web.

The design goals of ROAs and their advantages for a decentralized and large-scale service architectures align well the field of pervasive computing: millions to billions of available resources and loosely coupled clients, with potentially millions of concurrent interactions with one service provider. Based on these observations, we argue that RESTful architectures are the most effective solution for the Web of Things, as they scale better and are more robust than RPC-based architectures such as WS-* Web services.

In the next sections we illustrate how the 5 constraints of REST as well as the concept of Resource Oriented Architectures can be applied an adapted to fit the requirements of a global and distributed ecosystem of smart things offering a comprehensive and interoperable service layer.

RESTful Things: A Resource Oriented Architecture for Things

The WoT can be realized by applying principles of Web architecture, so that real-world objects and embedded devices can blend seamlessly into the Web. Instead of using the Web as a transport infrastructure we aim at making devices an integral part of the Web and its infrastructure and tools by using HTTP as an application layer protocol. In this section, we describe the use of REST as a universal interaction architecture, so that interactions with smart things can be built around universally supported methods. We describe the process of Web-enabling smart things into four main steps:

1. Resource Design: identify the functionality or services of a smart thing, organize the hierarchy of these services and link them together, fulfilling constraints C1 and C4.
2. Representation Design: decide which representations will be served for each service, fulfilling constraint C3.
3. Interface Design: decide on the actions allowed for each service, fulfilling constraint C2 and C5.
4. Implementation Strategy: choose a strategy to integrate the smart things to the Internet and the Web, either directly or through a Smart Gateway.

In the following, we provide a set of guidelines to Web-enable smart things based on these four main steps. As case study, we describe how it can be used to bring wireless sensor nodes to the World Wide Web. The abstract sensor node we use as an illustration is shown in Figure 2.2.

Resource Design: Modeling Functionality as Linked Resources

As mentioned before, the central idea of REST revolves around the notion of resources. In our context, a resource is any component of an application that is worth being uniquely identified and linked to. On the Web, the identification of resources relies on Uniform Resource Identifiers (URIs), and representations retrieved through resource interactions contain links to other resources, so that applications can follow links through an interconnected web of resources. Clients of RESTful services are supposed to follow these links, just like one browses Web pages, in order to find resources to interact with. This allows clients to *explore* a service simply by browsing it, and in many cases, services will use a variety of link types to establish different relationships between resources.

Resource Identification In the Web of Things we have several levels of resources. While some of them represent physical objects, others are virtual only. Resources on the Web are often organized in a hierarchy, the hierarchical way of organizing and linking resources is also very relevant in the physical world and can be used as a basis to identify the resources of a smart thing. As an example, from the abstract sensor node in Figure 2.2 we can extract resources as shown in Figure 2.3. From this hierarchy we understand that each node has sensors (light, temperature, etc.), actuators (speakers, LEDs, etc.). Each of these components is modeled as a resource and assigned a URI which is deduced from the name of the current resource and its predecessors in the hierarchy. For instance, the light sensor gets the URI: `/generic-nodes/1/sensors/light`.

In an HTTP context, these identifiers or URIs are also known as URLs. However, since the term URL has been officially deprecated we use URI in the remainder of this thesis. Nevertheless, the widespread use of the term URL lead to a contemporary definition in which “a URL is a type of URI that identifies a resource via a representation of its primary

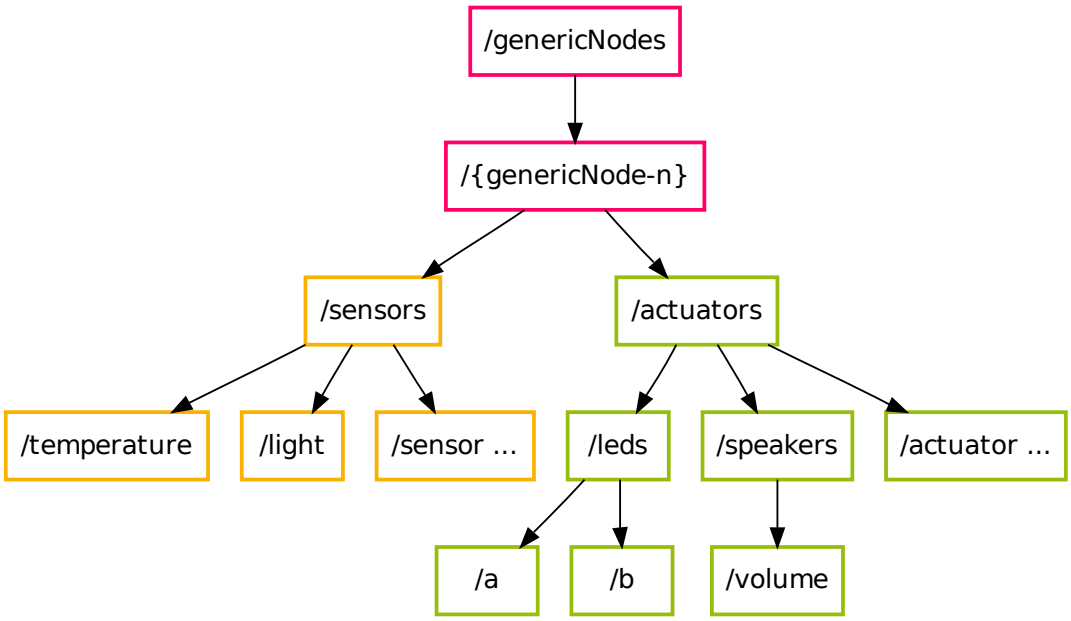


Figure 2.3: An example of resource hierarchy deduced from the abstract sensor. This hierarchy forms a tree where each resource has $0..n$ child resources.

access mechanism (e.g., its network location), rather than by some other attributes it may have" [278]. On the Web a URL is a URI beginning with the `http:` scheme and resolvable through the HTTP protocol.

We can form the absolute URI (or URL) of a smart thing's resource by adding a protocol scheme and root domain to the identifier. An important and powerful consequence of this is the addressability and portability of resource identifiers: They become unique (Internet or Intranet-wide, depending on the domain-name or assigned IP address) and can be resolved by any HTTP library or tool (e.g., a browser), bookmarked, exchanged in emails, instant messaging tools, encoded in QR-codes (Quick Response), etc.

For instance, typing a URI such as:
`http://<DOMAIN>:<PORT>/generic-nodes/1/sensors/light`
in a browser requests a representation of the resource light of the resource sensors of generic-node number 1.

Since there are no rules on the semantics of resources' identifiers, we cannot deduce a strict rule for naming physical resources. However, we suggest naming them according to two simple guidelines:

1. Use descriptive names: as the resource names appear in the URIs using names with some semantic value can be of great help to developers and users.

2. Use the plural form for aggregate resources: for instance if a smart thing has several sensors, then there should be a parent resource called `sensors` from which every

sensor is accessible with hyperlinks.

Linking A resource should also provides links back to its parent and forward to its children as well as to any related resource. As an example, the resource `generic-nodes/1/sensors/` provides a list of links to all the sensors offered by generic-node 1. This interlinking of resources that is established through both, resource links and hierarchical URI, is not strictly necessary, but well-designed URIs make it easier for developers and users to "understand" resource relationship and even allow non-link based ad-hoc interactions, such as hacking a URI by removing some structure and still expecting for it to work somehow. In some browsers this URI hacking is even part of the UI, where a "go up" function in the browser simply removes anything behind the last slash character in the current URI and expects that the Web site will serve a useful representation at that guessed URI.

Links are very important in Resource Oriented Architectures since they help clients to *discover* related resources. Using these links the client can discover other related services, either by browsing in the case of a human client or by crawling in the case of a machine. Thus, links in resource oriented architectures fulfill the constraint (C4) and enable the dynamic discovery of resources.

However, as we will see below, resources are not bound to a particular format but can be served using several formats. When a client requests an HTML representation then representing links is very straightforward as HTML has a standard mechanism for specifying links. With other formats such as JSON, however, there is no single standard format for providing links.

One could argue that the client can always fall back to an HTML representation when it is interested in related resources. However, this is inefficient in terms of HTTP calls required for a request which is especially relevant in resource-constrained environments such as the Web of Things. A good practice is thus to embed links consistently across all provided representations. One shortcoming of this approach is that the lack of standard link representation in formats such as JSON leads to a tighter coupling between the client and provided services.

Representation Design: Formatting the Resources

Resources are abstract entities and are not bound to any particular representation. Thus, several formats can be used to represent a service of a smart thing. However, agreed-upon resource representation formats make it much easier for a decentralized system of clients and servers to interact without the need for individual negotiations.

On the Web, media type support in HTTP and the Hypertext Markup Language (HTML) allow peers to cooperate without individual agreements. It further allows clients to navigate amongst the resources using hyperlinks. For machine-to-machine communication,

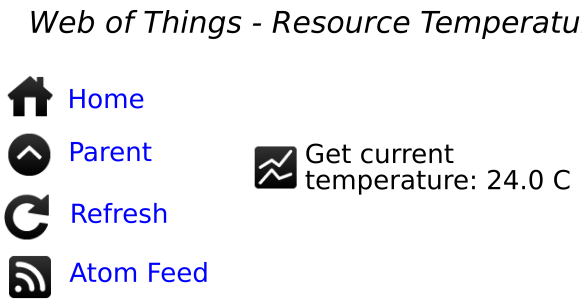


Figure 2.4: HTML representation (as rendered by a Web browser) of the temperature resource of a sensor node containing links to parent and related resources.

other media types, such as XML and JSON have gained widespread support across services and client platforms.

In the case of smart things, we suggest support for at least an HTML representation to ensure browsability by humans. Note that since HTML is a rather verbose format, it might not be directly served by the things themselves, but by intermediate reverse proxies, called Smart Gateways and described in Section 2.1.2.

For machine-to-machine communications, we suggest using JSON. Since JSON is a more lightweight format compared to XML, both is terms of message size and parsing time [212], it is better adapted to devices with limited capabilities such as smart things. Furthermore, it can directly be parsed to JavaScript objects. This makes it an ideal candidate for integration into Web Mashups and thus for creating physical mashups (see Section 2.4).

In the example of our generic-sensor, each resource provides both, an HTML and a JSON representation. As an example, Listing 2.1 shows the JSON representation of the temperature resource and Figure 2.4 shows the same resource represented as an HTML page with links to parents, subresources, and related resources.

```
{"resource":
  {"methods":["GET"],
  "name":"Temperature",
  "links":["/feed", "/rules"],
  "content":
  [{"description":"Current Temperature",
  "name":"Current Ambient Temperature",
  "value":"24.0",
  "unit": "celsius"}]}
}
```

Listing 2.1: JSON representation of the temperature resource of a generic node

Interface Design: Servicing Through a Uniform Interface

In REST, interacting with resources and retrieving their representations all happens through a uniform interface which specifies a service contract between the clients and

servers. The uniform interface is based on the identification of resources, and in case of the Web, this interface is defined by the HTTP protocol. We focus on three particular parts of this interface that can be used to model a smart thing's API: operations, content-negotiation, and status codes.

Operations on Resources As mentioned before, HTTP provides five main methods to interact with resources, often also referred to as *verbs*. Constraining operations to these methods is one of the keys to enable loose-coupling of services, as clients only need to support mechanisms to handle these methods [153].

In the Web of Things, these operations map rather naturally, since smart things usually offer quite simple and atomic services. As an example:

- `GET` can be used to retrieve the current consumption of a smart meter.
- `PUT` can be used to turn an LED on or off.
- `POST` can be used to create a new feed used to trace the location of an RFID tagged object.
- `DELETE` can for example be used to delete a threshold on a sensor or to shutdown a device.

More concretely, as an example, a `GET` on `/generic-nodes/1/sensors/temperature` returns the temperature observed by node 1, i.e., it retrieves the current representation of the temperature resource. A `PUT` on `/generic-nodes/1/actuators/leds/1` with the updated JSON representation {"status":"on"} (which was first retrieved with a `GET` on `/leds/1`) switches on the first LED of the node, i.e., it updates the state of the LED resource. A `POST` on `/generic-nodes/1/temperature/rules` with a JSON representation of the rule as {"threshold":35} encapsulated in the HTTP body, creates a rule that will notify the caller whenever the temperature is higher than 35 degrees, i.e., it creates a new rule resource without explicitly providing an identifier. Finally, a `DELETE` on `/generic-nodes/1` is used to shutdown the node, or a `DELETE` on `/generic-nodes/1/sensors/temperature/rules/1` is used to remove rule number 1.

Additionally, another less-known verb is specified in HTTP and implemented by most Web servers: `OPTIONS` can be used to retrieve the operations that are allowed on a resource as well as metadata about invocations on this resource. In a programmable Web of Things, this feature is very useful, since it allows applications to find out at runtime what operations are allowed for any URI. As an example, an `OPTIONS` request on `/generic-nodes/1/sensors/humidity/rules` returns `GET`, `POST`, `OPTIONS` as shown in the full HTTP response in Listing 2.2.

```
1 HTTP/1.1 200 The request has succeeded
2 Content-Length: 0
3 Allow: GET, POST, OPTIONS
4 Date: Tue, 19 Apr 2011 12:17:42 GMT
5 Accept-Ranges: bytes
```

```
6 Server: Noelios-Restlet-Engine/1.1.7
7 Connection: close
```

Listing 2.2: HTTP response of an `OPTIONS` request on a resource. It informs the client about the operations (`GET` and `POST`) available for the resource.

Content Negotiation Since resources are representation agnostic there is a need for clients and servers to be able to negotiate the right format for the right purpose. As a consequence, HTTP specifies a mechanism for clients and servers to communicate about the requested and provided representations for any given resource; this mechanism is called content negotiation. Since content negotiation is built into the uniform interface of HTTP, clients and servers have agreed-upon ways in which they can exchange information about requested and available resource representations, and the negotiation allows clients and servers to choose the best representation for a given scenario. For the abstract-node, a content negotiation message exchange looks as follows. The client begins with a `GET` request on `/generic-nodes/1/temperature/rules`. It also sets the `Accept` header of the HTTP request to a weighted list of media types it understands, for example to: "`application/json;q=1, application/xml;q=0.5`". The server then tries to serve the best possible format it knows about and specifies it in the `Content-Type` of the HTTP response. In our case, the generic-node cannot offer XML and would thus return a JSON representation and set the HTTP header to `Content-Type: application/json`.

While this is the standard way of negotiating a representation in HTTP, it has two drawbacks when implemented. First, it is unfortunately not implemented evenly by all the Web servers. More importantly, it encapsulates the required format in the HTTP packet directly and does not expose it to the users. Since the required format is a key parameter, we suggest supporting content negotiation directly in the URI as well in order to make it more natural for everyday users as well as directly testable and bookmarkable.

Thus, requests such as `/generic-nodes/1/sensors/temperature.json` should be supported as well and should return the temperature resource in the JSON format as shown in Listing 2.1. In case the smart thing does not accept this format it should return the closest possible format (e.g., XML in this case). Furthermore, it should set the appropriate response header: `Content-Type: application/json` just as with standard content negotiation.

Status Codes HTTP also offers a way of expressing errors and exceptions. Indeed, the status of an HTTP response is represented by standardized status codes sent back as part of the header in the HTTP response message. There exist several dozens of codes which each have well-known meanings for HTTP clients, these codes and their meanings are listed in the specification of HTTP 1.1 [51]. Furthermore, in [160] these codes are analyzed and explained in the context of ROAs with valuable examples.

In the Web of Things, these codes a very valuable since they provide a lightweight but yet powerful way of notifying abnormal and successful request execution.

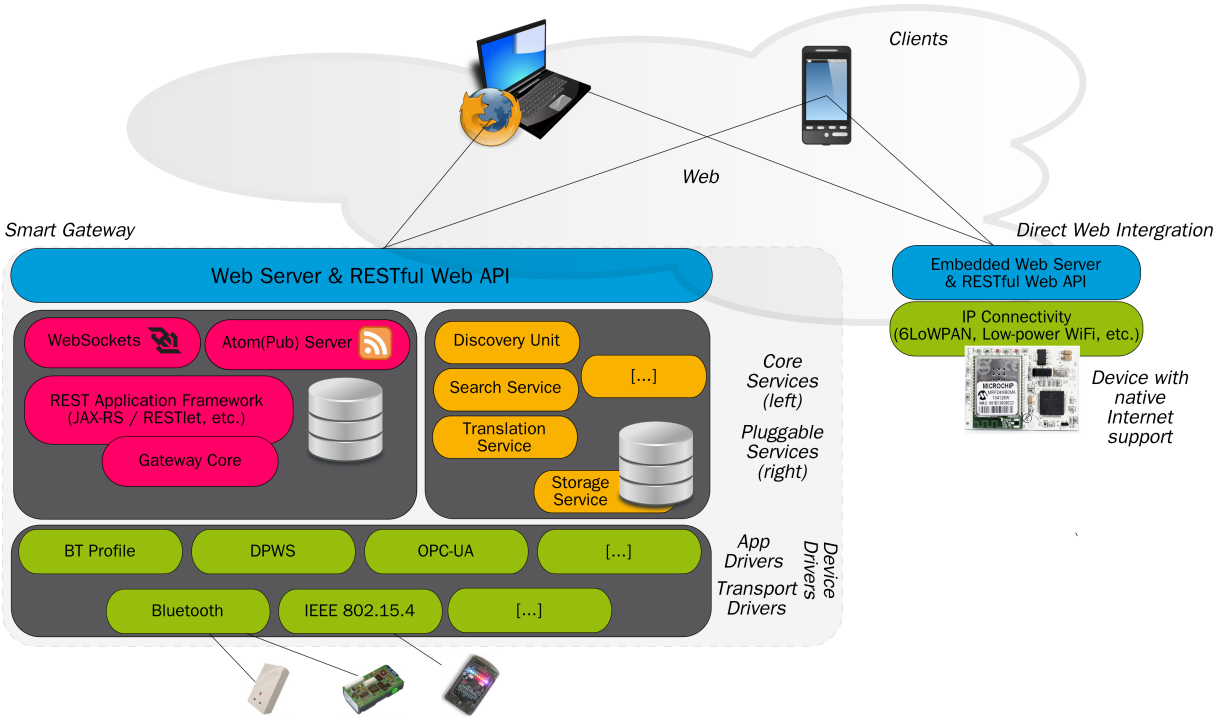


Figure 2.5: Web and Internet integration with Smart Gateways (left), direct integration (right). The Smart Gateways are small software application servers containing: Device Drivers to understand the low-level smart things, core services to create Web-APIs, pluggable services to offer additional functionality and a Web server.

As an example, a `POST` request on `/generic-nodes/1/sensors/humidity` returns a `405` status code. The client understands the status code as the notification that "the method specified in the request is not allowed for the resource identified by the request URI".

A concrete example of mapping domain-specific exceptions to Status Codes is provided in Chapter 4.3.2 where RFID exceptions are mapped to HTTP status codes.

2.1.2 Implementation Strategy: Connecting Things to the Internet

For a device to be part of the Web of Things, there are two basic requirements:

1. Implementation of the TCP/IP protocols ideally over a IEEE 802 (Ethernet) or IEEE 802.11 (WiFi) network.

2. Implementation of a Web server supporting the HTTP 1.1 protocol.

While an increasing number of embedded devices are supporting these two requirements natively, not all of them do, mainly because their computational, memory and communication bandwidth are too limited. Hence, in this section we propose two alternatives to integrate smart things to the Internet and the Web.

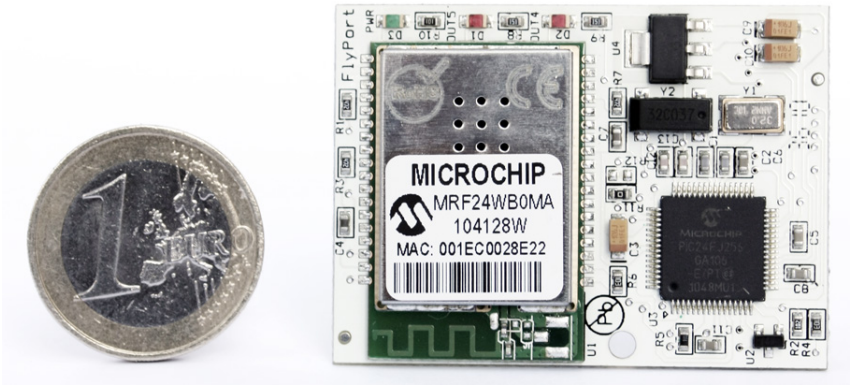


Figure 2.6: The Flyport embedded device offers a low-power WiFi module, full TCP/IP support and a native Web server supporting HTTP 1.1. (Reproduced with the kind authorization of OpenPicus, `www.openpicus.com`)

Native Internet and Web support

Research has shown that TCP/IP stacks can be implemented to meet the constraints of embedded devices. In [40] Dunkels implemented a full TCP/IP stack for 8 bits embedded devices with a footprint in the order of 10 kilobytes. More recent developments worked on adapting the IPv6 protocol to meet the energy constraints of these devices and proposed the 6LoWPAN [99] architecture. Similarly, an increasing number of sensor nodes and embedded devices are equipped with native low-power WiFi support (over IEEE 802.11) modules and embedded HTTP servers. This makes them seamlessly integrated to the Internet.

Previous work has also shown that embedding Web servers on resource and energy constrained devices is feasible [42, 123, 79]. Hence, it is a reasonable assumption that smart things will increasingly understand and implement the TCP/IP and HTTP 1.1 protocols. As an example, the off-the-shelve FlyPort shown in Figure 2.6 is an embedded device from the Openpicus open-source project [258]. It features a low-power WiFi module with full TCP/IP support and a Web server implementing HTTP 1.1. Similarly, the RN-131 nodes from Roving Networks [269] have TCP/IP over IEEE 802.11 connectivity. With world-wide consortia of industrial key-players appearing such as the IPSO Alliance [244], it is very likely that most of the future devices will have all the required elements with no need to translate HTTP requests from Web clients into the appropriate protocol for the different devices, as shown in the right part of Figure 2.5.

For these types of smart things to be truly part of the Web, their functionality should be available through a RESTful interface, i.e., they should implement the Device Accessibility Layer of the presented Web of Things Architecture.

Reverse Proxies: Smart Gateways

However, not all things can fulfill the requirements for TCP/IP and HTTP support. Indeed, for a number of smart things, these protocols are too demanding in terms of computation, memory, required bandwidth or battery life. As an example, it will probably take years until RFID tags will be powerful enough to implement these protocols and even then, it is unlikely for tags to communicate directly over IEEE 802.11. Similarly, for some sensor networks, ultra-optimized communication is a requirement and in these terms, dedicated low-power protocols such as Zigbee (over IEEE 802.15.4) or Bluetooth (over IEEE 802.15.1) or Ultra-Wideband (UWB, over IEEE 802.15.3) [120] with dedicated transport protocols are a better choice, even if native TCP/IP is increasingly being supported on some of these platforms, e.g., for IEEE 802.11 through 6LoWPAN [99].

Hence, when native TCP/IP and HTTP support is not possible or not desirable, we suggest that Web integration takes place using a software bridge. On the Web, similar bridges are called *reverse-proxies*. A reverse proxy takes requests from the Internet and forwards them to servers in an internal network. Reverse proxies have various interesting features, first they basically hide the internal network to the clients on the Internet. As a consequence, they can operate on the requests before they actually reach the services and are used for caching and load-balancing in several service oriented architectures.

For the Web of Things, we suggest taking a similar approach and propose the concept of *Smart Gateways* [193, 76, 81, 77] to capture the fact that it is an application level component that does more than only data forwarding. A smart thing basically hides the (proprietary) low-level protocols that smart things natively use and make them available on the Internet through TCP/IP support and on the Web through a Web server. From the Web clients' perspective, the actual Web-enabling process is fully transparent, as interactions are based on HTTP in both cases.

System Architecture As shown in the left-most part of Figure 2.5, a Smart Gateway is a software component composed of three basic layers. First, core to the concept of Smart Gateways are *Device Drivers*. Indeed, a smart thing can support several types of devices through a driver architecture as shown in Figure 2.5 where the gateway supports three types of devices and their corresponding communication protocols. To maximize re-usability, a Device Driver should be composed of two different software components: a *Transport Driver* and an *Application Driver*. The Transport Drivers are responsible for providing an API to communicate through a particular protocol such as IEEE 802.15.4 or Bluetooth. On top of these, the Application Drivers are responsible for the sometimes proprietary service protocols of devices. As an example, a Device Driver for an energy metering sensor node, would be composed of a Bluetooth Transport Driver (that can be reused for other devices) and an Application Driver that understands the proprietary service or application protocol of the node.

Application drivers are in charge of mapping the functionality of a device to a RESTful API. For this, they use a *REST Application Framework* which provides methods for

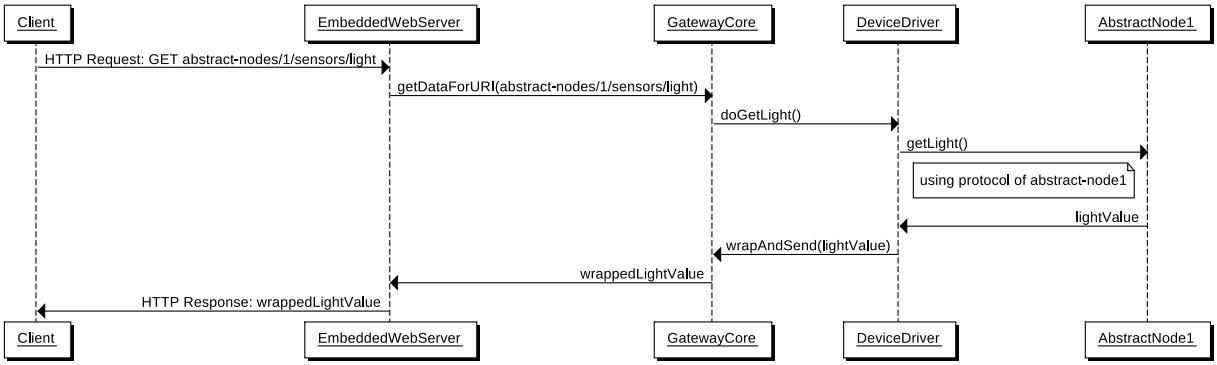


Figure 2.7: Simplified sequence diagram of the interaction between a client and a smart thing through a Smart Gateway. The Smart Gateway framework delegates the invocation to the corresponding Device Driver and takes care of converting the results into the appropriate format (e.g., JSON) and wrapping them into an HTTP packet returned to the client.

binding URIs to functionalities of the proxied smart things and formatting the responses using Web representations such as JSON or HTML.

Closely bound with the REST Application Framework, the *Embedded Web Server* serves the service requests through the HTTP protocol. Ideally, it should also feature an Atom-server (or at least Atom representations) and be a non-blocking Web server with support for HTML5 WebSockets (see Section 2.1.3).

Through these components, clients can use HTTP for requesting services on non-IP and non-Web devices. As an example, consider a request to an energy sensor node coming from the Web through the RESTful API to: `/energy-nodes/living-room/consumption.json`. The request is captured by the Embedded Web Server of the Smart Gateway unmarshalled into an object and further sent to the method previously bound to `/living-room/consumption`. This method is located in the Device Driver representing this particular energy sensor node. The Device Driver then translates the request in the appropriate format (e.g., a Bluetooth service call) to the Transport driver. The response is then transmitted back to the Device Driver which uses the REST Application Framework to marshal it into a Web format and further transmit it back to the embedded Web server. This process is summarized in Figure 2.7.

Software Implementation We implemented several Smart Gateways. Our first implementation was based on a small foot-print C++ software that was used to Web-enabled sensor nodes capable of measuring electricity consumption [76] (see also Section 3.2). Based on these implementations we realized that a lot of the written code could be re-used to Web-enable other smart things. We discuss two important points to foster the rapid integration of new devices and functionality to the Smart Gateways:

Modular Device/Application Drivers First, modular Device and Application Drivers prevents smart things integration from becoming too complex and fosters re-using standard features (e.g., Bluetooth communication or binding a URI to a method). To ensure

a high degree of modularity of these drivers, we implemented them using Java and in particular the OSGi framework [260]. OSGi is a modularization system built on top of Java that fosters re-usability through *bundles* [87]. Particularly interesting, is the concept of *Declarative Services* which facilitate the integration of different bundles. One of the big advantages of these service declarations is the ability to load a new (unknown) Bundle at run-time and having other components directly using it.

As a consequence, Device Drivers can be injected in a running Smart Gateway. This enables for instance the dynamic and remote injection of drivers to support new types of devices in an ad-hoc manner. Alternatively, the concept of Java Enterprise Application Servers [66], such as for example the Glassfish Application Server [236] can be used to create component-based Smart Gateways. Indeed, the latest generations of these applications servers has become more lightweight and components can be injected in a managed run-time environment that features a Web server.

Automatic Generation of Web Boilerplate Code Furthermore, we realized that a lot of the code necessary to generate these Drivers (and OSGi) bundles and the mappings from Web resources to methods in Java code could be automated. Hence, in the *AutoWoT* project [138] we propose a toolkit that enables developers to easily create new Device Drivers compatible with our Smart Gateway architecture. The toolkit was open-sourced and is available online [224].

An editor let's developers specify the resources of a smart thing in a visual manner. The editor then generates an XML description of the resource tree that is used to generate the interfaces and OSGi-specific code to create a driver. Then, all the developer has to do is to fill the callback methods triggered whenever a Web resource is invoked, with the smart things specific code, e.g., implementing the `doGetTemperature()` method called when a client invokes a `GET` request on `/temperature` in order for it to get the temperature data from the sensor node.

Deployment Ideally Smart Gateways should have a small memory footprint to be integrated into embedded computers already present in the infrastructure. For instance, when used to provide access to smart things in a home or office environment, the Smart Gateways can be deployed on devices such as Wireless routers or Network Attached Storage (NAS). Our implementation of the Smart Gateway OSGi software was tested successfully on MicroClients SR from Norhtec [257] featuring 500Mhz CPUs and 512 Mo of RAM each as well as an Asus WL-500gP Wireless router running the OpenWRT embedded Linux distribution [259].

2.1.3 Pushing Data from Smart Things and Smart Gateways

HTTP was designed as a client-server architecture, where clients can explicitly request (pull) data and receive it as a response. This makes REST and HTTP *well suited for*

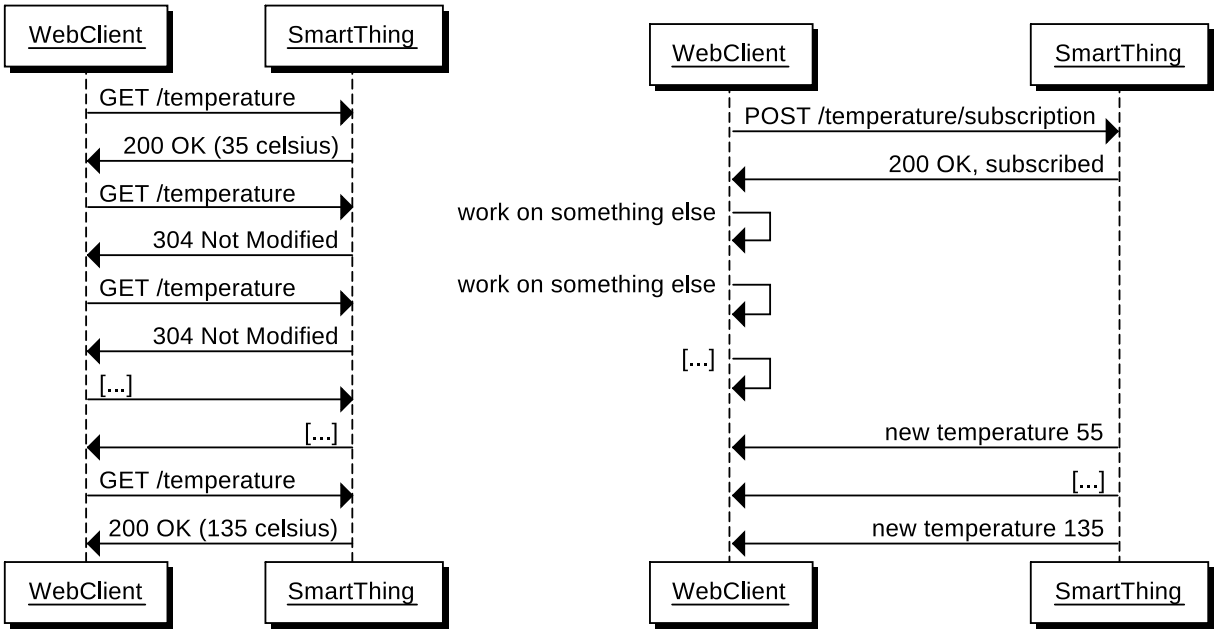


Figure 2.8: Sequence diagram of the communication between a client and a smart thing. On the left a traditional HTTP client-pull communication is started. The client has to constantly pull the smart things for updates. On the right a real-time Web approach is taken where the client is informed about the changes by the smart things.

controlling smart things, but this client-initiated interaction models seems unsuited for event-based systems, where data must be sent asynchronously to the clients as soon as it is produced.

This type of interaction is not really natural for some smart things applications and especially for *monitoring* applications [41, 191]. Consider for instance a sensor node used to detect a fire condition. As shown in Figure 2.8, in the protocol proposed by HTTP 1.1, the client constantly has to request updates (a.k.a. *polling*). With this protocol, in the best case most requests end up with empty responses (`304 Not Modified`) as the temperature did not change. In the worst case, the server (i.e., smart thing) transmits the same data after each request. This is sub-optimal for two reasons: First, it generates a great number of HTTP calls and a great part of these calls are void. Since reducing the number of HTTP calls to the minimum is key in scaling Web sites [182], this model raises scalability issues when considering monitoring applications in which several clients are connecting to a smart thing. Beyond scalability, numerous HTTP calls have more important consequences in the case of smart things such as their relatively high energy consumption which is important in the case of embedded devices running on batteries [214].

Furthermore, although near real-time can be simulated by polling the smart things very regularly, a protocol in which the smart things could push asynchronously to the client as soon as a condition is met enables providing real-time information in the WoT. In this section we discuss three architectural enhancements that contribute to solving these issues, while making sure that the proposed mechanisms can be integrated with the Web.

Feeds of smart things

```
<feed xmlns="http://www.w3.org/2005/Atom">
  <title type="text">[Title of Aggregation]</title>
    <author><name>[Name of the SmartGateway]</name></author>
    <link href="[Parent]" rel="related" type="[Type of
       representation]"/>
    <link href="[Child1]" rel="related" type="[Type of
       representation]"/>
    <link href="[Child2]" rel="related" type="[Type of
       representation]"/>
    <id>[uuid]</id>
    <entry>
      <title>[Content Description]</title>
      <id>[Unique ID for this event]</id>
      <author><name>[ID of the Smart Thing]</name></author>
      <uri>[Root URI of the Smart Thing]</uri>
      <published>[Date and time of event]</published>
      <content type="[text|html|xml|...]"</content>
    </entry>
    <entry>
      [...]
    </entry>
</feed>
```

Listing 2.3: Example of usage of the Atom format for providing historical information about a smart thing.

With Atom[1], the Web has a standardized and RESTful model for interacting with collections, and the *Atom Publishing Protocol (AtomPub)* extends Atom's read-only interactions with methods for write access to collections. Because Atom is RESTful, interactions with Atom feeds can be based on simple `GET` operations which can then be cached. While initially created to aggregate content on the Web, feeds have two interesting features for the WoT: First, they allow to create aggregates of smart things monitoring information. As an example a feed could be created to aggregate all information about energy sensors in a particular location (e.g., a building). Then feeds contain historical information and can thus be used to get not only the latest value of a sensor but rather its values over a period of time. Event more advanced scenarios can be based on feeds supporting query features, but this is an active area of research and there are not yet any standards [210].

Using feeds to contain data provided by smart things is rather straightforward as shown in Listing 2.3 and thus feeds can be supported (through content-negotiation) as a representation for any resource or aggregate of resources of smart things.

More importantly, in the case of the WoT, feeds can also be used to decouple clients from the actual resources. Indeed, the task of creating feeds can be delegated to completely

[1] See `http://tools.ietf.org/html/rfc4287`

external AtomPub compliant servers. As an example, it can be outsourced to the Smart Gateways we introduced before.

However, using Atom feeds as a representation format does not provide a solution to the fact that Web clients have to poll the data. Instead of polling it directly from the sensors, they now have to poll Atom servers. It is worth noting, that new mechanisms such as Pubsubhubbub (PuSH) [264] propose protocols to push feed updates back to the clients. However, these protocols requires additional infrastructure nodes (called hubs) and additional libraries on the client-side.

HTTP Callbacks (Web Hooks)

The most straightforward way to allow pushing to clients on the Web is to transform them into servers. This technique is often referred to as *Web Hooks* or *HTTP Callback* and is a very simple mechanism that can be used to have smart things pushing information to Web clients.

First, the client has to *subscribe* to a resource, for instance the energy consumption resource of a smart meter, it does so by `POST`ing a message to the `/subscribe` resource of the smart meter, alongside with a callback URI and usually a threshold (e.g., > 50 Watts). As a result, the smart things will `POST` data to the Web client whenever the threshold is met.

However, a very important issue with such a mechanism is that it places a rather hard constraint since every client also has to become an HTTP server. This constraint prevents clients such as Web browsers to interact with HTTP Callbacks directly unless some additional libraries or plugins are used. Furthermore, when clients are behind (corporate) firewalls traffic coming from the smart things through callbacks will very often be blocked.

WebSockets for the Real-time WoT

As a consequence of this constraint, several techniques appeared in order for servers to push data back to clients without having clients explicitly requesting it. Since browsers were not designed with server-sent events in mind, Web application developers have tried to work around several specification loopholes often referred to a *Comet* techniques [284]. Comet is an umbrella term for most work-around, two of which are used quite often in practice: *Long Polling* and *Streaming*.

In the first technique, *Long Polling*, the client issues a request that will end only when the server is ready to send some data. Directly after the response the client will re-issue a request and so forth. In the second technique, *Streaming*, the client issues a request and the server never signals the end of this request, instead it keeps sending data over the TCP connection. In the absence of events, some servers will regularly send dummy data to prevent the connection from being closed.

While these work-around are used in practice they have two drawbacks: First, they gen-

erate unnecessary traffic [125]. More importantly, they are extremely resource demanding for the vast majority of Web servers. Indeed, most currently deployed Web servers allocate one thread or process for each connected client. Unlike in traditional HTTP requests, Comet requests do not end and thus quickly overload the memory of servers. To prevent this, a new generation of Web servers sometimes called non-blocking servers feature routines that let them suspend connections and manage several of them in a single thread. Researchers have been implementing such a server for wireless sensor nodes that can manage up to 256 Comet connections [41].

More recently, WebSockets (part of the HTML5 drafts [282]) were proposed. WebSockets propose duplex communication with a single TCP/IP connection directly accessible from any compliant browser through a simple JavaScript API. The increasing support for HTML5 in Web and Mobile Web browsers and makes it a very good candidate for pushing data in the WoT. Furthermore since WebSockets basically consist of an initial handshake followed by basic message framing, layered over TCP, they can be implemented in a straightforward manner on all platforms supporting TCP/IP, not only browsers.

tPusher We propose to add support for WebSockets to the Web of Things Architecture in order to offer a Web real-time eventing mechanism to communicate with smart things. Rather than implementing the protocol directly on smart things we propose adding it as a component of our Smart Gateway architecture as shown in Figure 2.5. We call this new component tPusher (things pusher) as introduced in [72].

System Architecture tPusher's integration and usage is summarized by the sequence diagram Figure 2.9. The sequence of events to enable a real-time communication between Web clients and smart things is as follow:

Smart Gateway Subscription to the Smart Things First, the Smart Gateway needs to establish a communication with the smart things. This is done through an HTTP Callback (Web Hook) subscription. Alternatively, for smart things not supporting TCP/IP and HTTP communication, this can be done through the synchronization-based driver approach that we will present in Section 3.1.1 where the Smart Gateway polls the smart things regularly using its Device Driver. From this point on, the Smart Gateway will regularly get data either by getting it pushed by the device or by pulling it.

Client Upgrade to WebSockets The Web client then issues a `POST` request on the Smart Gateway (e.g., on `/topic/temperature`) and asks for a protocol `Upgrade` to WebSockets, note that the protocol `Upgrade` is a standard HTTP mechanism. The `Upgrade` is accepted and WebSocket messages can be sent back and forth between the Smart Gateway and the Web Client.

WebSocket Push From this point on, the Smart Gateway relays (through the tPusher module) the data to the Web client over the same TCP/IP socket that is being kept open.

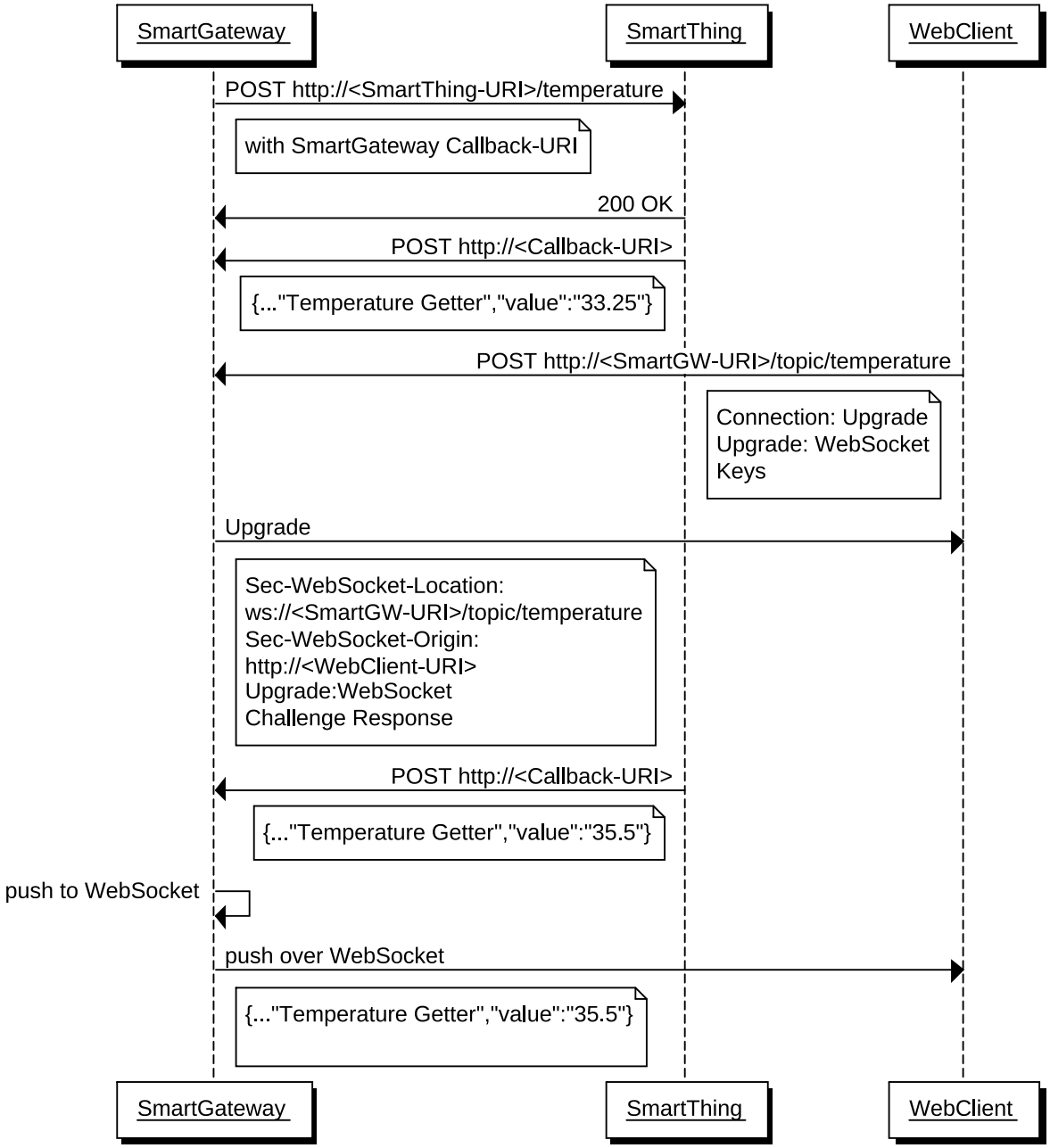


Figure 2.9: Sequence diagram of the real-time communication between Web clients and smart things through Smart Gateways and the tPusher component. tPusher is deployed on a Smart Gateway where it is used to serve content from smart things through a WebSocket interface.

The WebSocket specification also offers a JavaScript API that allows creating clients directly in browsers. The simplicity of this API (that should be supported by most browsers when HTML5 is finalized) is the power of WebSockets. Indeed, a shown in Listing 2.4 within 6 lines of simple JavaScript code, Web applications can open a WebSocket connection and thus, in our case, have a standard Web real-time communication with smart things.

```
1  var myWebSocket = new WebSocket("ws://www.webofthings.com");
2
3  myWebSocket.onopen = function(evt) {
4  alert("Connection open ..."); };
5  myWebSocket.onmessage = function(evt) {
6  alert( "Received Message: "  +  evt.data); };
7  myWebSocket.onclose = function(evt) {
8  alert("Connection closed."); };
9
10 myWebSocket.send("Hello Web Sockets!");
11 myWebSocket.close();
```

Listing 2.4: WebSockets JavaScript Client API. These lines of code are enough for a Web page to subscribe to a WebSocket and react on all possible incoming events.

Software Implementation Our implementation is based on Atmosphere [223], a Java abstraction framework for enabling push support on most Java Web servers. One of the advantages of this approach is to be able to deploy tPusher on recent Web Servers such as Grizzly [240], which are highly optimized to push events on the Web because of their usage of non-blocking threads for each new client. In order to support browsers or other clients that do not support HTML5 WebSockets yet, we use a client-side abstraction JavaScript library called Atmosphere JQuery Plugin which falls back to a Comet type of connection in case WebSockets are not supported by the client.

2.1.4 Summary and Applications

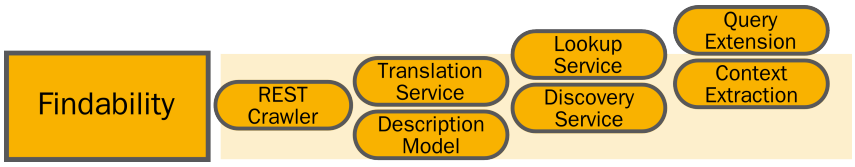
In this section we discussed the integration of smart things to the Internet and the Web. First, we applied the RESTful principles in a systematic manner. These guideline are then applied to Web-enable several smart things. In particular in Section 3.1 where they are applied to Wireless sensor nodes, or in Section 4.3.2 where we apply them to RFID systems.

Then, we discussed the concept of Smart Gateways to bring non TCP/IP and HTTP objects to the Web. The generic architecture of the Smart Gateway we described is used as a basis for the WSN Web-enabling described in Section 3.1.1 as well as a guideline to implement a Smart Gateway for smart meters described in Section 3.2.1.

Finally, we discussed several ways of adding support for smart things to push events to Web clients and proposed the tPusher service as an extension of Smart Gateways. The

tPusher service is evaluated in Section 4.5.2 where is it used to push data from RFID readers to mobile phones.

2.2 Findability Layer



By applying the architectural design presented in the Device Accessibility Layer, smart things become seamlessly part of the Web. While this presents several advantages, it also raises important challenges. Amongst these is searching and finding relevant services: *Given an ecosystem of billions of smart things, how do we find their services to integrate them into composite applications?*

The Web faced similar challenges when it moved from an hypertext of several thousands of documents to an application platform interconnecting an unprecedented number of documents, multimedia content and services. Rapidly, search engines such as Altavista, Yahoo and more recently Google appeared to offer search and indexes services.

The WoT will face similar problems, while finding smart things by browsing HTML pages with hyperlinks in a home environment is suitable and desirable, on a city, country or world-wide scale it becomes literally impossible. Hence, the ambient findability [144] of smart things need to be addressed, we need to make them searcheable and findable.

While we do not pretend providing the ultimate solution to this complex and heavily-researched problem [187, 166], we report on two aspects that we studied in the context of the Web of Things. First, we look at the integration of smart things to existing search engines and propose the use of a description model implemented with semantic annotations to enable this.

Then, we illustrate the shortcomings of basing the findability of smart things solely on existing search engines and propose a lookup and registration infrastructure adapted to the particular needs of the Web of Things and building upon the proposed description model. The combination of both solutions enables users and developers to run search queries such as looking for all the nearby temperature sensors or finding a device that can read video content in a particular building.

2.2.1 Search Engines and the Internet of Things

A Web page really becomes usable on the Web once it has been indexed by search engines. Thus, the most straightforward way of enabling the search for smart things is

through these search engines. However, searching for things is significantly more complicated than searching for documents.

First, smart things have no obvious easily indexable properties, such as human readable text in the case of documents. Then, they are tightly bound to contextual information, such as their absolute location (i.e., latitude and longitude), their abstract location (e.g., Room B, Floor 1) or current owner.

A Smart Things Description Model and Microformats for the WoT

Hence, smart things need a mechanism to describe themselves and their services to be (automatically) discovered and used. Since both humans and machines are going to use the things, we need a mechanism to describe a smart thing on the Web so that both, humans and machines, can understand what services it provides. This problem is not inherent to smart things, but more generally a complex problem of describing services, which has always been an important challenge to be tackled in the Web research community, usually in the area of the Semantic Web. For the WoT, the problem has also roots in the notion of *context* in Ubiquitous Computing [171, 168].

Here, we propose a model [77] (called Smart Things Metadata model) of the contextual information and metadata a smart thing should disclose on the Web to be searcheable and integrable into composite applications. We base our model on several surveys [94, 4, 38] of existing languages for semantically describing real-world objects and in particular, sensors [18] and industrial machines [103]. We describe a smart thing along 2 clusters of information each containing two sub-groups:

- First, static properties, as shown in Figure 2.10 are metadata that will not evolve over the life-cycle of the object:
    1. Product: contains a description of what the smart things is in terms of object.
    2. Services: contains a description of the services a smart thing offers (e.g., temperature monitoring, MP3 play-back, etc.)
- Then, dynamic properties, as shown in Figure 2.11 are those changing regularly depending on the context the object is located in:
    1. Location: contains information about the place where the thing is currently located.
    2. QoS (Quality of Service): contains information about how well the thing performs and performed.

This model is not exhaustive but, according to our experience [77], it covers the basic information required to describe smart things on the Web in order to make them and their services searcheable. Furthermore, the idea is to use this information as search engines do, i.e., in a best effort manner where the absence of some metadata does not

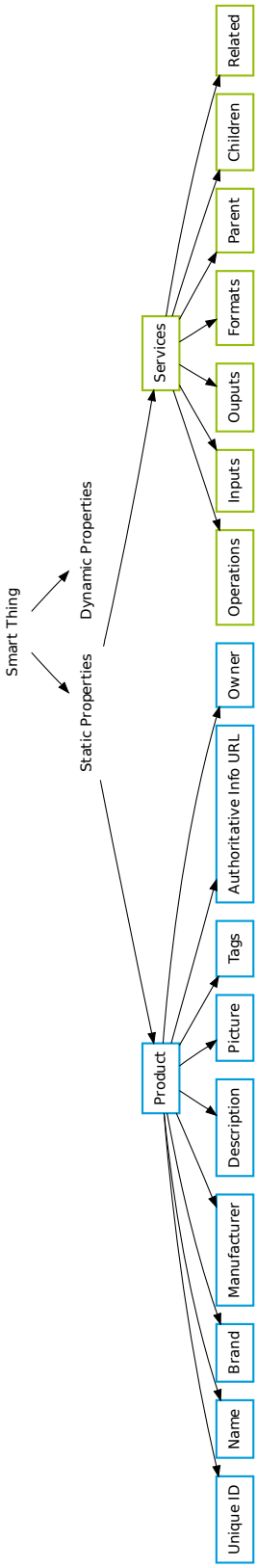


Figure 2.10: The smart thing metadata model, containing the most important elements of the description of a smart thing required for their findability on the Web. This graph contains the static properties of smart things, Figure 2.11 contains the dynamic properties.

mean the smart thing is not indexed. Rather it means that customization of its rendering or indexed keywords will simply be limited.

The description proposed here can potentially be materialized into several formats such as WSDL (Web Service Definition Language) files, DPWS Metadata [103] or SensorML [18] documents. Unfortunately they are not exposed on the Web as Web-browsers and search engines, for the most part, do not understand them.

To overcome the rather limited descriptive power of resources on the Web, several languages have been proposed as standards. Two of them, RDFa [265] and microformats [254] have the interesting feature of being used to semantically enhance the elements of HTML pages.

Designed for both, human and machines, microformats provide a simple way to add semantics to Web resources [9]. There is not one single microformat, but rather a number of them, each one 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.

Microformats are especially interesting in the Web of Things for three reasons; first, like RDFa they are directly embedded into Web pages and thus can be used to semantically annotate the HTML representation of a thing's RESTful API. Secondly, each microformat undergoes an open community-driven standardization process. This ensures that the number of formats stays relatively small and that their content is to be widely understood and used when accepted. Finally, many microformats are already supported by search engines, such as Google and Yahoo, where they are used to enhance search results and render them differently. For example, the `geo` microformat is used to localize search results close to a user or `hReview` is used to rank search results according to the users' opinion.

As a consequence, we propose the use of microformats to describe smart things in the Web of Things. Rather than proposing a new microformat encompassing the model described in Figure 2.10, we can re-use a compound of existing, standardized microformats. This helps the things to be directly searcheable by humans using existing general purpose or dedicated search engines, but it also helps them being discovered and understood by software applications in order to automatically use them and render adapted user interfaces.

To illustrate this, we show that by using a compound of 5 microformats and by leveraging the structure of RESTful APIs, we can create a description of a sensor node that fulfills the model presented in Figure 2.10 and Figure 2.11.

Product Description One of the most important metadata required in order to enable the search for smart things and their services is a description of what object they are. Web sites such as e-commerce services, are often based on unstructured product data which makes it hard for browsers and search engine to render and index useful metadata

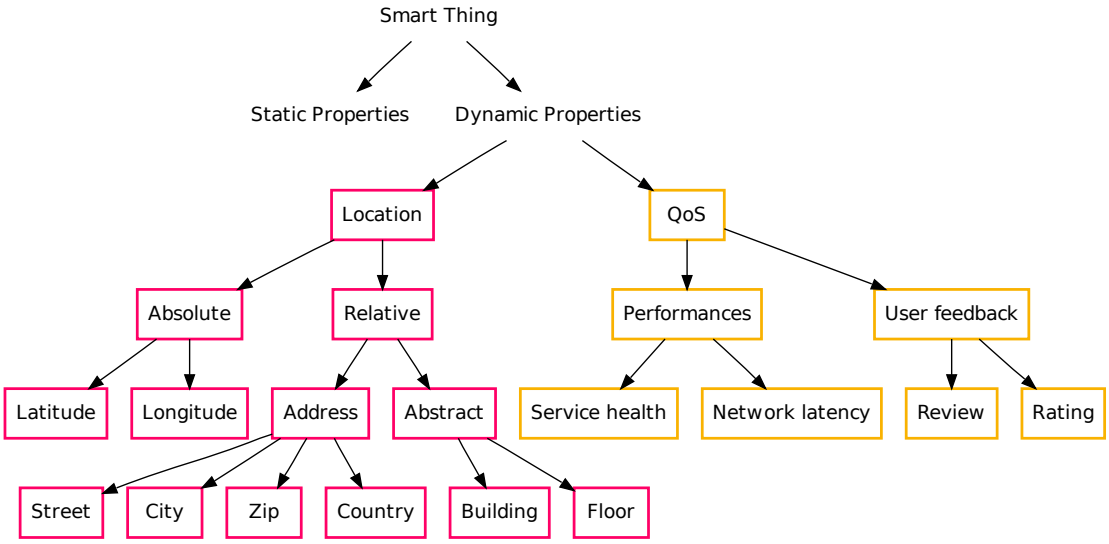


Figure 2.11: Second part of the smart things metadata model, containing the most important dynamic properties of the description of a smart thing required for the findability of their services on the Web. Figure 2.10 contains the static properties of the STM model.

about products. The *hProduct* [297] microformat was created to give a structure to this metadata. Through its vast usage, browsers, search engines and other Web applications have a way to help facilitate the best product choice for consumers. It also gives a way for manufacturers and retailers to better describe their products. Although it is officially still a draft microformat [297] at the time of writing, hProduct is already widely used and implemented on the Web. As an example, the BestBuy e-commerce site uses it for providing metadata about all its products and Google [290] supports it to better render the results of product searches.

Interestingly enough, the hProduct microformat provides information about the object itself and its manufacturer and covers most of the fields required in the Product description of the STM model. As shown in Table 2.1, except for the Owner and Manufacturer, hProduct covers all the STM model Product related fields. The Owner and Manufacturer is implemented using the hCard microformat that we will present below. Listing 2.5 presents an example of how a generic sensor node could be represented using hProduct. Note that the microformats' attributes are directly embedded into the HTML representation of the smart thing. As a results, browsers, search engines and applications discovering the generic node by browsing will be able to render its UI and visualization in a metadata enhanced manner.

In Listing 2.5, the smart thing unique identifier is implemented using an EPC number. Using Electronic Product Code numbers [169] has the advantage of offering a world-wide, static way of identifying objects which is very valuable in the Web of Things where objects might move from one domain to the other, thus changing their absolute URI over time. We will discuss the properties of EPC numbers in greater details in Chapter 4.

STM element	Microformat	MF Attribute	Meaning
Unique ID	hProduct	identifier (type, value)	unique identifier for this object
Name	hProduct	fn	human-friendly name of the smart thing
Brand	hProduct	brand	company name
Description	hProduct	description	human-friendly description
Picture	hProduct	picture	image of the product
Authoritative URL	hProduct	url	manufacturer's Web page containing information about the product
Tags	hProduct and rel-tag	category (rel-tag)	tags describing the smart thing
Owner	hCard	see Table 2.2	name and address of the owner
Manufacturer	hCard	see Table 2.2	name and address of the manufacturer

Table 2.1: Elements of the STM model for the Product cluster that can be implemented in a Web-oriented way using standard microformats.

```
<html>
    <head>
        <title></title>
        <meta http-equiv="Content-Type" content="text/html;
           charset=UTF-8">
    </head>
    <body>
        <span class="hproduct">
            <span class="fn">Generic Sensor Node</span>
                <span class="identifier">,
                    <span class="type">epc</span> unique
                       identifier
                    <span class="value">urn:epc:id:gid
                       :2808.64085.88828</span>
                </span>
                <span class="category"><a href="http://www.
                   webofthings.com/tags/wsn" rel="tag">
                   Wireless Sensor Nodes</a></span>
                <span class="brand">Generic Electronics
                   Company</span>
                <span class="description">This is a Web-
                   enabled sensor node that can
                    help you monitor your energy consumption</
                       span>
                <img alt="photo of the generic sensor node"
```

```
18                          src="http://www.webofthings.com/wp-
                               content/themes/paperpunch/images/logo.
                               png" class="photo"/>
19                      <a href="http://www.webofthings.com/?s=wsn"
                           class="URL">More information about this
                           device.</a>
20                      <span class="price">20$</span>
21                  </span>
22      </body>
23  </html>
```

Listing 2.5: Describing a smart thing using hProduct.

Location One of the most important differences between virtual and physical objects is that the latter have a location in a physical context. This information is very valuable and should be leveraged when providing metadata for smart things.

Initially created to represent people, companies and organizations, the hCard microformat [294] is a also simple and Web interoperable way of representing places. It is based on the vCard specification [296] and has reached the status of standard microformat. As a consequence, it is widely implemented on sites across the Web and used for adapted rendering and context extraction by several Web resources and applications. As an example, it is used by Google both to render special results for businesses, showing their location on a Map, as well as to enable their customers to "export places" from Google Maps [300]. Similarly, the Yahoo Local Search engine uses hCards to render the location of businesses and organizations [289].

In addition to hCard, the geo microformat [292] makes it possible to embed absolute location information in Web pages in the form of geographic coordinates.

In the context of smart things, we use hCards and geo to implement three parts of the STM model. First, for the static properties, hCards are used to describe the owner and manufacturer of an object. Then, for dynamic properties, we use hCard and geo to describe the location of a smart thing. The mapping of the STM model location properties to hCards attributes is quite natural and shown in Table 2.2.

STM element	**Microformat**	**MF Attribute**	**Meaning**
Latitude	geo	latitude	current latitude of the smart thing, owner or manufacturer
Longitude	geo	longitude	current longitude
Address	hCard	adr (street-address, locality, postal-code, country-name)	comprehensive postal address

Table 2.2: Elements of the STM model for the Location cluster that can be implemented in a Web-oriented way using the hCard and Geo standard microformats.

It is worth noting that these microformats do not cover relative abstract locations. The reason behind this is that this part of a location cannot be leveraged globally in a standard way (e.g., by a search engine) as it requires a specific knowledge of the current environment. In Section 2.2.2 we propose a way to implement this part of the STM model using a lookup infrastructure.

Quality of Service In a Web of Things populated by billions of smart things quality of service information can be of great help to choose the right smart thing for the right application. Parameters such as bandwidth, up-time, average response time help taking the right decision. These data can be based on monitoring service [77] or provided by the smart thing manufacturer.

However, with the advent of the Web 2.0, we increasingly rely on external user experiences when choosing our products and services. The strong influence of recommendation systems on the way people pick Web sites or buy products online has been extensively studied [175] and demonstrated [200]. For smart things we can take a similar approach and offer a standard way for providing user-generated reviews as well as performance information.

hReview is a microformat for embedding reviews of products, services, businesses and events in Web representations and especially in HTML [293]. Several Web sites already implement hReview for their reviews. As an example, the New York Times Web site [275] and Yahoo Local search use it to rate listed venues such as restaurants and businesses.

In the Web of Things context, we can use hReview to implement both QoS properties listed in the STM model. Indeed, as shown in Table 2.3 the standard attributes of hReview cover the metadata related to performances and user feedback. Listing 2.6 shows how

STM element	**Microformat**	**MF Attribute**	**Meaning**
Review	hReview	description & type (product)	feedback from the owner/user of the smart thing
Rating	hReview	rating (value, worst, best)	owner/user rating between worst and best or 1.0 to 5.0 if scale is omitted
Address	hCard	adr (street-address, locality, postal-code, country-name)	comprehensive postal address
Service health	hReview & rel-tag	tag (rel-tag)	specifies that a review is a service health parameter
Network latency	hReview & rel-tag	tag (rel-tag)	specifies that a review is a service health parameter

Table 2.3: Elements of the STM model for the QoS cluster that can be implemented in a Web-oriented way using the hReview standard microformat.

the QoS properties of the STM model can be implemented using hReview. The listing

contains two QoS elements: First, the owner of the smart thing published a user feedback. Then, the smart thing generated a service health review.

```
<div class="hReview">
  <span><span class="rating">4</span> out of 5 stars</span>
  <h4 class="summary">Good all purpose sensor node</h4>
  <span class="reviewer vcard">Added by owner: <span class="fn
     ">Dominique Guinard</span></span>
  <div class="description item">I use this generic sensor node
      for monitoring the temperature inside my house. It is
     quite reliable but I turn it off on weekends.
  </div>
</div>
<div class="hReview">
  <span>
    <a href="http://www.webofthings.com/tags/serviceHealth"
       rel="tag">Service Health:
      <span class="value">70</span>/
      <span class="best">100</span>
    </a>
    </span>
    <h4 class="summary">Service health is good</h4>
    <span class="reviewer vcard">Added by manufacturer:
      <span class="fn">Generic Electronics Company</span>
    </span>
    <div class="description item">The service health of this
       node is good, this means that most requests will
       succeed within less than 1 second.
    </div>
</div>
```

Listing 2.6: Quality of service for a smart thing described using the hReview microformat.

Service Description: Discovery by Crawling The last part of the STM model does not necessarily need to be supported by an explicit semantic description. Indeed, if we consider that the Device Accessibility Layer was implemented as described, we can assume that all the smart things will serve their functionality through a RESTful interface.

A direct consequence of respecting the constraints of REST is that useful meta information can be extracted simply by crawling their HTML representation [138, 7] and leveraging the HTTP protocol.

From the root HTML page of the smart thing, a crawler typically is able to find a number of the service properties suggested in the STM model. First, to satisfy constraint C4 (Hypermedia Driving Application State, see Section 2.1.1), the HTML representation of a smart thing should contains links to related and descendant resources. Hence, from this

constraint a crawler can extract the Children, Parents and related URIs as specified in the STM model.

With these URIs, the crawler can then use the HTTP `OPTION` method to retrieve all verbs supported for a particular resource, e.g., `PUT`, `POST`, `GET`, implementing the Operations property of the STM model. Finally, with content-negotiation as described in Section 2.1.1, the crawler gets information about the Service Format, Input and Output properties.

We implemented and empirically tested the described crawling algorithm in [71, 138]. The pseudo code of this algorithm is shown in Listing 2.7.

```
1  crawl(Link currentLink) {
2    new Resource() r;
3    r.setUri = currentLink.getURI();
4    r.setShortDescription = currentLink.text();
5    r.setLongDescription = currentLink.invokeVerb(GET).
        extractDescriptionFromResults();
6    r.setOperations = currentLink.invokeVerb(OPTIONS).getVerbs()
        ;
7    foreach (Format formats: currentFormat) {
8      r.setAcceptedFormats = currentLink.invokeVerb(GET).
          setAcceptHeader(currentFormat);
9    }
10   if (currentLink.hasNext()) crawl(currentLink.getNext());
11 }
12 foreach (Link currentPage.extractLinks(): currentLink);
```

Listing 2.7: The smart things Service Description Crawling Algorithm.

The crawling approach to extract service metadata is interesting because it does not require the semantics of services to be represented in an additional format. As a consequence valuable information can be extracted even if the smart thing to index implements the Device Accessibility Layer only. In fact the information that can be extracted by crawling is rich enough to index a smart thing, a few keywords and to locate all its resources. Hence, the smart thing already becomes searcheable.

However, the crawling approach has two main limitations. First, the approach strongly relies on the respect of the REST constraints. As a consequence, for some smart things' APIs such as those based on hybrid architectures [160] not fully respecting the constraints of REST, the service metadata might be only partially extracted [7].

Furthermore, the crawling approach requires many HTTP calls to extract a metadata profile that matches the service properties of the SDT model. This is problematic since HTTP calls are the most costly parts of the communication between clients and services [182]. As a consequence, a single HTTP call returning a significant amount of data is more efficient than several calls returning the same total amount of data. This is especially important in the context of the Web of Things where clients will interact with services deployed on resource constrained devices such as Smart Gateways or smart things.

Several solutions exist to provide service metadata for RESTful APIs. The most well known one is called WADL (Web Application Description Language [160]). This language, directly inspired from the WSDL (Web Service Description Language), provides a way of describing HTTP based Web applications. A WADL document is an external document describing, from a client point of view, how to interact with a given HTTP based service. The main drawback of the approach in our context is that it requires clients to understand a new format. Furthermore, when compared to microformats, the rather low adoption rate of the WADL format [160] does not enable applications to leverage existing search engines.

hRESTs is a microformat [111] sharing similar goals with WADL, with the advantage of being directly embedded in the HTML representations of services. Because most of the metadata it offers is implicitly available in a well designed RESTful API, hRESTS is sometimes criticized by the Web community. More importantly, hREST is the work of three researchers and, at the date of writing it is not yet an official community-driven microformat which severely hinders its support by services such as search engines.

However, in practice hRESTs offers the advantage of providing the service metadata without crawling the resources. As a consequence, the metadata is more strictly organized and easier to extract which reduces the number of required HTTP calls. To benefit from this, hRESTs should be used to annotate a global description of the smart thing's services, for example accessible at the root HTML page of the smart thing.

Understanding the Benefits of Microformats Following our guidelines, a smart thing is best described by a compound of five microformats covering the STM model: hProduct, hCard, hReview, rel-tag and possibly hRESTs.

The benefits of this approach are manifolds. First, smart things become directly searcheable with traditional search engines such as Google or Yahoo. Moreover, these search engines can use the metadata to provide contextual search results. As an example, searching for a temperature sensor nearby from a mobile phone can use the geo microformat of the smart things to match the GPS coordinates of the mobile phone. Search engines will also be able to render the search results differently based on the metadata of smart things.

Similarly, based on this metadata, clients such as Web browsers or mobile phone applications are able to render the user interfaces to smart things in a customized way, we illustrate this with examples of dynamically rendered UIs and mashup modules for Wireless Sensor Nodes in Chapter 3.

Towards an STM Translation Service While we suggest implementing the STM model using the proposed compound microformat for the best current integration experience, it is clear that this is not a one-size-fits-all format. From the history of metadata formats such as DPWS, WSDL, WADL, SensorML [18] or SA-REST [177] it is quite clear that no metadata language can impose itself up to a point where others vanish, because there

is no single best way of describing a smart thing. Hence, smart things will most likely be and already are described using other metadata formats.

However, the format should not matter, what should is the metadata. Hence, the idea is to introduce a level of indirection in order to support a broad spectrum of metadata formats. As introduced in a common work with Simon Mayer [138] as well as in [77, 4], we implement a STM Translation Service that acts as a converter. On the one hand-side it can extract (or crawl) information from several metadata formats and on the other end, through a RESTful Web API, it offers to clients to retrieved the extracted metadata using the representation they wish (if supported).

2.2.2 A Web-Oriented Discovery and Lookup Infrastructure

Relying on search engines to enable searching for smart things is interesting because it uses existing, well-known and widely adopted services. However, the approach has a number of limitations. First, because of their mobile nature, smart things tend to be moved from one context to another on a regular basis: sensors attached to shipments move from the factory to a warehouse. Mobile phones entirely change their context several times per day. Environmental monitoring systems are moved from one observation area to the other. While improving support for real-time search (e.g., through integration with real-time information services such as Twitter), search engines still largely function based on scheduled indexing and might not reflect the latest context of registered smart things. Thus, the need for *local search engines and lookup services for smart things.*

Moreover, the bootstrapping of smart things is a problem: *How does a smart thing announce its existence in a particular context?* Currently, search engines discover new resources by following links. For the Web of Things, we need to be able to access smart things as soon as they connect thus the need for a *discovery service for the Web of Things.*

We present a distributed and Web-oriented registration and lookup infrastructure for the WoT federating our joint work in [137, 192] as well as our work on defining a search and discovery process for real-world services running on smart things [77][2]. We begin by demonstrating how the infrastructure offers a discovery protocol for smart things. We then show how this infrastructure can be used to perform local search queries.

Distributed Infrastructure

Our Discovery and Lookup infrastructure is composed of several Local Lookup and Discovery Units (LLDUs) [77, 137]. These software components allow smart things to announce themselves and clients to search for specific (local) services offered by connected smart things. The internal structure of an LLDU is shown in Figure 2.16 and will be described in details in the next sections.

[2]Parts of the described process were patented in [70].

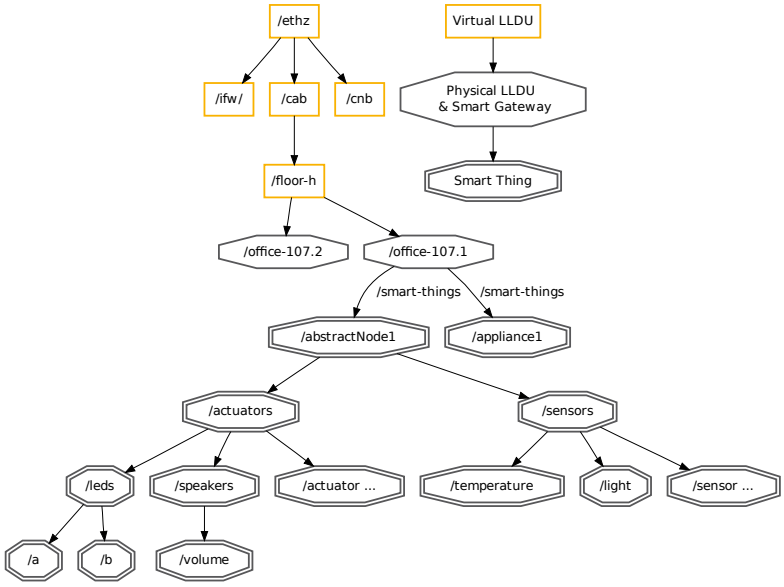


Figure 2.12: Hierarchical organization of the Local Lookup and and Discover Units (LLDU). Virtual LLDUs are can be located anywhere whereas Physical LLDUs are coupled with Smart Gateways.

Based on common work with Trifa et al. [190] we suggest than rather than having a flat structure for LLDUs, we deploy them in a hierarchical way, reflecting the abstract locations of the current context in the resources' URIs. Using abstract location information in URIs has a great value since it facilitates browsing for smart things in a particular context. For instance it lets you navigate through all the smart things in a building, floor or room simply by pointing to the correct URI of following the provided hyperlinks structure.

A typical hierarchy is shown in Figure 2.12. The first three levels are virtual LLDUs. Virtual LLDUs can be deployed on any machine anywhere in the world and one physical machine can host several virtual LLDUs. Smart things do not directly communicate with them as their purpose is only to encapsulate the hierarchy of abstract locations and to serve search queries for these locations. As an example, in Figure 2.12 the `ethz, ifw, cab` and `floor-h` are virtual LLDUs all hosted on the same physical machine.

Like virtual LLDUs, physical LLDUs serve search queries for the abstract locations they represent, however physical LLDUs also serve as Discovery Services for smart things. A physical LLDU is a software component that can be loaded in a Smart Gateway. As shown in Figure 2.12 the `office-107.1` LLDU node covers the abstract location encapsulated in the following URI: `/ethz/cab/floor-h/office-107.1/`. Directly below this node are attached the resource trees (see Section 2.1.1) formed by smart things managed by the smart gateway in `office-107.1`.

Concretely, the LLDUs can be deployed and configured using a `PUT` request to the root URI of a running LLDU with a payload specifying its configuration and context. As

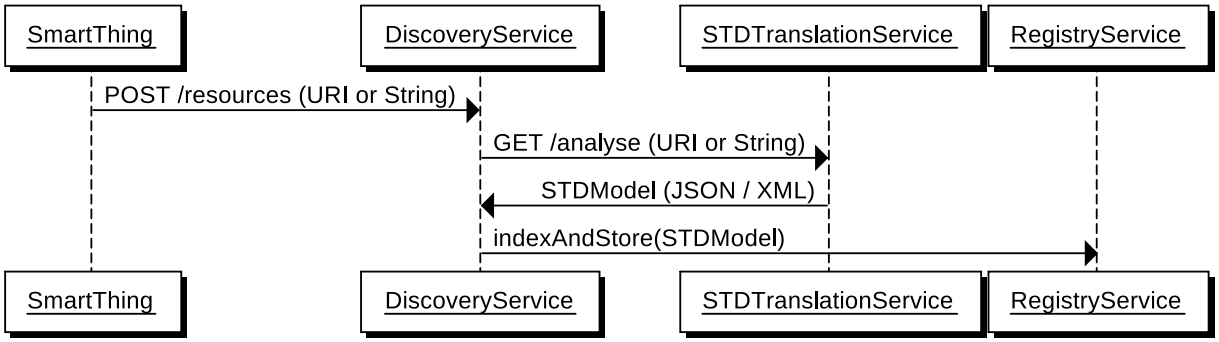


Figure 2.13: Sequence diagram of the discovery process. The LLDU Discovery Service uses a STM Translation Service to extract metadata for the smart thing. The returned information is sent to the LLDU Registry Service which indexes and stores the metadata of the discovered smart thing.

an example Listing 2.8 is a JSON document that configures a new LLDU located at `/eth`. The rest of the specified contextual information (e.g., latitude, longitude) will be inherited by smart things connected to this LLDU that cannot deliver a full STM model, for instance because they do not have a GPS module.

```
{
  "resourceUrl": "http://webofthings.com:2401",
  "uuids": [{"uuidType": "infraWoT", "uuidValue": "eth"}],
  "name": "LLDU for ETH",
  "context": {
    "hierachical": {"hierarchyString": "eth/", "
       hierarchyDelimiter": "/"},
    "postal": "Universitaetsstrasse 6, CH-8092 Zurich,
       Switzerland",
    "geographical": {"longitude": 8.550003, "latitude":
       47.367347}
  }
}
```

Listing 2.8: JSON document that configures a LLDU called "LLDU for ETH" and located at ETH.

Discovery Services

To solve the bootstrapping problem of smart things, physical LLDUs offer a Discovery Service. The discovery process is started by a smart thing wanting to be part of the Web of Things infrastructure.

As shown in Figure 2.13, once connected to the local network, the smart thing or gateway issues a `POST` request to the `/resources` end-point of the physical LLDU. As a payload of the request, the smart thing can either send its root URI or a payload describing its resources. The Discovery Service sends this to the STM Translation Service which will extract semantic metadata based on a best-effort principle, supporting several types of

metadata formats. The STM Translation Service returns a JSON representation of the extracted data.

The Discovery Service then binds the smart thing to the physical LLDU's absolute URI. As an example, after the discovery process, the `abstractNode1` in Figure 2.13 gets bound to `/ethz/cab/floor-h/office-107.1/smart-things/abstractNode1`. Then, the resources tree of the `abstractNode1` itself is also accessible through the LLDU as shown in Figure 2.12 below the `abstractNode1` resource.

Furthermore, to enable keywords-based search, the Discovery Service passes the STM model to a Registry Service. The role of this latter is to store the representation of the model as well as to extract two inverted indexes from it. Inverted indexes are a central component of search engines algorithms and are well suited for keywords-based textual searches [218]. For each resource the Registry Service creates three different entries, as show on Listing 2.9. First, it store the JSON string corresponding to the metadata of the resource in a file. It then adds entries into two inverted indexes. In the first index, it adds all the keywords that could be extracted from the resource's metadata. In the second index, it adds keywords extracted from the metadata related to the output of the resource.

```
1 // Store resource in table
2 writeStringToFile(resource.toJSONObject().toString(),
     databaseCoreTable);
3
4 // Get keywords for the resource and add inverted index
     entries
5 for (String keyword : resource.getKeywords())
     writeReverseEntryToFile(keyword.toLowerCase(), resource.
     toJSONObject().toString(), keywordReverseTable);
6
7 // Get REST Output from entity and add inverted index entries
8 for (String restOutput : entity.getRESTOutput())
     writeReverseEntryToFile(restOutput.toLowerCase(), entity.
     toJSONObject().toString(), restOutputReverseTable);
```

Listing 2.9: Indexing the resources in two inverted indexes based on extracted keywords of the STM model.

Lookup Services

One of the benefits of deploying an infrastructure of LLDUs is the ability to perform localized search queries. The lookup service offers a query interface for clients such as developers looking for real-world services to integrate into their composite applications, end-users wanting to discover the registered services for a particular place or applications dynamically looking for simple services.

Types of Parameters The Query Service of LLDUs is built on top of the Registry Service. Clients can access it by sending a `POST` request to the `<LLDU-URI>/query` resource on an LLDU. The actual query should be specified either through `application/x-www-form-urlencoded` parameters or as a JSON payload. Since queries will be distributed amongst the infrastructure of the LLDUs (traveling down or up the resources tree), LLDUs also pass queries to each-other using the same mechanism.

Queries can be formulated according to the following parameters basically corresponding to most relevant fields of the STM model that were extracted by the STM Translation Service during the discovery process:

Keywords A number of free-text, unstructured keywords can be provided. The matching algorithm is a traditional keywords search process iterating through the following properties that were extracted from the device's representation of the STM model: name, category, brand, description and user provided tags. These keywords can then be extended by the system using external services as explained in Section 2.2.2.

Name As several smart things support user provided names, searching for these might be really valuable to users and thus is offered by the API.

Unique ID Queries by universal unique identifiers e.g., Bluetooth IDs, Zigbee MAC addresses, IPs, Electronic Product Codes (EPC, see Section 4.1), are a straightforward way for applications to search for a particular smart thing.

Ratings Clients can use user generated or smart things provided quality of service ratings as specified in the STM model. For instance this type of query parameters can be used to find the most reliable wireless sensor node of a certain type as in practice it is often the case that one node is more reliable than the other.

REST Service The matching algorithm activated by this parameter leverages the metadata enhanced description of RESTful APIs based on the hRESTs microformat.

When performing a search, the results sets for each type of parameter are fetched and the intersection of all the sets is returned to the client through the RESTful Web query interface.

Types of Queries Parameters define the keywords of a query but thanks to the tree-structure formed by all LLDUs the locality of searches or scope can also be leveraged. This concept is encapsulated in the `queryType` parameter that has to be provided by clients when using the querying API.

We consider three types of queries that can be performed on the LLDU infrastructure and illustrate their particular interest when looking for services provided by smart things:

Exhaustive Queries These queries start at the LLDU where the request originated and are pushed to all children LLDU nodes, eventually returning all the resources that matched within the subtree. Thus, such a query will go down to the leafs and up again through every node until the originator is reached again. As an example such

a query can be used to retrieve all the temperature sensors in a city in order to compute an average temperature.

Cardinality Queries These queries are used to find exactly n resources corresponding to the query parameters. The query process is launched on the children LLDU nodes and will be stopped as soon as n services are found in the result set. However, since the process is distributed amongst several subtrees in its current implementation the process may retrieve more than n services, hence the result set is eventually filtered to keep only n results, giving more weight to LLDUs located higher in the subtree. Such a query could be used for instance to find pairs of smart meters that monitor a certain type of device (e.g., a fridge) to compare their actual energy consumption.

Best Effort Queries Such a query is in fact a Cardinality Query with a stopping condition of $n = 1$. It is used to find the first resource that fits the user needs. As an example it can be used to find a usable printer in a facility.

Located Queries Since the other types of queries will start their tree-traversal only from the location of the originator LLDU, there is a need to support an arbitrary starting point. Located Queries implement this feature. When a hierarchical location is specified in a query, a Located Query is triggered and sent to the corresponding LLDU in the hierarchy where the query is started. Such a query can be used for instance in the case a user is located in a particular room (bound to the physical LLDU in this room) but wants to query for the energy consumption of the whole department he is located in.

Query Augmentation Service To provide better results without requiring additional semantics on the smart things side, the Query Service can be extended with a query with a Query Augmentation Service we proposed in [77].

In conventional service discovery applications, the keywords entered by the user are sent to a service repository to find types of services corresponding to the keywords. The problem with this simple keyword matching mechanism is that it lacks flexibility required in the special case of real-world objects. As an example lets assume a developer or a user who wants to find services offered by a *smart meter*, a term often used to describe a device that can measure the energy consumption of other devices and possibly control them depending on built-in logic. Typing "smart meter" only, will likely not lead to finding all the corresponding services, because services dealing with energy consumption monitoring might not be tagged with the *smart meter* keywords but simply with *electronic power-meter*. However, since we want to avoid the construction of domain specific ontologies, and to minimize the amount of data that smart things need to provide upon network discovery and service registration, we propose a system that uses services on the Web to extend queries without involving communication with the smart things or requiring domain specific service descriptions from them.

The basic idea is to use existing knowledge repositories such as Web encyclopedias (e.g., Wikipedia), search engines (e.g. Google, Yahoo! Web Search) or domain-specific portals

(e.g., the Metering portal [253]), in order to extract *lightweight ontologies* [93] or vocabularies of terms from the Web resources' semi-structured results. The basic concept of the Query Augmentation is to call 1..n Web search engines or encyclopedias with the search terms provided by the user, for instance "smart meter". The HTML result page from each Web resource is then automatically downloaded and analyzed. The result is a list of keywords, which frequently appeared on pages related to "smart meter". A number of the resulting keywords are thus related to the initial keyword i.e., "smart meter" and therefore can be used when searching for types of services corresponding to the initial input.

Software Architecture An invocable Web-resource together with several filters and analysis applied to the results is called a *Query Strategy*. The structure is based on the Strategy Pattern [62], which enables us to encapsulate algorithms into entirely independent and interchangeable classes. This eases the implementation of new strategies based on Web resources containing relevant terms for a particular domain. A simplified class diagram of the Query Strategy framework is depicted on Figure 2.14. Any Query Strategy has to implement the `AbstractStrategy` class which provides the general definition of the algorithm. As an example the `YahooStrategy` is a concrete implementation of this algorithm using the Yahoo! Search service. Furthermore, strategies can have extensions, adding more specific functionality to a concrete instance of a Query Strategy. As an example the `WikipediaStrategy` can be extended with the `WikipediaBacklinks` class. This particular extension is using the backlinks operation offered by Wikipedia in order to know what pages are linking to the currently analyzed page similarly to what the well-known PageRank used to rank websites [21]. This information is then used by the `WikipediaStrategy` to browse to related pages and gather relevant keywords. As such, our approach builds on top of existing ranking and connectivity approaches on the Web.

Furthermore, Query Strategies can be combined in order to get a final result that reflects the successive results of calling a number of Web-resources. The resulting list of related keywords is then returned to the user, where he can (optionally) remove keywords that are not relevant. The implementation of the Query Strategy architecture makes it easy to test combinations of several strategies together with their extensions. We implemented a number of these, and their evaluation is presented in Section 2.2.3.

Context Extractor One of the main differences between services provided by smart things and virtual services is that smart things services are directly linked to the physical world. As a consequence, the context in which a service exists as well as the context in which the user or user initiates the discovery of a service are highly relevant. Context is information that qualifies the physical world, and it can help in both reducing the number of services returned to the user, as well as in finding the most appropriate services for the current environment [11].

To satisfy the requirements of real-world service discovery, we propose modeling the context into two distinct parts inspired from [171]: the *Digital Environment*, which we define

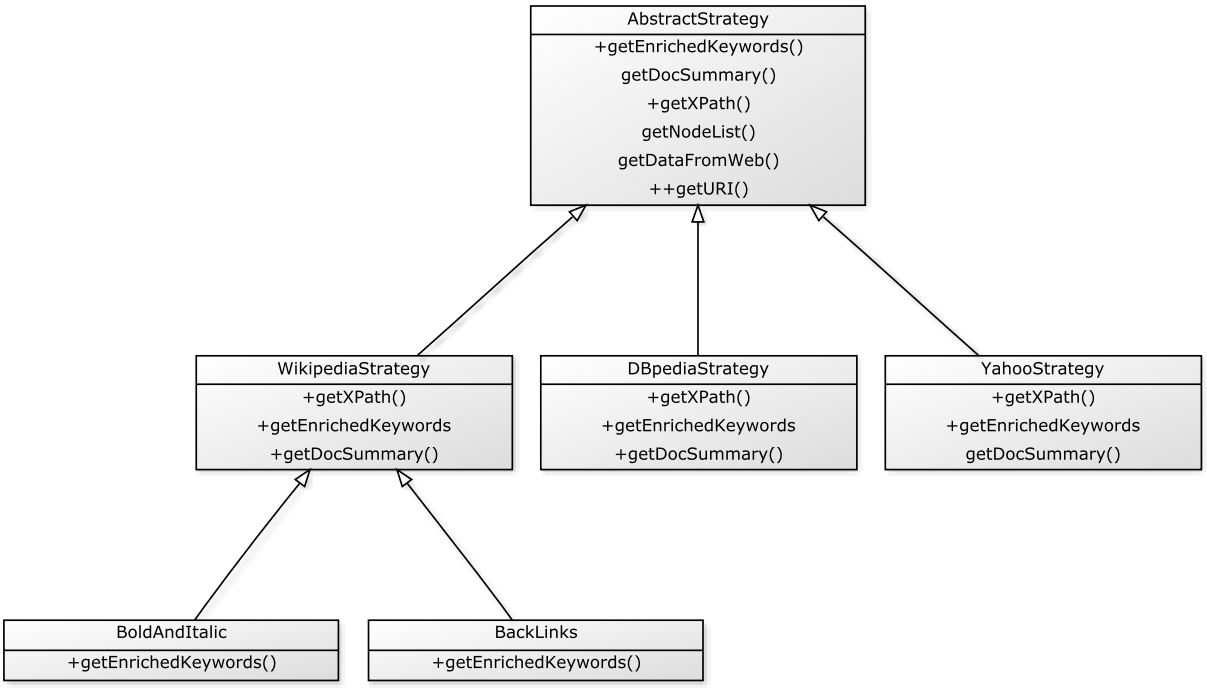


Figure 2.14: Architectural overview of the Query Strategies based on the Strategy and Template software design patterns.

as everything that is related to the virtual world the user is using, and the *Physical Environment*, which refers to properties of the physical situation the user currently is located in or wants to discover services about.

The *Digital Environment* is composed of Application Context and Quality of Service. The *Application Context* describes the business application the user uses when trying to discover services, e.g., the type of application he is currently developing or the language currently set as default. Such information co-determines the services a user or developer is looking for and can reduce the discovery scope. The *QoS Information* reflects the expectations of the user (or of the application he is currently using) in terms of how the discovered service is expected to perform. Our current implementation supports service health and network latency, i.e., the current status of the service and the network transmission delay usually measured when calling it.

The Physical Environment is mainly composed of information about location. Developers are likely to be looking for real-world services located at a particular place, unlike when searching most virtual services. We decompose the location into two sub-parts following the Location API for Mobile Devices (as defined in Java Specification Request JSR-179). The Address encapsulates the virtual description of the current location, with information such as building name, floor, street, country, etc. and the Coordinates are GPS coordinates. In our implementation the location can either be automatically extracted e.g., if the user looks for a real-world service close to his location, or it can be explicitly specified if he wants a service located close to a particular location e.g., in a form of radius.

Extraction of the context on the user side is done when starting the query in the smart

things lookup service Web user interface, the user can also influence these parameters by setting up preferences. It is worth noting that the context on the user side is meant to reflect the expectations or requirements with regard to the services that are going to be returned. As an example, during this phase the user can express the wish for a service to be physically close to his current location, or he can quantify the importance of context parameters such as Quality of Service.

This user-quality information is then going to be compared with the context stored on the LLDUs' indexes extracted by the STM Translation Service when discovering the smart things. This is done by the Service Ranking component in order to select and rank the most relevant resources.

Context of Smart Things The Digital Environment context parameters such as the device description or Quality of Service, are extracted upon discovery by the Discovery Service based on the microformat implementation of the STM model.

Getting the context parameters related to the Physical Environment of a service instance is slightly more complicated. Indeed, as an example it can not be expected from each smart things to know its location. Thus, we suggest taking a best effort strategy, where each actor of the discovery process is trying to further fill-in the context object. As an example, consider a mobile sensor node without a coordinates-resolving module (e.g., a GPS).

Upon discovery by a LLDU, the sensor node does not know its location and thus can not fill-in the Address and Coordinates fields of the STM model. The LLDU however, is a usually immobile component and is configured at setup time with its location and current address as explained in Section 2.2.2. As a consequence the LLDU can provided the Address and Coordinate information of the sensor node based on its own location (within a specific radius). While not entirely accurate with respect to the sensor's exact location, this information will already provide a useful approximation. Similarly, since we can not expect every LLDU to provide a full contextual profile, the LLDUs can also share their contextual information to complement one another.

Ranking Service Lookup Results The Service Ranking component is responsible for sorting the resources according to their compliance with the context specified by the user or extracted from his machine. This component receives a number of service lookup results alongside with their context profiles. It then uses a Ranking Strategy to sort the list of results. For instance, a Ranking Strategy can use the network latency so that the services are listed sorted according to their network latency; another could rank instances according to their compliance with the current location of a user or the target location he provided.

As for Query Strategies, Ranking strategies can be well modeled using the Strategy pattern. In this way, new strategies can be easily implemented and integrated. Furthermore, we extend the pattern to support chained ranking strategies, in order for the resulting

ranking to reflect a multi-criteria evaluation. Each ranking criterion can use both the context information of the instances gathered during the discovery process, and the context information extracted on the user side. Thus, instances can be ranked against each other and/or against the context of the user (e.g., his location). The output of the ranking process is an ordered list of running services offered by resources on smart things corresponding both to the extended keywords and to the requirements in terms of context expressed by the user either implicitly or explicitly.

Lookup Process Summary

A simplified summary of the complete lookup process is provided in Figure 2.15. First the client (e.g., a user or client application) sends a query request to the LLDU of his choice. Then, the LLDU can contact the Query Extension service which will enrich the user query with a number of related keywords extracted from relevant Web services. Then, these keywords are packed with the query and sent to the relevant LLDUs, which LLDUs are relevant is determined by the type of query. This initiates a recursive tree exploration. Eventually, a result set is returned to the LLDU where the query started. There, the LLDU can use the Ranking service which will sort the results based on a chained list of ranking strategies and on the user specified (or extracted) context. A ranked list of resources (and their provided services) is returned to the client.

Software Implementation

To facilitate integration with Smart Gateway framework presented in Chapter 2, Section 2.1.2, the Lookup and Discovery infrastructure is implemented as several OSGi bundles that communicate with each other via OSGi declarative services (OSGi DS). The integration of these services to the Smart Gateway framework is shown in Figure 2.16.

Basically, an LLDU can run on any machine. As mentioned before, physical LLDUs are coupled with Smart Gateways to simplify the deployment virtual LLDUs, that do not need physical access to the devices can be deployed anywhere.

The implementation is based on 5 internal services (Registry Service, Lookup Service, Infrastructure Service, Discovery Service) that are to be deployed with each LLDU (virtual or physical). The two other services (Query Augmentation Service and the STM Model Translation Service) are ideally deployed outside (e.g., on the Web) because there is no benefit to run them locally as all LLDUs can use the same instance of these services.

2.2.3 Evaluation

We structure the evaluation into two parts. First, in a quantitative evaluation we analyze the response time when querying the lookup infrastructure. Then, we evaluate the

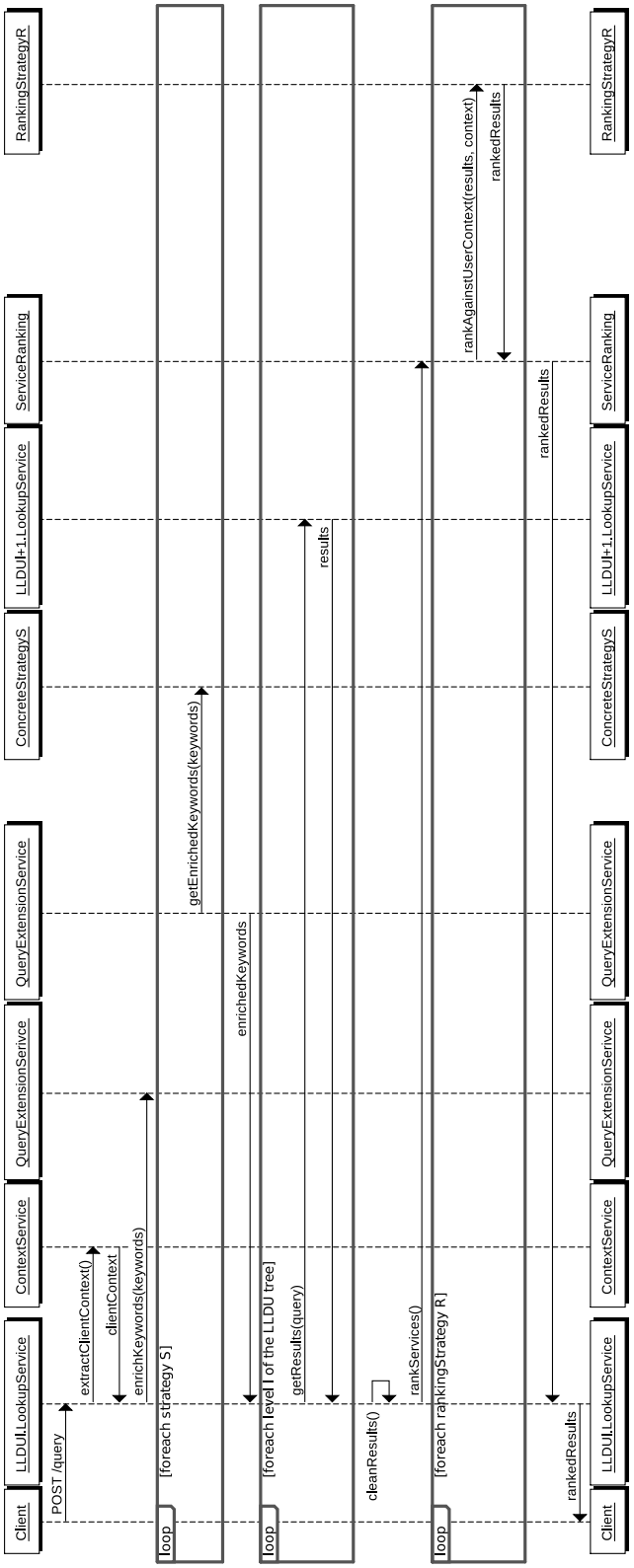


Figure 2.15: Sequence diagram of the lookup process. Clients contact the `/query` resource on an LLDU, the keywords of the query can then be extended using the `QueryExtension` service. Once a lightweight ontology of keywords has been extracted, the query is distributed along the LLDU tree and results are aggregated and ranked before sending them back to the clients.

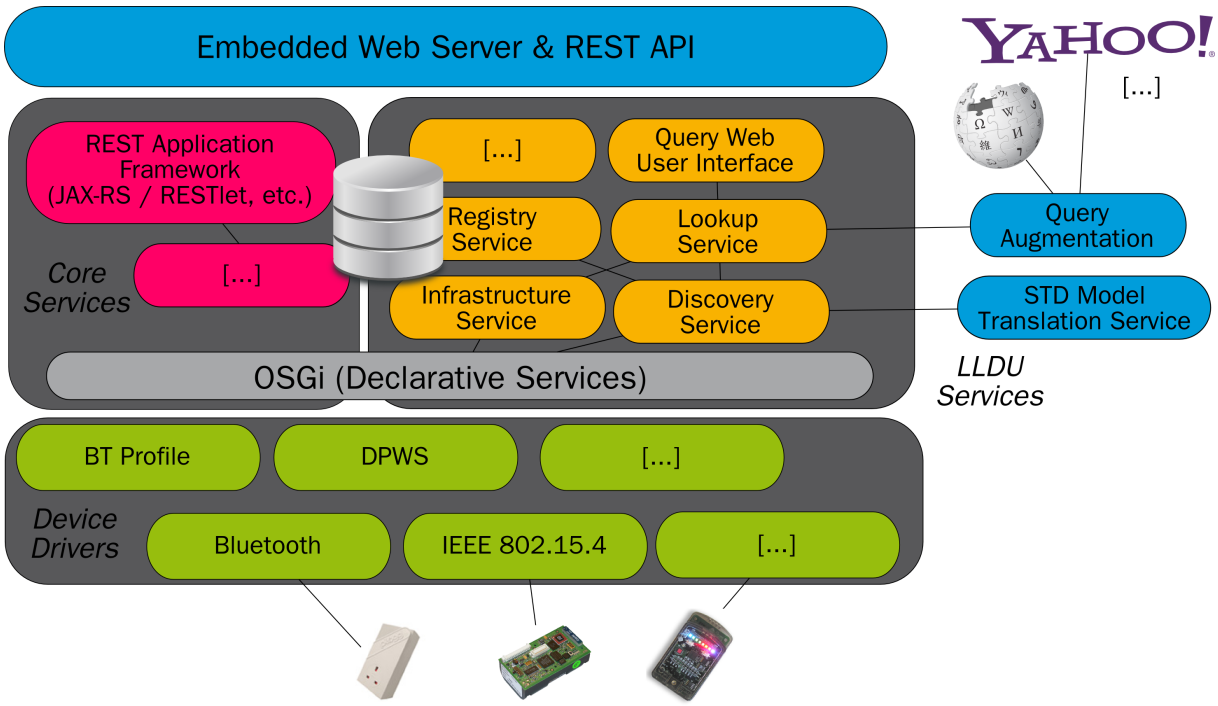


Figure 2.16: The modules of the Lookup and Discovery Infrastructure are integrated in the Smart Gateway framework through OGSi bundles. A LLDU is formed of 5 main services that are implemented as OSGi bundles running locally. Two additional services (Query Augmentation and Translation Service) can run outside the local environment (e.g., on the Web) as they can be used by several LLDUs.

Query Extensions mechanism with real-world data generated by 17 experienced developers during a user-study [77].

Evaluation of the Lookup Service A scenario was implemented in order to assess the feasibility of running service lookup queries on top of proposed distributed infrastructure of LLDUs.

The tree structure of the implemented scenario is shown in Figure 2.17. Each node in this three represents an instance of an LLDU. However, all instances were deployed on the same machine located in the `cnb` building at ETH Zurich. A Smart Gateway is deployed in this LLDU and connected to two sensor nodes (Sun SPOTs nodes, see Chapter 3). One sensor node binds itself to the `europe/ch/ethz/cnb` LLDU and one to the `europe/ch/ethz/cnb/h/107-2/` LLDU. As a consequence, their resources trees becomes part of the overall resource tree of the infrastructure as show in Figure 2.17. However, the depth of the tree used in a lookup comprises only the hierarchy of LLDUs and thus has a maximal depth of 6 because the rest of the actual smart things resources trees where already indexed upon discovery by LLDUs. To generate some noise, a total of 61 virtual resources were attached to the virtual and physical LLDUs.

The machine on which the LLDU resources tree is deployed is a Linux Ubuntu Intel dual-core PC 2.4 GHz with 2 GB of RAM. The Web server used for this implementation is

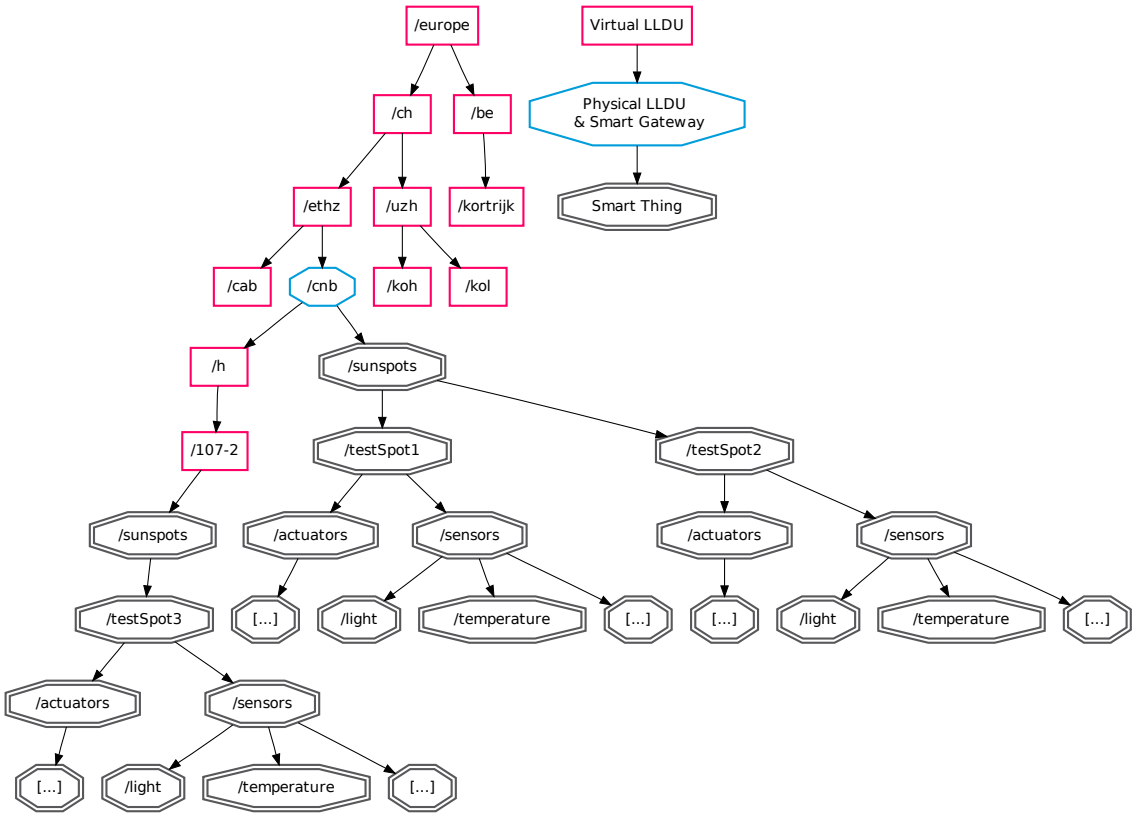


Figure 2.17: Tree representation of the LLDU infrastructure deployed for the evaluation. The `/cnb` node is a physical LLDU to which two concrete wireless sensor nodes are connected.

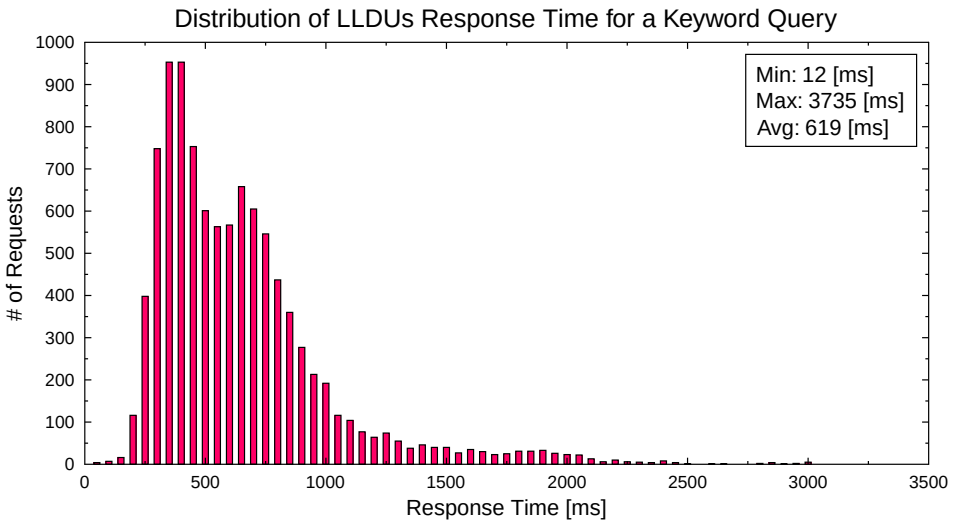


Figure 2.18: Response times when running a keyword query on the test deployment. Most queries get answered within 250 to 750 milliseconds.

based on the Noelios Restlet Engine 1.1.7 [268].

We perform 10000 keywords queries looking for services matching the *light* keyword. With this setup, the minimal observed response time is 12 ms, the maximum 3753 ms with an average response time of 619 ms as detailed in Figure 2.18.

The aim of this evaluation is not to prove that our implementation is performing best but rather to illustrate that the response times are reasonable. It is worth noting however, that in this scenario all LLDUs were run by the same machine and network latency between the LLDUs of an infrastructure would have to be taken into account in a real-world deployment.

Evaluation of Types Query and Candidate Search In this second part we evaluate the impact of the proposed query extensions mechanisms on the search for services provided by smart things.

In order to have a neutral base of smart things and their services on which to perform the evaluation we selected seventeen experienced developers and asked them to write the description of a selected device and of at least two services it could offer. The developers were given the documentation of a concrete device according to the projects they were currently working on. Based on these descriptions we generated thirty types of services offered by sixteen different smart things ranging from RFID readers to robots and sensor boards. Out of these, 1000 devices were simulated on a host PC.

It is worth noting that the STM model based service descriptions were generated as DPWS metadata [77]. However, as the expressive power of the microformat implementation of the STM model is greater than what can be expressed with DPWS metadata and as a STM

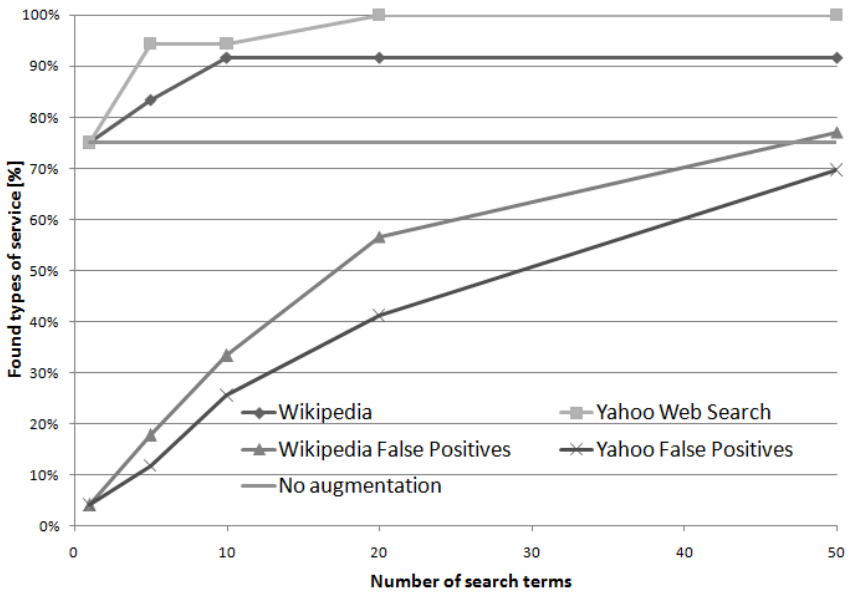


Figure 2.19: Results for the Query Augmentation with Yahoo! and Wikipedia, the Query Augmentation has a positive impact on the number of services found but it also generates more false positives.

Translation Service translates all metadata formats into a single internal representation, the results are applicable to any implementation of the STM model.

The main idea of the evaluation was to find out whether:

1. Augmenting users' input with related keywords could help in finding more services on smart things.

2. What type of combination of query strategies is the most suitable.

Two types of strategies were used. In the first we used a human generated index (i.e., Wikipedia), and in the second a robot generated index (i.e., Yahoo! Web Search). The input keywords were selected by seven volunteers, all working in IT. They provided seventy-one search terms (composed of one to two words) reflecting what they would use if they were to search for services provided by the seventeen smart things when wanting to develop new applications with these smart things. These terms were entered one by one and all the results were logged.

The trends extracted from these experiments is shown in Figure 2.19. Two results can be drawn. First the Query Augmentation process does help in finding more smart things services. Without augmentation 75% (plain gray line in Figure 2.19) of the resources corresponding to the queries were found and using the Query Augmentation up to a 100%.

However, the Query Augmentation generates a number of false positives, i.e., resources that are returned even if they are not related to the provided keywords (depicted by the two lines at the bottom of Figure 2.19). Thus we need to restrict the number of keywords added to the initial ones. The observed optimum is between 5 and 10 added keywords,

leading to less than 20% false positives out of 95% services found. The second result can be seen in Figure 2.19 which reveals that using Yahoo!, the approach performs slightly better than when using Wikipedia.

Looking more at the details we see that approximately 50% of the keywords used against Wikipedia did not lead to any page, simply because they do not have yet dedicated articles, even if Wikipedia is growing at a rate of more than 1000 articles per day (as of 2011) [272]. However, when results where extracted from Wikipedia pages they were actually more relevant for the searched real-world services. Thus, a good solution would be to chain the strategies so that first human generated indexes are called and then robot generated ones, in case the first part did not lead to any results.

The Ranking Service Lookup was evaluated based on a proof of concept implementation. We tested two chained ranking strategies for the generated services; one comparing service health and given weight of 30% as well as one comparing network latency and given a weight of 50%. They performed as expected, sorting the lists of retrieved service instances according to the ranking strategies which, we believe helps users finding their way across the results, but would need to be tested with neutral volunteers.

We implemented the sorting using the merge sort algorithm which has a complexity of $O(n \log n)$, and since the strategies can be chained we have an overhead for the ranking of $O(mn \log n)$ where m is the number of strategies and n the number of resource descriptions.

2.2.4 Summary and Applications

In this section we proposed a metadata model for describing smart things and their services. Furthermore, we proposed an implementation of the model based on microformats that are well understood on the Web for example by search engines. In Chapter 3 (see Section 3.1.2) we apply this model and its implementation to the description of a general purpose wireless sensor platform and illustrate how it can be leveraged to dynamically render UIs for interacting with smart things or to make them searcheable on the Web and in our lookup infrastructure. Furthermore, we will see the benefits of such a model in the next layers, the Sharing Layer and the Composition Layer.

We also presented an infrastructure that can be deployed together with Smart Gateways in order to encapsulate the abstract location of smart things as well as to offer a localized discovery and lookup infrastructure. A concrete usage of this infrastructure is evaluated in Section 3.1.2 as well.

2.3 Sharing Layer

With the Device Accessibility Layer of the Web of Things Architecture we ensure that digitally augmented everyday objects are seamlessly integrated to the Web. With the Findability Layer we enable humans and applications to find the smart thing's services they look for directly from the Web and leveraging contextual information.

Enabling this model for the Web of Things requires a sharing mechanism for smart things, by allowing access to services offered by devices as Web resources. An implementation of the two previous layers fulfills this requirement as devices become openly available to the world directly from the Web and without restrictions. For example, one could share the energy consumption sensors in his house with the community. However, since these devices are part of our everyday lives, their unrestricted public sharing might result in serious privacy violations [139, 118]. In this section we propose a Web architecture that tackles these challenges.

2.3.1 Requirements for a WoT Sharing Platform

HTTP already provides authentication mechanisms for securely sharing resources. The HTTP Basic Access Authentication [102] is a method that allows Web clients (and in particular Web browsers) to provide credentials (user names and passwords) when making an HTTP request on a server. In practice, HTTP Basic Access Authentication [102] is coupled with SSL/TLS in order to make sure that the user names and passwords are not transmitted in clear text over the wire. HTTP Digest Authentication on the other hand, ensures that the credentials are always encrypted.

While these two solutions are already available on (embedded) Web servers they present a number of drawbacks. First, when considering a large number of smart things it becomes quite unmanageable to create and share credentials for each of them and for each contact one wants to share with. Then, the credentials used in these systems are often impersonal and do not reflect any social or trust structures already in place. Then, as the shared resources are not advertised anywhere, sharing also requires the use of (unsecured) secondary channels such as sending emails containing credentials to people. Looking at pitfalls of the existing solutions we propose three requirements for a sharing platform for the WoT:

Security The most basic requirement for a WoT sharing platform is to be secure in order to make sure that access to smart things is not granted to attackers.

Ease of Use People are concerned with the security of their private data, for example

in home environments [139]. However, it has been shown that the ease of use of a secure sharing system has a significant influence on its adoption and effective usage [101]. Hence, a WoT sharing platform should be straightforward and easy to use.

Reflect existing trust models The sharing platform should also reflect mental models users are already familiar with [139]. In particular it should as much as possible reflect the existing trust and social models of users.

Interoperability In order not to hinder the benefits of adopting an interoperable Web architecture, as for the Device Accessibility Layer and Findability Layer, the protocols used by the sharing platforms should be interoperable with the Web and understood by most Web tools and clients. Furthermore, the sharing platform should not be bound to a specific social model but should be able to adapt to several systems.

Integrated Advertisement A WoT sharing platform should also support advertising the shared things directly on the Web. In order to reduce the load for users and improve security, sharing a smart thing and advertising the fact that it was shared should occur on the same channel without explicitly disclosing credentials.

First meant for creating groups of people and enabling communication amongst these groups, social networks rapidly evolved into data sharing hubs [19]. Social networks make it very easy to share data (e.g., pictures) with groups of people such as family and friends [141]. The social network takes care of the authentication of these individuals and manages *access control lists* for the users' data.

We propose leveraging social networks as sharing hubs for smart things. In the Sharing Layer we introduce an architecture and its implementation in a platform that enables the selective sharing of smart things. It uses social networks and their social graphs already in place for sharing smart things with people relevant to smart things owners, creating a *Social Web of Things*.

Furthermore we illustrate how social networks can be used as service advertising platforms and how they support the implementation of the physical feeds aggregators components of the Web of Things Architecture.

2.3.2 Social Access Control: An Architecture for the Social Web of Things

A promising solution to the problem of sharing smart things is to leverage existing social structures and build upon social networks (e.g. Facebook, Linkedin, Twitter, etc.) and in particular their social graphs accessible through data access APIs (e.g., OpenSocial) and their authorization APIs (e.g., OAuth).

Using social networks enables users to share things with people they know and trust such as relatives, friends, colleagues, fellow researchers, etc. This is achieved without the need

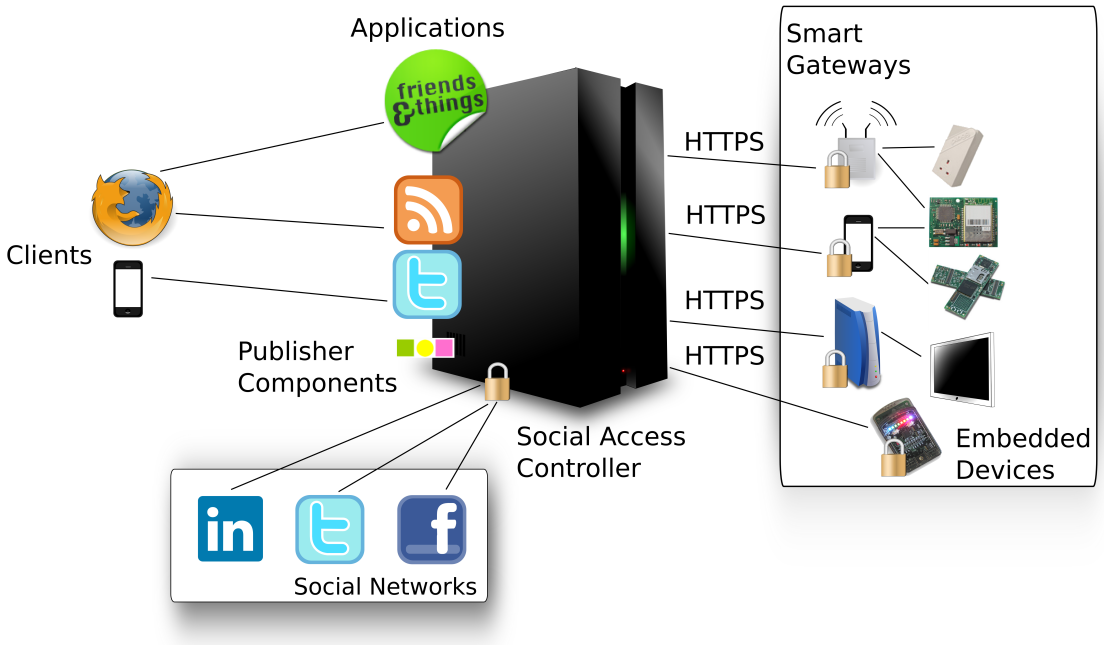


Figure 2.20: Simplified components architecture of the Social Access Controller. SAC serves as authentication proxy between clients and embedded devices. It holds the credentials for accessing smart things and provides access to selected trusted connections of the owners' social networks. It further offers an API upon which applications can be built.

to recreate yet another social network or user database from scratch on a new online service. Additionally, it enables advertising and sharing through a unique channel: users can tell their friends about the sensors they shared with them by automatically posting messages to their profile or newsfeeds.

We propose a system to share things and facilitate access to real-world services offering a RESTful Web API. Our core contribution is a Web architecture and its implementation called Social Access Controller [71, 131] (SAC) which offers the following functionality:

Authentication Proxy A SAC identifies users based on existing credentials rather than requiring the creation of new, impersonal credentials for each smart thing or Smart Gateway.

Authorization Proxy A SAC is an authorization proxy that sits between clients and smart things and authorizes clients applications (e.g., browsers) to access the smart things.

Access Control Manager A SAC helps users to fine-tune the nature of interactions they want to allow for their objects (e.g., read-only, read-write, etc.) and manages access control based on existing social graphs.

Advertisement Channel A SAC can advertise shared smart things using the notification services of social networks such as user newsfeeds or walls.

Overall, the architecture enables owners of Web-enabled smart things to easily share them on the Web. Consider for example an smart meter that implements the Device Accessibility Layer. Sharing the energy consumption recorded by all these smart meters on the Web, enables the creation of very interesting applications. For instance a mashup on a map can show the the consumption of each individual in a group of friends. Similarly, a Web-enabled Hi-Fi system can enable songs to be played remotely through a RESTful interface. Sharing it with close friends enables them to remotely play songs for you. You can also use the system to inform all your friends, on their favorite target device (e.g., mobile phone, laptop, TV, etc.) that you will be a little late. The global system architecture shown in Figure 2.20 addresses these use-cases. It is composed of a central Web application, the SAC server, which creates the link between social networks and smart things.

In this section, we define *owners* as people owing or administrating smart things (e.g., a Web-enabled sensor node) and *trusted connections* as the people owners share their smart things with (e.g., friends, colleagues or relatives). It is worth noting, however, that owners and trusted connections can also be applications.

2.3.3 Retrieving the Owners' Social Graphs

The process of sharing smart things with trusted connections occurs in three phases. In the first phase, the owner accesses the SAC server by logging in using one or several of his social networks credentials as shown in the step 1 of Figure 2.22. Then, the owner's lists of

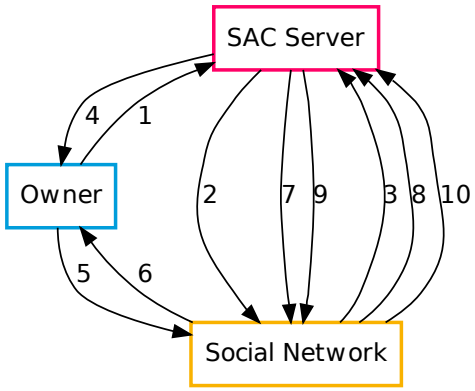


Figure 2.21: Using OAuth for a SAC to retrieve an owners' trusted connections: 1) Click on social network button 2) RequestToken? 3) OK, RequestToken + RequestSecret 4) Redirect to social network with RequestToken 5) Login + Grant permission to SAC 6) OK, Redirect to SAC Server 7) Exchange RequestToken to AccessToken? 8) Owner login OK, AccessToken + AccessSecret 9) List of Friends? 10) AccessToken + ConsumerKey OK, List of Friends

trusted connections (i.e., social graphs) need to be retrieved using delegated authorization on the social networks as shown in step 2 of Figure 2.22.

Leveraging Web Authorization Protocols: OAuth

As mentioned before a SAC is an authorization and authentication proxy between clients (e.g., Web browsers) and the smart things. Rather than maintaining its own database or list of trusted connections and credentials – as it would be done with Basic or Digest HTTP Authentication – it connects to a number of social networks to extract all potential users and groups one could share with. As a consequence owners need to be authenticated to their social networks and a SAC need to be authorized to retrieve lists of trusted connections.

To achieve this first step, we use the OAuth 1.0 protocol [48]. OAuth is a delegated authorization protocol that enables users to allow client applications to access their data without revealing the users credentials [8]. OAuth has the advantage of fulfilling our requirements for a Social Access Controller. First, it is supported by a vast majority of social networks. Secondly, as an open standard it is backed by a large community of developers and security experts and is being monitored for security breaches which makes it a rather secure option. Thirdly, it is built on top of HTTP and thus is well integrated to the Web and interoperable. Finally, since the user-facing part of the protocol is relying on the existing login systems of social networks, users are familiar with them and thus the system can be considered as relatively easy to use.

The OAuth protocol is based on a so called three-legged scheme because there are three parties involved in the protocol [8]. The client, the service provider and a user. In

the context of a SAC, the client is the SAC server, the user is the owner of a smart thing and the service providers are the social networks the owner has an account with. The communication to authenticate the user and authorize a SAC to access the trusted connections of the owner is shown in Figure 2.21 and detailed here:

1. The SAC server gets a `ConsumerKey` and `ConsumerSecret` from the social network.
2. The owner selects a social network by clicking on its corresponding button on the SAC server login page.
3. The SAC server asks the selected social network for a `RequestToken`.
4. The `RequestToken` is granted together with a `RequestSecret`.
5. The owner is redirected to the social network where he logs in and grants the permissions to the SAC server.
6. The login was successful and the owner is redirected to the SAC server.
7. The SAC server asks to exchange its `RequestToken` for an `AccessToken`.
8. Since the owner login was successful the `AccessToken` is granted by the social network.
9. The SAC server can now request owner related data from the social network.
10. The data is granted since the SAC has a valid `AccessToken` and `ConsumerKey`.

Ensuring Statelessness It is worth noting that in order to respect the constraint of Stateless Interactions of RESTful architecture described in Section 2.1, a SAC should not store the access tokens and secrets given by the social networks upon successful authentication and authorization. Indeed, this ensures that requests to a SAC can be cached and proxied [160]. Hence, the SAC requires clients to store this information. This is best achieved by storing it in the form of cookies [114]. These cookies are then sent in the header of each subsequent request to a SAC. As a consequence, the transmission of this information should occur over an encrypted channel (e.g., HTTPS) as an attacker could use the unencrypted tokens to impersonate a user.

However, an attacker could not use these tokens to compromise and use the user's social data. Indeed, access to the data is only granted to particular server application here, a SAC server. This is ensured through signing the request with an API secret key only known to the SAC server.

Leveraging Social Network APIs

OAuth is meant to authorize applications to access, on users' behalves, other applications such as social networks. However, the specification does not propose a standard way of accessing the social network data once authorized and authenticated, it does not standardize the reads/writes API of a social network. Nevertheless, providing an open Web

API is one of the success factors of social networks themselves. Indeed, these APIs allow third-parties Web applications to be built using partial data extracted from the social networks and thus to enhance the functionality and value of the social networks.

OpenSocial The OpenSocial [88] specification was created to fulfill the need for standard social network APIs. It defines a common API for application to access data across several social networks. OpenSocial uses OAuth for authorizing an application to access the social network. Then, the access to the standard social API is using a REST or RPC architecture.

When building a Social Access Controller, this type of standard is central as it enables to retrieve lists of trusted connections from any compliant social network and thus keeps to architecture open for integrating new and existing social networks. Once authorized to access the social data with OAuth, a SAC server has a standard way of accessing trusted connections. As an example, the OpenSocial RESTful API [299] call for downloading all the contacts of a user is a `GET` request on: `/people/{USER_ID}/@all`.

Unfortunately, some major social networks such as Facebook or Twitter do not comply with OpenSocial. For these networks, proprietary APIs such as the Facebook (Connect) API have to be used. While similar to OpenSocial in terms of functionality, the APIs significantly differ, making it impossible to access the data of these networks in a uniform manner.

Thus, we suggest for SAC servers to support access to social network data using a plugin architecture, enabling the support of both OpenSocial based and proprietary APIs. We further describe this plugin architecture in Section 2.3.7.

For each social network a user is currently logged in with, the SAC server uses the Web API of the social networks (through an OAuth authorization) and queries them for lists of friends and other connections as shown in Figure 2.22. All these connections are then compiled into a global list of potential connections that the owner can share with.

2.3.4 Registering and Sharing Smart Things and Smart Gateways

In the second phase of the sharing process, the owner registers the smart things and Smart Gateways he owns. The prerequisites for a smart thing to be shared with a SAC are based on a relaxed subset of the Device Accessibility Layer presented in Section 2.1 that can be summarized as follow:

- **Addressability:** All the shareable functions offered by smart things should be modeled as resources [50] which are addressable and identified by resolvable URIs.
- **Uniform Interface:** The actions available on resources should comply with the HTTP verbs (e.g., a `GET` on a resource retrieves a representation of that resource).

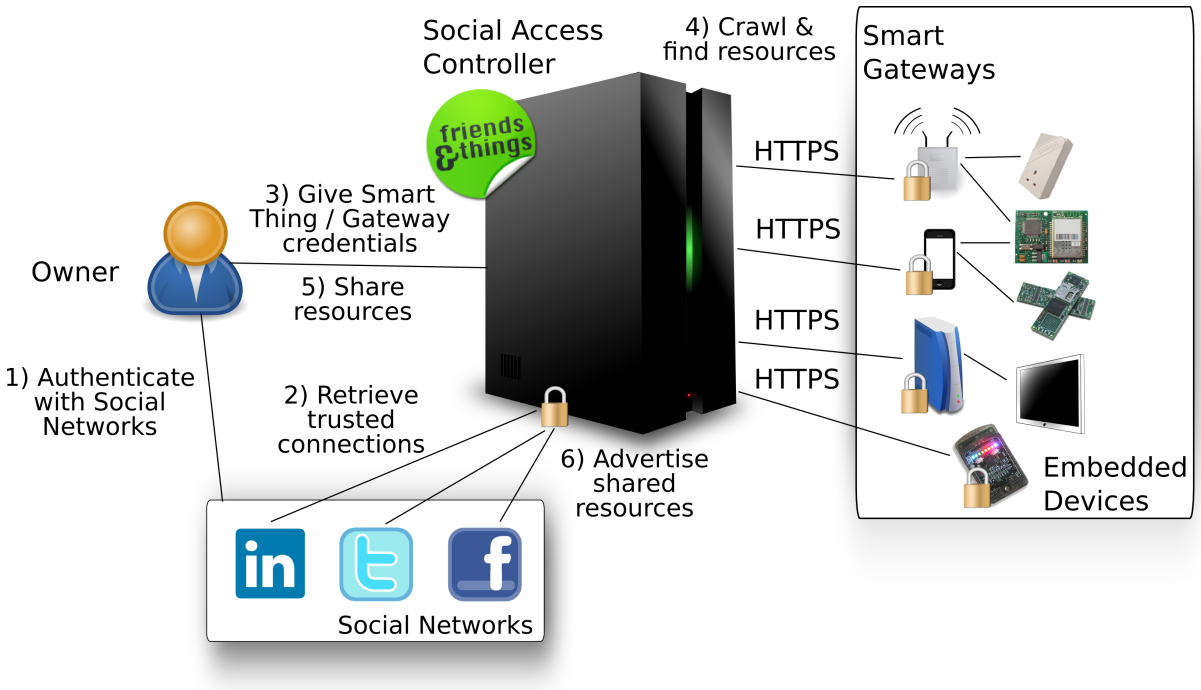


Figure 2.22: Process for registering and sharing a smart thing using a SAC server. The owner authenticates himself using a social network (1,2). He then provides his credentials to access the smart thing (3). The smart thing is crawled for resources (4) that the owner can share (5). The shared resources are advertised on social networks (6).

- **Resource Description:** The embedded Web servers on smart things (or Smart Gateway) should support one of the service metadata description methods proposed in Section 2.2.1. It is worth noting that the only real requirement is to respect the *connectedness* constraint as the core information required to share smart things with a SAC can be obtained simply by crawling the RESTful Web API. Additional metadata can be used to provide owners and trusted connections with more detailed descriptions of the resources thus improving the system's usability.

Additionally, to ensure that the smart things are secured, their direct access should be restricted as shown in the rightmost part of Figure 2.20 using a standard HTTP method (e.g., HTTP Basic Authentication with SSL/TLS or HTTP Digest Authentication). If not provided "out-of-the-box", this can be done by setting up the Web server at the device level or at the gateway level to accept only authenticated HTTPS traffic and require credentials for any incoming request.

The actual registration and sharing process is depicted in Figure 2.22. First, the owner logs in to one or more of his social networks using the SAC cross-network OAuth client. He can start registering the smart things and Smart Gateways that belong to him and sharing them with the trusted connections retrieved by the SAC server (step 2 in Figure 2.22). To do so, he provides the credentials to access a smart thing or the credentials of a Smart Gateway that manages several smart things as shown on step 3 of Figure 2.22.

Using these credentials, the SAC server accesses the smart things and crawls them (step 4 of Figure 2.22). This is done by using an STM Translation Service (see Section 2.2.1). Using this service, the SAC server is able to identify the available services and expose them for sharing in a user-friendly manner as show in Figure 2.28.

The owner can then share the discovered services of smart things with trusted connections such as friends, relatives, or colleagues (step 5 of Figure 2.22). He can either share complete smart things (e.g., a sensor node) or their sub-resources only (e.g., the temperature sensor of the sensor node only). Furthermore, he can choose to share resources in read-only (i.e., allowing the `GET` verb only) or read-write (i.e., giving access to all the available HTTP methods). Figure 2.28 shows a user interface to share smart things implemented on top of a SAC server.

Finally, for each shared resource of a smart thing, the SAC server uses the corresponding social network messaging system to post a notification to the trusted connection the resource was shared with as shown in step 6 of Figure 2.22. As for retrieving lists of trusted connections, this is done through the social network API. In the case of an OpenSocial compliant social network, this is done simply with a `POST` request on: `/messages/{USER-ID}`.

In the case of other social networks, the SAC server has to use the proprietary API for sending notifications through the network. In our implementation, for Facebook, the SAC server publishes a message to the newsfeed of the trusted connection. In the case of Twitter it simply tweets a message to the trusted connection. Note that the posted message does not contain any credentials but a link pointing back to the SAC server where the data of the shared resource can be fetched by the trusted connection once authenticated.

2.3.5 Accessing Shared Smart Things

Once a trusted connection gets notified of the fact that resources of a smart thing were shared with him, he uses the provided URI to access it as shown in step 1 of Figure 2.23.

The shared URI points back to an instance of a SAC. When receiving the HTTP request, the SAC server prompts the trusted connection for log in if no cookie corresponding to a successful authentication on one of the SAC recognized social network is found. Indeed, just as smart thing owners need to be authenticated and to grant access to their social network data, trusted connections wanting to access the shared smart things need to get authenticated. Because it authorizes applications in the name of users, OAuth can also be used to authenticate trusted connections. However, as trusted connections simply use the system as a proxy there is no need for a complete delegated authorization process.

In this case, the SAC server simply needs to confirm that the trusted connection is the person it pretends to be as shown in step 2 of Figure 2.23. A simple, user-friendly manner to ensure this on the Web is through delegated authentication.

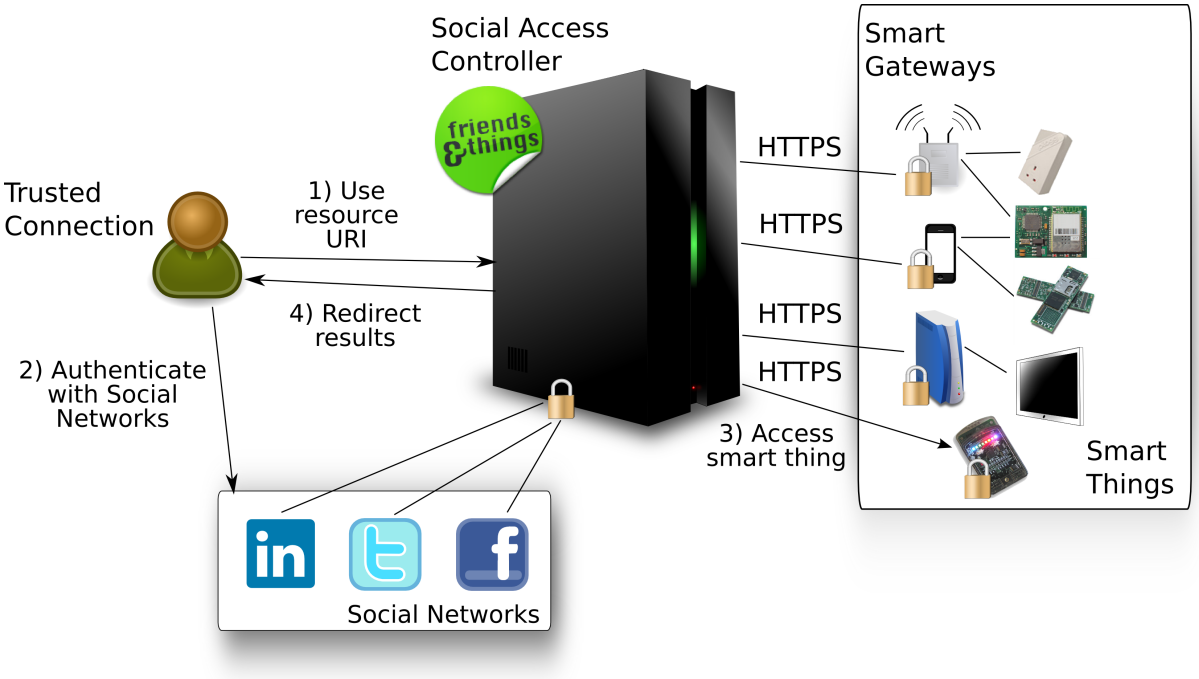


Figure 2.23: Accessing shared objects using delegated authentication through the Social Access Controller. The trusted connection requests the shared resource's URI (1). If not logged in with the corresponding social network, the SAC server asks the trusted connection to login (2). The SAC server then access the smart thing and redirects the results to the trusted connection's client (3, 4).

Leveraging Delegated Authentication A delegated authentication for a SAC presents two advantages. First, trusted connections do not need special credentials or a dedicated registration for accessing the shared resources, as they can use the credentials of any social network or service on the Web that supports delegated authentication. Second, the SAC does not need to hold profile information about the users (a user ID is enough) and can support several social networks for a single trusted connection.

OpenId is the dominating protocol for delegated authentication on the Web. Its core idea is to offer Web users a mechanism for transporting their identity from site to site, avoiding for them to have to go through a registration process for each site. Unlike OAuth, which is both a user authentication and an application authorization protocol, OpenId does not grant data access to the client Web site beyond a limited user profile.

After the SAC server successfully confirmed the identity of the trusted connection using a delegated authentication client (OpenId or OAuth), it internally checks whether this person also has access to the requested resource. If it is the case the SAC server logs on the shared resource using the credentials provided by the owner when registering the resource. It then redirects the HTTP request of the trusted connection to the shared resource as shown in step 3 of Figure 2.23. Finally, it redirects the result directly to the HTTP client of the trusted connection (step 4), e.g., to a Web browser.

2.3.6 Physical Feeds Aggregation

While direct access to a single device might be interesting for control scenarios, as for instance playing a song on a Hi-Fi system or turning off a device remotely, monitoring use-cases require a system that allows to group several events coming from smart things together and publish them to a messaging or feeds server on the Web.

Thus, SAC provides a syndication mechanism that can be used to monitor several smart things at once. It consists of an `Updater` component which periodically polls the smart things for updates and sends the updates to a syndication server (e.g., an Atom server). `Updater`s can be parameterized by specifying the amount (number of characters or percentage of change) of content that should be changed in order to generate a new event. A regular expression which should be satisfied can also be specified. Finally, another regular expression can be used to reformat the content of the event before publishing it. The new events are then published by the `Publisher` components which are abstraction of Web publishing mechanisms. Similarly to the `NetworkConnector`s, `Publisher`s rely on an extensible architecture to be able to quickly add support for new services.

An example scenario for this system is a friend who can be informed when you leave work. By monitoring the energy consumption of your computer, a notification will be generated and transmitted when your computer is turned off. Another scenario is a friend who creates a simple physical mashup with Google Maps that shows friends available in the neighborhood. This mashup could be simply based on an Atom feed to which the `Publisher` Component sends update events whenever a friend leaves his workplace.

2.3.7 Software Architecture

In this section, we present the software architecture of our SAC implementation, based on a Resource Oriented Architecture implemented using the Object Oriented paradigm.

Our architecture further uses the EJB (Enterprise Java Beans) patterns [143] and thus is organized around a data model describing the actors, or entities of the system, managed by a number of managers implementing the business logic. We first present the data model and then the most important manager components.

Data Model A SAC server is implemented around a relatively small set of data classes, also called `Entities` in the Object Oriented terminology, that we briefly describe here.

Administrators and Gateways The primary purpose of SAC is to enforce rules for accessing WoT devices. Hence, the `Gateway`s data classes represent either actual Smart Gateways or smart things since they both expose the same type of Device Accessibility Layer. Owners of smart things i.e., users who registered the smart things, are called `Administrator`s.

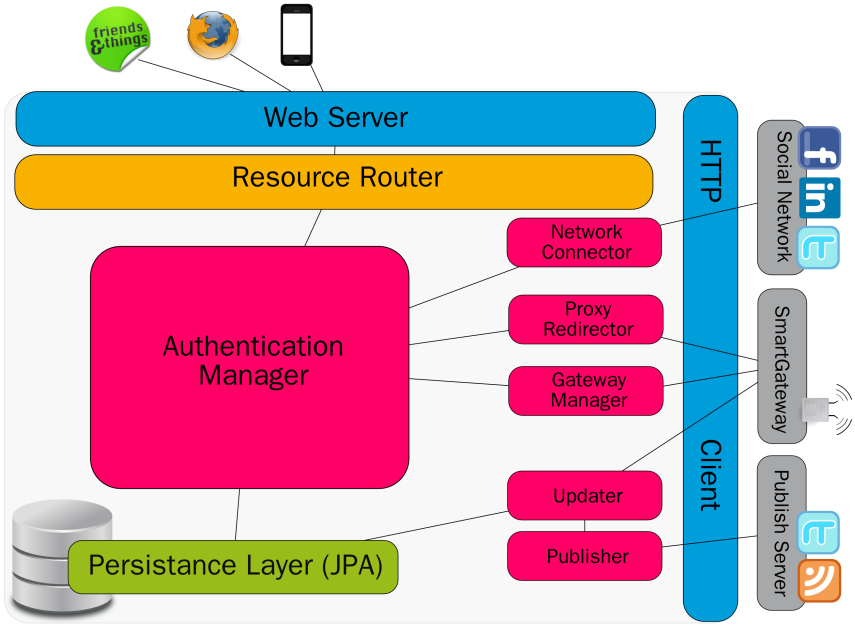


Figure 2.24: Overview of the SAC software architecture. At the core of the architecture, the `AuthenticationManager` is responsible for managing a list of the connected users. The `NetworkConnectors` are used to authenticate and authorize users with all supported social networks. The `GatewayManager` manages the Smart Gateways and the `ProxyRedirector` fetches resources from the Smart Gateways in the name of trusted connections. Finally, `Updaters` and `Publishers` manage the feeds.

Social Networks and Accounts `SocialNetwork` entities represent social networks that can be used for authentication and authorization. These entities are linked to users through the `Account` classes. `Account` classes represent all the social network a user (owner or trusted connection) logged in successfully with using the SAC OAuth client.

Resources and Shares These entities represent the smart things. A `Resource` is a shareable functionality on a smart thing (e.g., the `temperature` resource of a sensor node). Once shared, a `Resource` is linked to a `Share` entity which is turn is linked to `Permission` entities representing whether the resource is shared in read-only or read-write mode.

Publish Systems and Updater `Publishsystems` represent Web services where updates of a resource can be published. A `RegisteredFeed` is a scheduled periodic task that takes care of posting an update to a `PublishSystem` if the new state of the `Share` it monitors meets the specified conditions.

The entities are stored and loaded through a `PersistenceLayer` as shown in Figure 2.24 which is a simplified overview of the software architecture of our SAC implementation. Following the EJB patterns [143], entities are managed by a number of `Manager` components that can be seen in Figure 2.24.

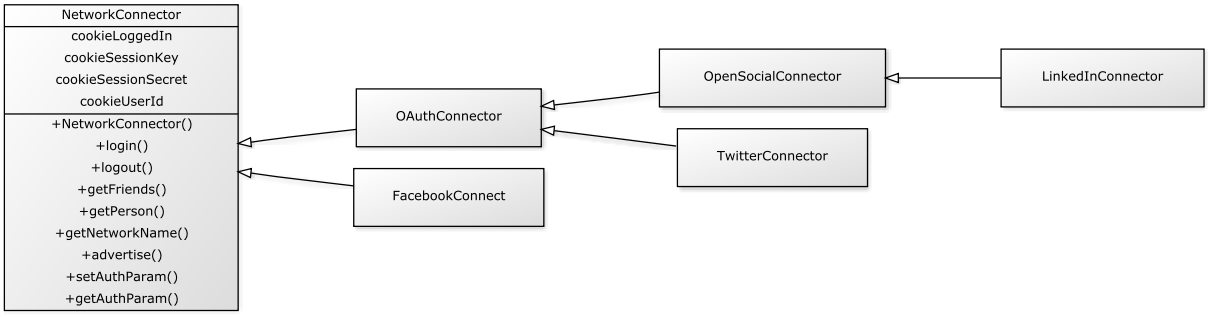


Figure 2.25: Simplified class diagram of the network connectors architecture of SAC. Thanks to the modular architecture new connectors can be easily added to the system.

Business Logic Just as smart things in the Web of Things are accessible through RESTful APIs, access to SAC functionality is done through a REST architecture. As shown in Figure 2.24, incoming HTTP requests reach the `ResourceRouter`. This component is based on a REST framework enclosing an HTTP 1.1 compliant Web server and is in charge of forwarding the incoming requests to the matching components which can be a resource, another router or a guard. A resource component simply corresponds to the implementation of a REST resource. Guards are used to protect sensitive components of the architecture and authorize their access only to authenticated and authorized requests.

At the core of the platform lies the components which implement the business logic of the SAC and use the entities of our data model. We briefly describe the most important business logic components of the SAC architecture:

Proxies and Gateways Managers The `ProxyRedirector` is a central component of SAC as it implements the access to shared smart things as described in Section 2.3.5. The `ResourceRouter` redirects all requests from trusted connections to access smart things to the `ProxyRedirector`. The `ProxyRedirector` then adds the required credentials to the request and forwards it to the secured smart things. It finally returns the results back to the client of the trusted connection. When an owner shares a resource with a trusted connection he also, by default, allows access to all resources below the shared one. As a consequence, the `ProxyRedirector` must also parse the responses and replace all the absolute links (as well as `Location` headers) pointing directly to the smart things in order for them to point to the smart things proxy.

The `GatewayManager` is in charge of securely holding the credentials to access the smart things as well as discovering the services they provide. To do so, it either uses the internal `CrawlingModule` or uses an external STM Translation Service (see Section 2.2.1).

Network Connectors The `NetworkConnector`s are the SAC link to social networks: They encapsulate the necessary logic to login to a social networks using delegated authentication as well as to extract data from it using delegated authorization. While OAuth and OpenSocial compliant networks can be accessed using a generic Network Connector, other networks need dedicated connectors that understand their specific

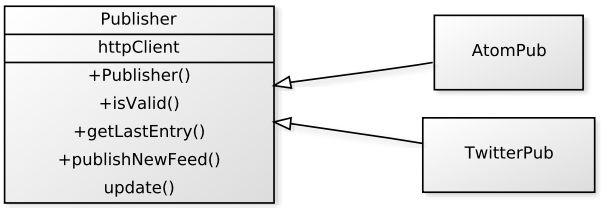


Figure 2.26: Simplified class diagram of the publishers architecture of SAC.

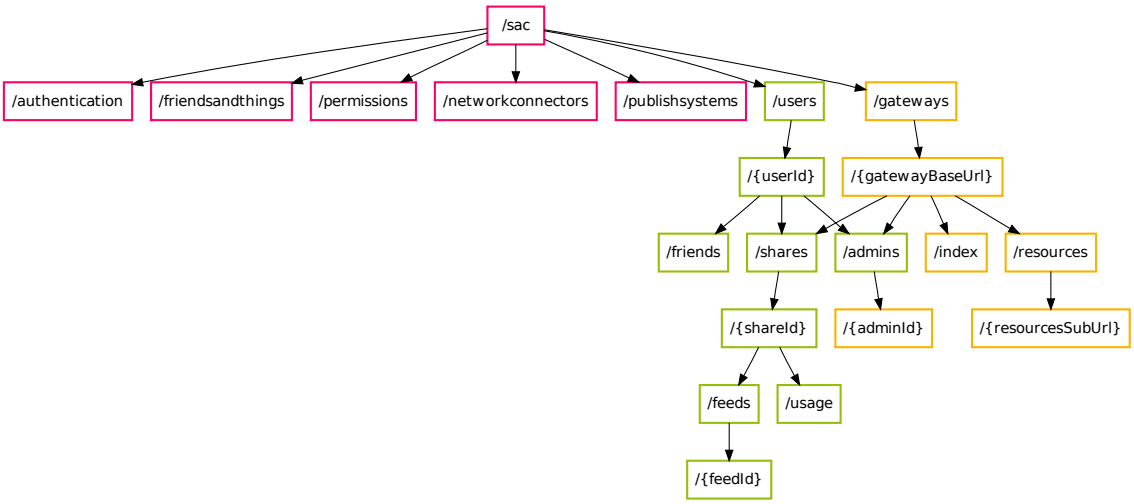


Figure 2.27: Tree visualization of the SAC RESTful Web API.

(non-standard) API. The `NetworkConnector` architecture of SAC is presented in Figure 2.25. It features an abstract `NetworkConnector` which can be extended to created dedicated connectors with OAuth, OpenSocial support or with proprietary authentication and authorization protocols (e.g., `FacebookConnect`).

Authentication Manager The `AuthenticationManager` works between the proxies and the network connectors. It manages a list of the connected users and their respective social networks. As mentioned before, SAC does not store the access tokens and secrets returned by each social network. These are rather sent along by clients in the HTTP header of each request. As a consequence, the authentication manager is responsible for extracting the information from the cookie and using it on the corresponding `NetworkConnector` to ensure that the user is still authenticated.

Updaters and Publishers In order to enable the aggregation of physical feeds from shared smart things, SAC uses an `Updater` which is in charge of monitoring a resource and pushing their updates to a `Publisher`. As shown in Figure 2.26, `Publisher` is an abstract component that can be extended to support a concrete publishing mechanism. We implemented an `AtomPublisher` supporting any AtomPub compliant server and a `TwitterPublisher` to publish notifications through the Twitter service.

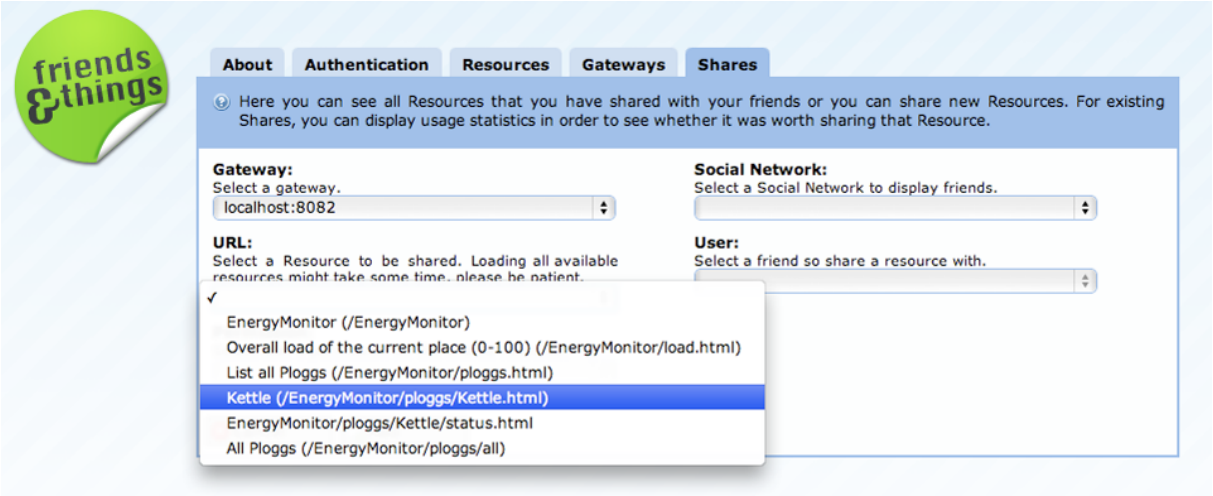


Figure 2.28: Screenshot of the FAT (Friends and Things) application built on top of SAC. The owner can select the smart things he wants to share with some of his trusted connections.

RESTful API In order for applications to leverage it, a SAC server should implement a RESTful Web API. Our implementation of the SAC server offers an API that can be used to: share, add, manage, syndicate, and interact with shared smart things. This makes SAC an integral part of the WoT since its API is accessible on the Web and can be used to further build applications. For example, a new Smart Gateway or smart thing can be added to SAC for sharing by sending a `POST` request to `/gateways` alongside with the data of the new element (URI, name, description, etc.) as payload.

All the resources available in the API are shown in Figure 2.27. Each resource can be delivered as an HTML, JSON or XML representation.

2.3.8 Friends and Things: A Social WoT Web Application

To exemplify how applications can be built on top of the implementation of SAC, we developed the Friends and Things (FAT) application. FAT is a Web application that allows users to share and use shared smart things in a user-friendly manner. FAT is written in JavaScript and HTML and uses the SAC RESTful Web API. It basically provides a user interface to access the main functionality of the SAC server. It helps owners to login with several social networks in parallel, lets them manage smart things and Smart Gateways and share them with trusted connections (see Figure 2.28). Finally, it provides owners with usage statistics (see Section 4.4 for a concrete example).

Furthermore, FAT helps trusted connections leveraging smart things that were shared with them. Upon login, trusted connections find a list of smart things that were shared with them. As shown in Figure 2.29, using the RESTful Invocation Tool, they can directly test all the authorized HTTP methods (e.g., `PUT`, `POST`, `GET`) and add HTTP payloads to their requests. Finally, trusted connections can create feeds combining different smart things that were shared with them and have these feeds automatically updated according to rules specified using regular expressions.

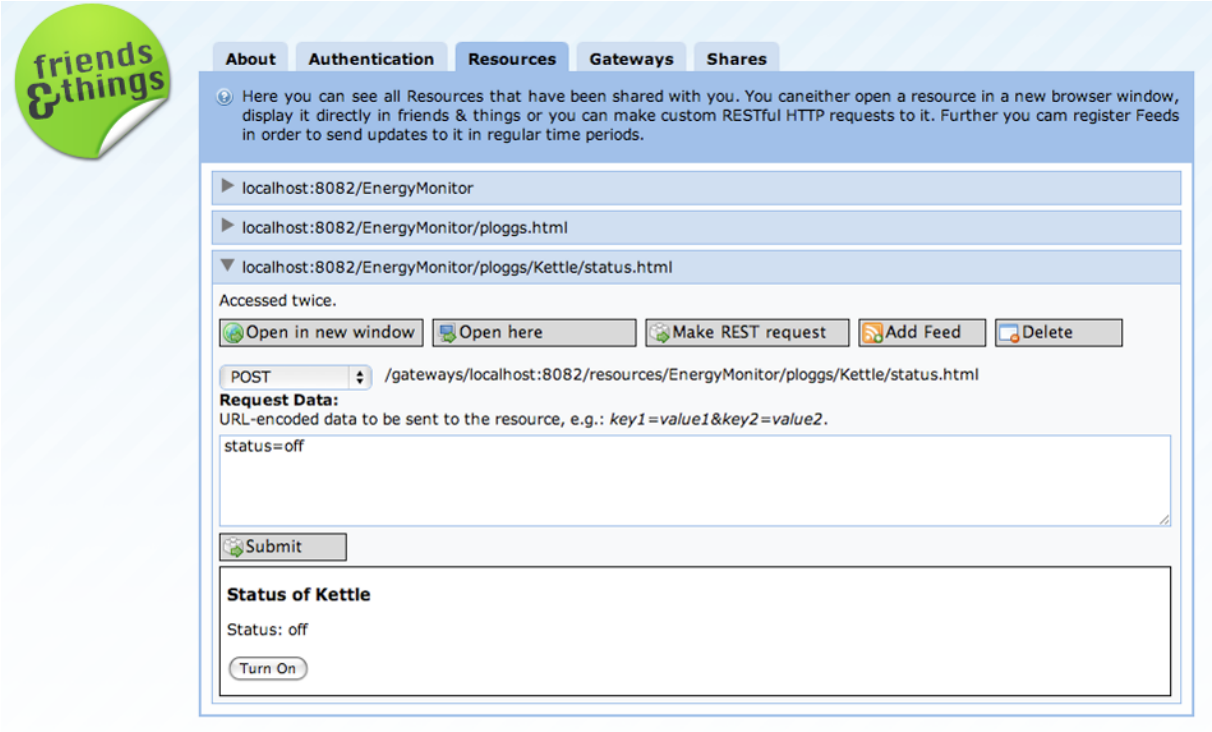


Figure 2.29: Using the RESTful Invocation Tool, a trusted connection can directly test all the HTTP methods available to him and add HTTP payloads.

2.3.9 Summary and Applications

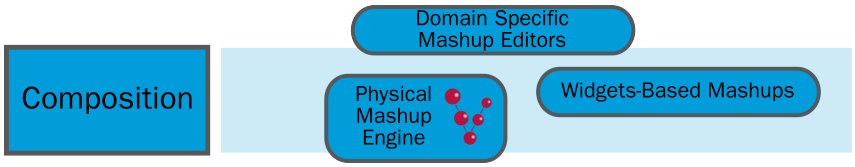
Bringing devices to the Web paves the way for a new breed of applications that seamlessly blend the physical world with existing services on the Web. SAC and FAT are simple examples of the possibilities revealed by Web-enabling physical objects.

In this section we have presented a system for sharing and controlling access to resources in the Web of Things. The core idea is to leverage existing online social structures rather than relying on closed databases of credentials. Thus, the SAC architecture provides a framework which builds upon fast growing social networks such as Facebook, Twitter or LinkedIn to allow users to share physical objects with actual friends, relatives or colleagues.

SAC also provides a programmable basis upon which composite Web applications can built. Thanks to the RESTful API of SAC, physical mashups and other Web applications can use the SAC functionality to share and use shared smart things. To illustrate this we introduced the Friends and Things Web application which directly builds upon the API for our implementation of a SAC server.

The architecture described in this section and its implementation are evaluated with Wireless Sensor Networks in Chapter 3 (see Section 3.3) where the overhead of using the architecture and its underlying protocol is assessed. Finally, it is used to share access to traces of RFID-tagged objects in Chapter 4 (see Section 4.4).

2.4 Composition Layer



At large, the Web of Things materializes into an open ecosystem of digitally augmented objects on top of which applications can be created using standard Web languages and tools. With the previous layers, we allowed developers to access and search for Web-enabled smart things and owners to have a simple and scalable mechanism to share them. Much is to gain from Web integration as it drastically eases the usually rather tedious development of applications on top of smart things.

In this last layer, we would like to push further the boundaries of the WoT so that from getting close to developers, it also gets closer to end-users and enables them to create simple composite applications on top of smart things. Indeed, the previous layers also deliver more flexibility and customization possibilities for end-users.

2.4.1 Physical Mashups in the Web of Things

In this section we look at the concepts of Web 2.0 Mashups and further define the notion of Physical Mashups. We then discuss the special requirements of the Web of Things and propose a Physical Mashups architecture based on these requirements.

Web 2.0 Mashups

Web Mashups are defined as: "Web applications generated by combining content, presentation, or application functionality from disparate Web sources. They aim at combining these sources to create useful new applications or services" by Yu et al. [216].

Yee [215] characterizes mashups along the combinations of three actions or patterns:

1. Data is extracted from a source web site.
2. The data is translated into a form meaningful to the destination web site.
3. The repackaged data is sent to the destination site.

Following this pattern, Housingmaps [241] is one of the most well-known Web mashups. It extracts the list of apartments, rooms or flats that are available for rent or sale on the Craigslist Web site [226] and displays them on Google Maps [237] according to their location [215]. The result is a new service which helps people visually and geographically identifying real estate listings.

Mashup creators often also share their mashups on the Web (sometimes through directories such as the Programmable Web API directory [263]) and expose them through open APIs as well, making the ecosystem grow with each application and mashup.

The creation of composite applications is key in the idea of mashups. However, according to the literature [216, 154, 90, 20], there are several differences between Web mashups and traditional composite applications:

Lightweightness and Simplicity The technologies used for mashups mainly involve Web standards (e.g., HTML, HTTP, Atom, RSS, Microformats), scripting languages (e.g., JavaScript) and Web programming languages (e.g., PHP, Ruby, Python, etc.) [215, 216]. As a consequence mashups are rather lightweight applications that can be brought to several clients through Web (and mobile Web) browsers.

Accessibility to a Larger Public A direct consequence of the simplicity of mashups is their accessibility to a larger public than traditional composite applications. Manual mashups [216], i.e., mashups that are created without the use of dedicated tools are still mostly targeted towards (Web) developers. However, through the use of mashup editors, lightweight Web composition is brought closer to tech-savvies thanks to the use of visual metaphors and wizard assistants.

Prototypical and Opportunistic Nature Traditional composite applications in the enterprise software business are often achieved either using proprietary programming solutions or WS-* services with composition standards such as BPEL (Business Process Execution Language) [97, 154]. On the contrary mashups are often used for more ad-hoc applications such as rapid prototypes or to create applications that fits the needs of individuals or a handful of people with more relaxed quality of service and security requirements [154, 20]. However, in the last few years, mashups have evolved to be also considered as a valid development technique for the world of enterprise applications [97].

Physical Mashups

We propose a unified view of the Web of today and tomorrow's Web of Things in applications called *Physical Mashups* [209, 79, 81]. Tech-savvies, i.e., end-users at ease with new technologies, can create Physical Mashups by composing virtual and physical services. Following the trend of Web 2.0 participatory services and in particular Web mashups, users can create applications mixing real-world devices such as home appliances or sensors with virtual services on the Web. As an example, a Hi-Fi system could be connected to Facebook or Twitter in order to post the songs one listens to the most.

We distinguish three mashup development approaches for Physical Mashups and relate them to their main target groups:

Manual Mashup Development Introduced in [216], we refine this type of development in a WoT context as: development of composite applications that involve smart

things by means of Web technologies such as HTML, HTTP, Atom and JavaScript but without requiring the use of specific mashup tools. This type of development is meant to be undertaken by actual developers. However, thanks to the previously presented layers and approaches, smart things are brought to Web developers rather than embedded systems specialists [146]. We used this type of development approach for instance to realize the *Energie Visible* mashup presented in Section 3.2.2.

Widget Based Mashup Development In this type of development a software framework, sometimes called *portal* communicates with the smart things and makes their data available through a black-board [216] approach where the data are constantly written to variables in memory. The developers then simply have to create widgets (or portlets) that read and write to these variables. These widgets are usually written using a combination of HTML and JavaScript code. Since it fully abstracts the communication with smart things this model is a *higher abstraction level*. A direct consequence of this development model is that domain experts (e.g., experts in supply chain management) with IT skills can build composite applications for their domain without having to learn the subtleties of embedded systems. This development approach was used to create the *EPC Dashboard Mashup* presented in Section 4.5.3.

End-User Development with Mashup Editors This development approach enables end-users to create their own composite applications. In the case of Web 2.0 Mashups this type of application is usually developed through a mashup editor, e.g., Yahoo Pipes [287], which is a Web platform that enables people to visually create simple rules to compose Web sites and data sources. A similar approach can be applied to empower users to create small applications tailored to their needs on top of their smart things. In the next section we discuss the specific requirements of Physical Mashup Editors and describe the architecture of a platform for building these editors.

2.4.2 From Web 2.0 Mashups Editors to Physical Mashup Editors

While Web 2.0 Mashup techniques and tools can be largely re-used for the Web of Things, the physical world has some special constraints that need to be addressed when designing Physical Mashups editors. We deduce these constraints based on a case-study in which we adapted an existing Web 2.0 Mashup Editor to be used as a Physical Mashups editor. Then, for the identified constraints we propose a number of requirements [68].

2.4.3 Adapting a Web 2.0 Mashup Editor to the Web of Things

To better understand the requirements of a Physical Mashups editor, we adapted an existing Web 2.0 Mashup Editor to include building-blocks featuring access to smart things. Our case-study is based on the Clickscript project [149]. Clickscript if a Web

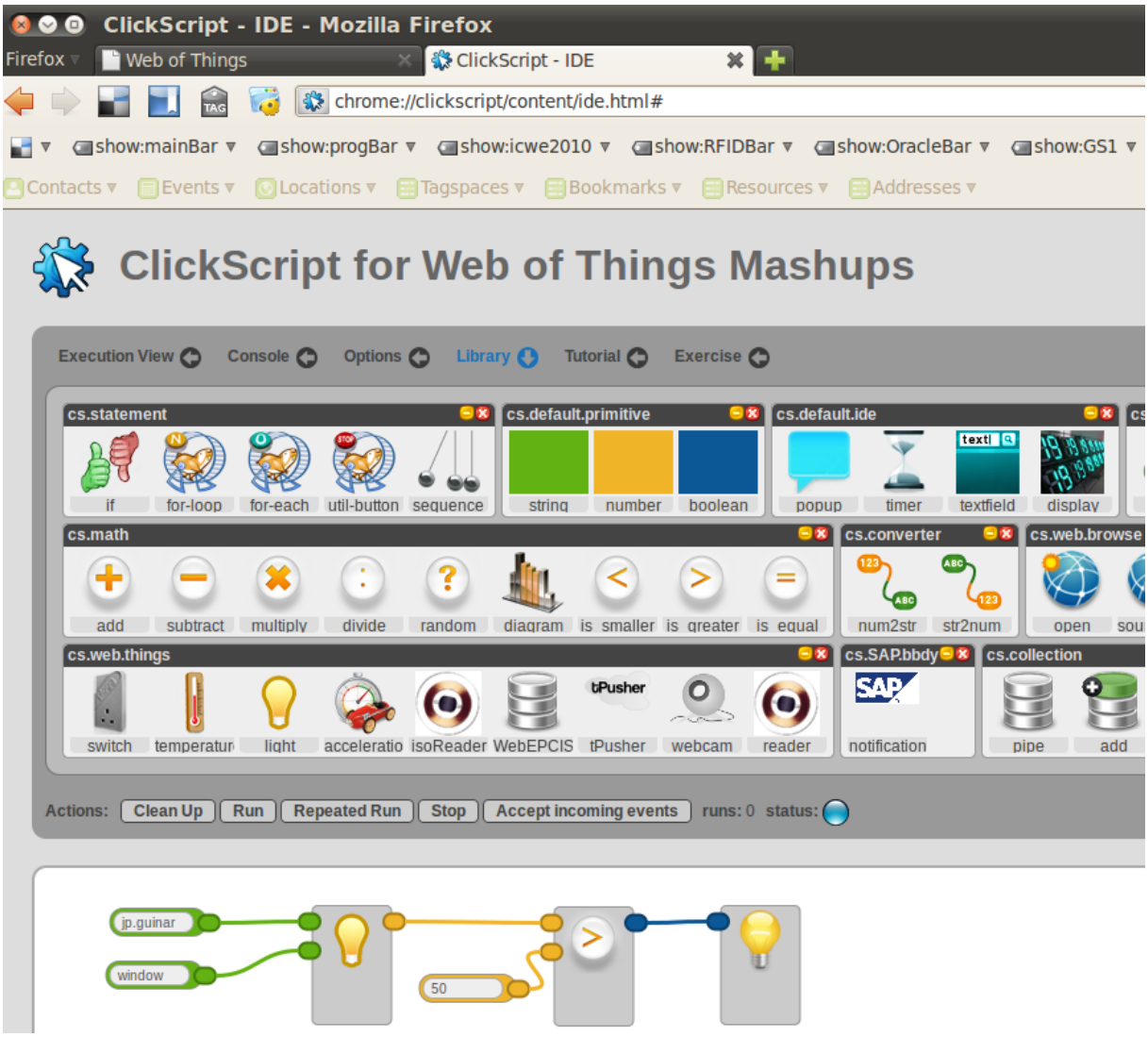


Figure 2.30: A Physical Mashup with a modified version of the Clickscript Mashup editor. The mashup turns a lamp on whenever the light level observed by a real-world sensor is above a threshold.

platform [298] written in JavaScript and HTML on top of two popular JavaScript libraries (Dojo [277] and JQuery [250]). Clickscript allows people to visually create Web 2.0 Mashups by connecting building-blocks of resources (e.g., Web pages, strings, etc.) and operations (e.g., greater than, if/then, loops, etc.).

We decided to use Clickscript for two main reasons: First, since the editor was created only with client-side Web technologies its deployment and extension is very straightforward and can illustrate well the integration of smart things to pure Web scripting languages. Then, Clickscript was created with the aim of teaching young children the basics of programming. As a consequence, its usage is very simple and accessible even to non-technical people [149].

Since Clickscript is written in JavaScript and running in the browser, it cannot use resources based on low-level protocols such as Bluetooth or Zigbee. However, it offers full HTTP support and hence can easily access RESTful services. As WoT devices implemented using the architecture described in the Device Accessibility Layer, Findability Layer and Sharing Layer are fully accessible through a RESTful Web API, it is straightforward to create Clickscript building-blocks supporting smart things.

We used this approach to create Clickscript building-blocks for all the devices we present in the case-studies of this thesis (see Chapter 3 and Chapter 4). The generic JavaScript code required to integrate a smart thing as a ClickScript building-block is shown in Listing 2.10. This concise snippet of code is a template of all that is required to integrate any smart things that implements, at least, the Device Accessibility Layer of the Web of Things Architecture.

The result of this script is a new ClickScript building-block that can be used by end-users to create simple Physical Mashups. As an example, the mashup shown in Chapter 3 (Figure 3.22) gets the light level in a room by `GET`ting the light resource of a sensor. If it is bigger than a given threshold, it turns the light off by sending it a `PUT` request.

```
1  // This creates a temperature sensor building block.
2  csComponentContainer.push({
3      name : "cs.web.things.[Name_of_Smart_Thing]",
4      description : "[Description]",
5      inputs :
6      [
7      {
8          name: "URL", // The Smart Things' resource URL
9          type: "cs.type.String"
10     }, [...] // additional parameters
11     ],
12     outputs:
13     [
14     {
15         name: "[Result provided by the Smart Thing]",
16         type: "cs.type.[Number | String]"
17     }
18     ],
```

```
19      image: "web/things/[Icon]",
20      exec : function(state){
21          this.setAsync();
22          var component = this;
23          $.ajax({
24              // Content negotiation, alternatively a .json
                   could simply
25              // be added to the request URL.
26              beforeSend: function(xhrObj){
27                  xhrObj.setRequestHeader("Accept","application/
                       json");
28              },
29              //Read the URL provided as parameter
30              url: state.inputs.item(0).getValue(),
31              type: ["GET" | "PUT" | "POST" | "DELETE"],
32              success: function(result){
33              // Triggered when the HTTP response arrives
34              // Process the response if required
35              // Write it to the output of the building block
36                state.outputs.item(0).setValue(result.resource.
                     getters[0].value);
37                component.finishAsync();
38              },
39              error: function(msg){
40                  alert("Error on: "+aurl);
41              }
42          });
43      }
44  });
```

Listing 2.10: Generic JavaScript code required to integrate a new smart thing to the Clickscript mashup editor as a Clickscript building-block

This readily illustrates the simplicity of adapting an existing mashup editor to WoT devices thanks to their Web integration. However, it also illustrates the shortcomings of the approach. First, while creating mashups can be done by end-users, creating new building-blocks is still only accessible to the community of Web developers. To prevent this, a smart things discovery mechanism should be implemented in order to automate the creation of the corresponding building-blocks.

Second, for the mashup shown in Figure 3.22, the mashup editor has to constantly pull the temperature from the device which is sub-optimal. Hence the need to support push mechanisms as described in Section 2.1.3. This lack of push support is a common characteristic of client-side mashup editors since an HTTP Callback (Web Hook) approach is not possible in this case. However, we adapted Clickscript to support HTML5 WebSockets.

The original version of Clickscript offers two ways of executing mashups: First, the end-user can manually start the mashup by pressing a `Run` button triggering the execution

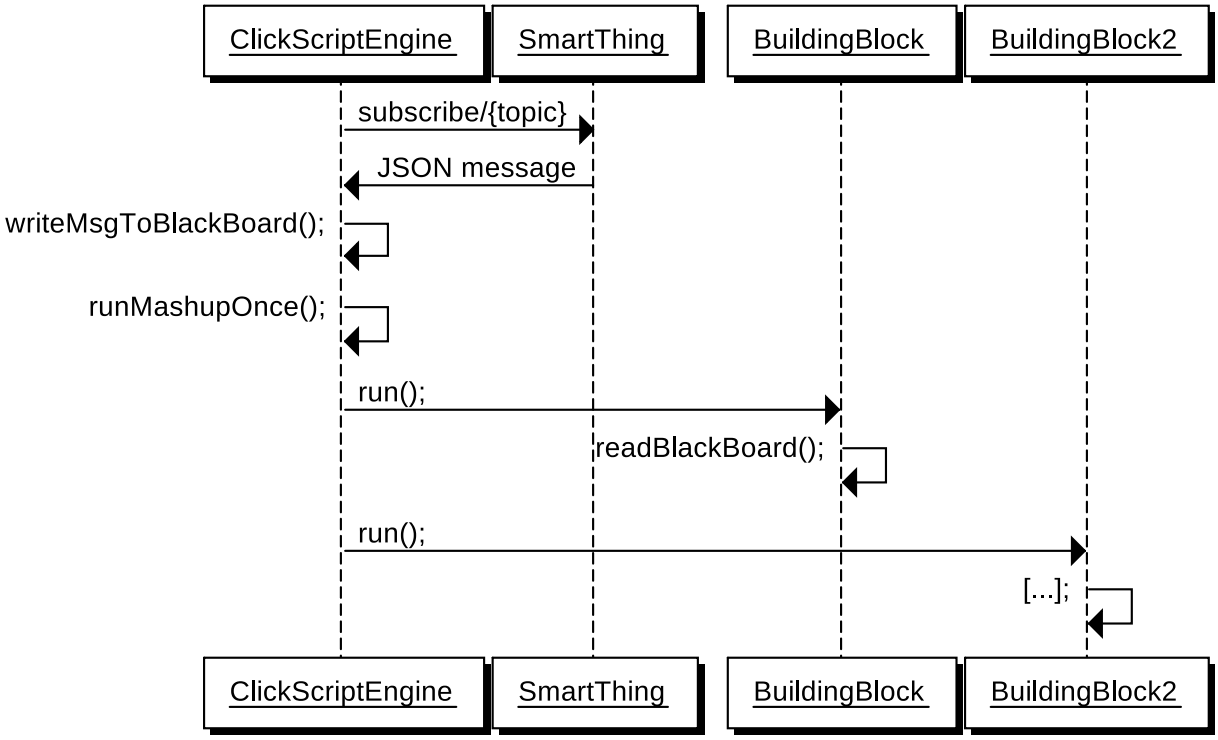


Figure 2.31: Sequence diagram of a typical Web-push triggered execution in the extended version of the Clickscript mashup editor. The editor subscribes to a WebSocket (or Comet) topic and writes the incoming messages to a black-board variable which is then read by building-blocks.

process. Alternatively, he can use a `Repeated-Run` button which runs the mashup in an infinite loop. We added an asynchronous, push-based execution model. The extended execution method using WebSockets is shown in Figure 2.31. When using this option, the user is prompted for the URI of an HTML5 WebSocket enabled server, e.g., an instance of the tPusher service running on a Smart Gateway (see Section 2.1.3). This URI is used to register the Clickscript client to the WebSocket server.

Each incoming HTML5 WebSocket message triggers the execution process. Furthermore, following a black-board approach [216], the payload of each incoming HTML5 message is extracted and written to a variable accessible to all building-blocks. A concrete Physical Mashups prototype based on this Web-push enabled version of Clickscript is described in Chapter 4, Section 4.5.4.

2.4.4 Requirements for Physical Mashup Editors

As a result of the prototype built on top of the Clickscript Web 2.0 Mashup editor, we propose a number of requirements for editors of Physical Mashups [68]:

Support for event-based mashups The current Web, and thus the vast majority of Web 2.0 mashup editors, are based on the concept of clients pulling information from servers [216]. Several studies [41, 191] have shown that while this model matches the requirements for controlling smart things, it is inefficient for real-world

monitoring applications. Hence, the need for Physical Mashups editors to offer core support for event-based interactions, where parts of the workflows of mashups can be triggered based on events pushed from smart things to the editors using Web push mechanisms.

Support for dynamic building-blocks Manually creating building-blocks for each thing does not scale with the heterogeneity of objects in the physical world. Thus, the need for the mashup editors to support partially automated integration through service discovery techniques leveraging the Findability Layer.

Support for non-desktop platforms Web 2.0 mashup editors are for the large part meant to run in Web browsers of desktop computers. However, in the case of the Web of Things, the in situ development of Physical Mashups e.g., on mobile phones or tablets should be fostered as virtual interactions with the physical world can really benefit from occurring beyond the desktop metaphor [22, 194].

Support application specific editors Due to the heterogeneity of use-cases in the Web of Things, a "one-size-fits-all" mashup editor is very unlikely to use adapted metaphors and tools for a particular domain. Hence, rather than creating a mashup editor, the architecture should be a mashup platform exposing an API that can be used to develop specific mashup editors (e.g., a mashup editor for supply chain related or home automation use-cases). This also lets users create their mashups locally, e.g., on a mobile phone, and export them to a more robust framework for execution.

2.4.5 A Platform for Physical Mashups Editors

The goal of the Physical Mashups Framework [4, 68, 113] is to offer a platform that fulfills the requirements discussed before. Rather than providing a generic mashup editor, the Physical Mashups Framework is a mashup engine, i.e., a Web service capable of running mashups [216].

As shown in Figure 2.32, the framework is a composition environment between Web-enabled smart things and virtual services such as messaging or visualization services. It features a RESTful Web API using which actual mashup editors can be built. These editors use the framework for managing the life-cycles of mashups, from the definition of mashups to the discovery of virtual and smart things' services and the actual execution of the mashups.

The Physical Mashups Framework is not a mashup editor itself. Indeed, the idea is for the framework to support the creation of domain-domain specific mashup editors. For instance, in Section 3.4 we build a mobile mashup editor dedicated to create simple applications that optimize the energy consumption of household appliances. The mobile application uses the Physical Mashups Framework RESTful Web API to create and run the mashups in the framework's engine.

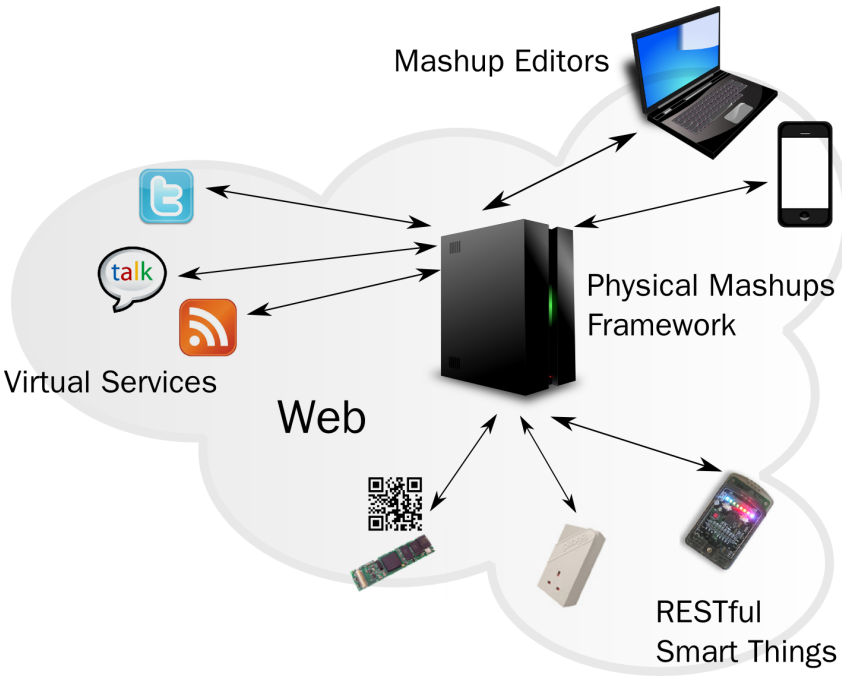


Figure 2.32: Overview of the Physical Mashups Framework. The framework is the mediator between smart things, virtual services and clients (domain-specific mashup editors).

2.4.6 System Architecture

In this section we further describe the functionality of the Physical Mashups Framework by focusing on the most important components of the architecture as shown in Figure 2.33.

Discovery Component

The `DiscoveryComponent` implements the requirement for *supporting dynamic mashup building-blocks*. It is an implementation of the STM Translation Service described in Section 2.2.1 that uses semantic annotations crawled from the smart things HTML representation to generate an internal representation of the Smart Thing Description model.

Mashup editors can then retrieve a serialized version of this description (in the form of a WADL file) that they can use to dynamically generate relevant user interfaces for the building-blocks corresponding to the newly discovered smart thing.

Workflow Engine

Core to the Physical Mashups Framework is a mashup engine. This engine is responsible for the life-cycle of Physical Mashups Framework. It compiles the mashups into a runnable workflow composed of several building-blocks and runs it.

Rather than creating an engine from scratch, the Physical Mashups Framework is based

on the Ruote workflow engine [270]. Ruote is an open-source lightweight workflow engine that is well suited to manage workflows that call several services on the Web, especially when these services are HTTP-based and RESTful.

A Domain Specific Language for Workflows To create workflows, Ruote provides a Domain Specific Language [270] that we reuse in the workflow engine of the Physical Mashups Framework. We briefly describe the most important language construct of the workflow DSL:

Expression A Ruote-based workflow describes a process composed of `Expressions`. Each step in the process is represented as an `Expression`.

Workitem `Expressions` communicate with each other based on a message passing mechanism. The message is initialized at the beginning of the process and modified by each `Expression`. In the Ruote DSL, such a message is called `Workitem`.

Participant are the most important form of `Expressions`, they perform the business logic of the workflow at each step of the process. The engine manages the orchestration among `Participants` by sending and receiving the `Workitems`. Ruote provides a large set of predefined `Participants` but new ones can be added very easily by implementing two methods: `initialize` and `consume`. The former is called whenever a participants is added to a workflow whereas the latter is called when a `Workitem` is applied by the engine to the `Participant`.

Further constructs of the workflow DSL are common language elements such as process-definition constructs, sequences, conditional expressions, loops, subprocesses and listeners. Listing 2.11 presents a simple process (i.e., workflow) definition in XML. Ruote supports such definitions in XML, Ruby or JSON.

```
1 <process-definition name="my_workflow">
2         <sequence>
3                 <participant ref="getTemperatureFromSensor"/>
4                 <participant ref="applyRules"/>
5         </sequence>
6 </process-definition>
```

Listing 2.11: A typical process (workflow) definition using an XML representation of the Ruote Workflow DSL language.

Physical Mashups Building-Blocks Library

To adapt it to Physical Mashups, we extend the Ruote DSL with WoT specific building-blocks. These blocks complement the existing DSL in the form of `Participants` constructs.

The interesting aspect is that rather than specifying them using a programming language, these blocks can be specified through the RESTful API of the Physical Mashups Frame-

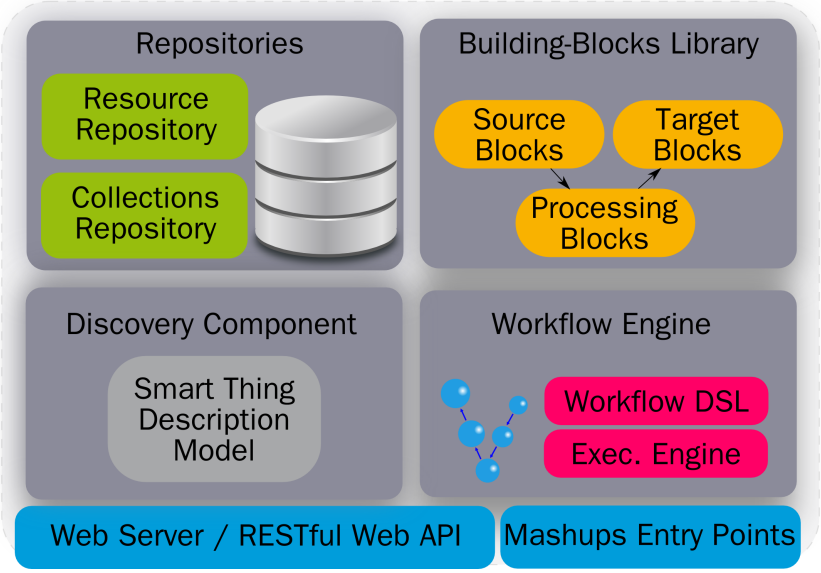


Figure 2.33: Most important components of the Physical Mashups Framework. The Workflow Engine manages the life-cycle of mashups and runs them as workflows. The Discovery Component supports the dynamic integration of new metadata-annotated smart things. The Building-Blocks Library provides the building-blocks for creating Physical Mashups and the Repositories are used to store mashups, building-blocks and data coming from smart things.

work using either XML or JSON representations sent as payloads of HTTP messages. As a consequence, clients (i.e., mashup editors) can build mashups online, block by block using the framework API.

As shown in Figure 2.33, we created three types of building-blocks, *source*, *processing* and *target* blocks. We describe each of these tasks below.

Source Blocks These building-blocks correspond to the inputs of the mashups, i.e., smart things or virtual services on the Web. Four types of source blocks are defined and implemented:

REST Blocks Are the most important building-blocks from a WoT perspective. They encapsulate HTTP interactions with resources. They can be used to add smart things, that were previously discovered by the `DiscoveryComponent`, to the workflows. Similarly, they can be used to interact with RESTful virtual services on the Web. Listing 2.12 shows an example of `RESTBlock` used to retrieve the temperature value of a Web-enabled sensor node.

SOAP Blocks In order to support WS-* services, a SOAP building-block is implemented. Since the workflow engine takes care of the invocation, this component is especially interesting in the case of mashup editors running on resource-constrained devices such as mobile phones as these might not be able to invoke WS-* services directly.

Repository Blocks In more complex Physical Mashups, a solution to persist data is

often required. As an example reacting on the power consumption of a particular device might require to store the measurements over time and perform analysis on aggregated data. For this purpose, a `RepositoryBlock` allows to create persistent collections of data that can be queried later using processing blocks.

Subscription Blocks As we have seen before, smart things might provide their functionality asynchronously, pushing it back to the clients whenever it becomes relevant, hence the need to support *event-based mashups*. The `SubscriptionBlock`s implement this requirement using an HTTP Callback approach (see Section 2.1.3). These blocks can be used to subscribe to events and clients can specify which building-block of the workflow will be the recipient of incoming events. Internally, a `SubscriptionBlock` generates a callback URI containing the name of the recipient block that will be caught by the Mashup Entry Point component and routed to the correct block.

```
1 <process-definition name='x' revision='y'>
2  <sequence>
3   <participant ref='RestBuildingBlock'  name='temperature'
       method='get' site='http://DOMAIN' path='/generic-nodes/1/
       sensors/temperature.json' args='{}' />
4  </sequence>
5 </process-definition>
```

Listing 2.12: XML definition of a REST building-block retrieving the temperature value from a WoT sensor node.

Processing Blocks As shown in Figure 2.33, Processing Blocks represent the logic (e.g., mathematical operations, filtering, querying, etc.) between inputs and outputs. As defined by Yee [215] a common operation in mashups is to translate the data into a form meaningful to the destination service. Part of the translation process is the extraction of the relevant data. `QueryBlock`s offer a simple language construct to perform the most frequent data extraction operations.

The basic input of a `QueryBlock` is a data source or a set of data sources. Data sources can be either XML or JSON representations specified by URIs pointing either to actual resources on the Web or to internal resources stored using a `RepositoryBlock`.

A `QueryBlock` can be configured by means of several parameters:

- `Select` allows to configure which attributes should be extracted as outputs of the query. This parameter is functionally similar to a SQL `select`.
- `First` allows limiting the number of returned attributes (e.g., the first 10 attributes).
- `Filter` allows to do some simple filtering on the data. It supports logical operators, comparison operators as well as simple regular expressions.
- `Sort` is similar to the SQL `order by` operator.

- `GroupBy` allows to get a summarized view of the queried data. Similarly to the SQL `group by` statement it aggregates the data. The supported aggregation functions are max, min, sum, count and average.

Listing 2.13 provides an example of the definition of a `QueryBlock`. The output of the block is a JSON document that contains the values of the last trace recorded by a GPS sensor (e.g., a mobile phone).

```
<process-definition name="getLastLocation">
 <sequence>
  <participant ref="json_query" name="LastLocation"
   user_collection="locationsFromMobile"
   user="rachel" source="notification"
   first="1" sort="notification.time␣desc"/>
 </sequence>
...
</process-definition>
```

Listing 2.13: XML definition of a Query building-block. This block extracts, from a repository, the latest location sent by a GPS sensor.

Target Blocks As shown in Figure 2.33, `TargetBlock`s are the outputs of compositions. They represent services that can be actuated as a result of processing the input data. Four types of `TargetBlock`s are supported by the framework:

AtomPub Blocks These components supports publishing JSON, XML or HTML data to any server complying with the AtomPub protocol described in Section 2.1.3.

Twitter Blocks These components encapsulate the Twitter API, letting data coming from smart things being pushed to Twitter.

XMPP Blocks These components support the XMPP messaging protocol used in several Web messaging clients such as Google Chat and sometimes used to provide real-time data from smart things [96, 95].

Visualization Blocks These blocks offer a set of visualization methods such as graphs that are implemented using the Google Visualization API [238].

Additionally, the `RESTBlock`s and `RepositoryBlock`s can be used as `TargetBlock`s as well. `RESTBlock`s enable, for instance, smart things to smart things communication and actuation through the Physical Mashups Framework. `RepositoryBlock`s, allows to use the framework as a database for smart things, where the real-world data can be preprocessed using `QueryBlock`s before storage.

Repositories

The Physical Mashups Framework offers two repositories. The Collection Repository is used by `RepositoryBlock`s to store JSON or XML data. The Resource Repository

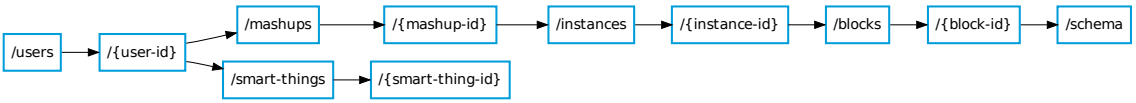


Figure 2.34: Tree-representation of the RESTful API of the Physical Mashups Framework.

mainly provides a persistent storage for mashups as well as extracted instances of the Smart Things Description model. Through this repository parts of or complete workflows can be referenced and used in other mashups, allowing for a reuse and sharing mechanism.

RESTful API

As with every component of the Web of Things Architecture, the Physical Mashups Framework is based on a RESTful architecture and features a RESTful API as shown in Figure 2.34. Mashup editors and other clients can use it to build the mashups using an XML or JSON representation of the building-blocks. Furthermore, the API allows client applications to retrieve existing mashups and manage running instances.

The API is centered around the notion of a `user` where mashup definitions, discovered smart things and collections are associated with a specific user of the framework but can be shared amongst users.

2.4.7 Discussion and Summary

In this chapter we introduced the idea of Physical Mashups. We defined three categories of development approaches for Physical Mashups: First, *Manual Mashup Development* which helps developers building upon smart things by streamlining the development process to simple Web development. In Section 3.2.2 we demonstrate how this development approach made it easy to realize the *Energie Visible* mashup for energy-awareness. Similarly, in Section 4.3.4 we illustrate how it made possible to create the EPCFind prototype that leverages real-world RFID data to track and trace someone's belongings. Furthermore, with the *Ambient Meter* prototype presented in Section 3.4.1, we illustrate how the approach can also help smart things to smart things communication.

Then, with the *Widget Based Mashup Development* the actual communication with smart things is transparent to the developer who simply has to use the incoming data (in a blackboard approach) to create new applications encapsulated in JavaScript and HTML Widgets. We will demonstrate this development approach with the *EPC Dashboard Mashup* presented in Section 4.5.3.

Finally, with the *End-User Development with Mashup Editors* we explain how visual metaphors and simple editors can be used to enable end-users to create simple compositions.

We illustrate this a straightforward adaptation of an existing mashup editor to WoT devices. We further introduce the Physical Mashups Framework architecture that allows to build domain and device specific Physical Mashups editors and run the created mashups in the cloud. In Section 3.4.3 we propose a mashup editor that can be used to create simple applications on top of Web-enabled home appliances in order to make homes more energy aware and efficient.

2.5 Developers Perspectives on the WS-* Alternative Architecture

The application architecture proposed in this chapter is definitely not the only way to create a uniform integration layer for smart things and we will discuss several alternatives in the related work section (see Section 2.7).

However, in this section we consider WS-* services as an important alternative[3]. Out of the possible alternatives, WS-* services stand out for two reasons: First their current market penetration, especially in the field of enterprise software business is significant [301]. Second, rather than proposing a simple network layer integration, the WS-* galaxy of standards forms an ecosystem that addresses, in a standard way, several layers of a Web of Things Architecture.

WS-* services declare their functionality and interfaces in a Web Services Description Language (WSDL) file. Client requests and service responses objects are encapsulated using the Simple Object Access Protocol (SOAP) and transmitted over the network, usually using the HTTP protocol. Further WS-* standards define concepts such as addressing, security, discovery or service composition. Although WS-* was initially created to achieve interoperability of enterprise applications, work has been done to adapt it to the needs of resource-constrained devices [92, 157, 214, 183]. Furthermore, lighter forms of WS-* services, such as the Devices Profile for Web Services (DPWS) [286], were proposed [103].

While they share similar goals, REST and WS-* are not compatible; they tackle loose coupling and interoperability differently. Consequently, work has been done to evaluate these two approaches. In [153, 154], REST and WS-* are compared in terms of reusability and loose coupling for business applications. The authors suggest that WS-* services should be preferred for "professional enterprise application integration scenarios" and RESTful services for tactical, ad-hoc integration over the Web.

Internet of Things applications pose novel requirements and challenges as neither WS-* nor RESTful Web services were primarily designed to run and be used on smart things, but rather on business or Web servers. This development thus necessitates assessing the suitability of the two approaches for devices with limited capabilities. Yazar et al. [214]

[3] An introduction to WS-* services is outside the scope of this thesis and we invited the reader to refer to books such as [10] or [49] for an exhaustive overview of the underlying technologies.

analyze the performance of WS-* and RESTful applications when deployed on wireless sensor nodes with limited resources and conclude that REST performs better.

However, evaluating the performance of a system when deployed on a particular platform is not enough to make the architectural decision that will foster adoption and third-party (public) innovation. Indeed, studies like the Technology Acceptance Model [32] and more recent extensions [63, 132] show that the perceived ease of use of an IT system is key to its adoption. As many manufacturers of smart things are moving from providing devices with a few applications to building devices as platforms with APIs, they increasingly rely on external communities of developers to build innovative services for their hardware (e.g., the Apple App Store or Android Marketplace). An easy to learn API is, therefore, key in fostering a broad community of developers for smart things. Hence, choosing the service architecture that provides the best developer experience is instrumental to the success of the Internet of Things and the Web of Things on a larger scale.

In this section we complement the decision framework that can be used when picking the right architecture for IoT applications and platforms. We supplement previous work [153, 154, 214] by evaluating, in a structured way, the actual *developers' experience* when using each architecture in an IoT context. We analyze the perceived ease of use and suitability of WS-* and RESTful Web service architectures for IoT applications. We base our study on the qualitative feedback and quantitative results from 69 computer science students who developed two applications that accesses temperature and light measurements from wireless sensor nodes. For the one of the applications, the participants used a sensor node implementing the Device Accessibility Layer and Findability Layer of the Web of Things Architecture, thus offering a semantically annotated RESTful Web API. In the second case, they were accessing a WS-* (WSDL + SOAP based) sensor node.

Our results show that participants almost unanimously found RESTful Web services easier to learn, more intuitive and more suitable for programming IoT applications than WS-*. The main advantages of REST as reported by the participants are, intuitiveness, flexibility, and the fact that it is lightweight. WS-* is perceived to support more advanced security requirements and benefits from a clearer standardization process.

The following sections are structured as follows. Section 2.5.1 describes the study methodology. Section 2.5.2 presents and analyses the results. Finally, Section 2.6 discusses the implications of our findings and devises guidelines. The results of the presented study were published in [73].

2.5.1 Methodology

Our study is based on a programming exercise and the feedback received from a group of 69 computer science students who had to learn RESTful and WS-* Web service architecture and implement, in teams, mobile phone applications that accessed sensor data from different sensor nodes using both approaches as shown in Figure 2.35. The exercise

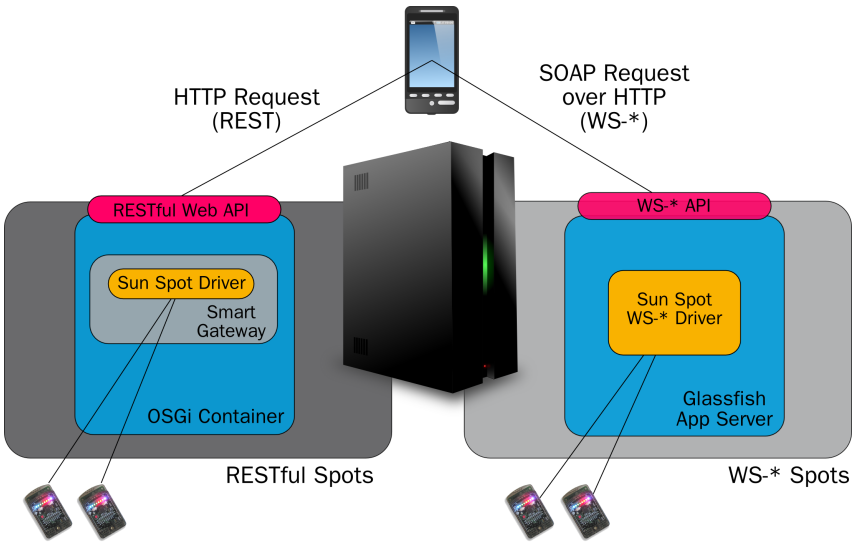


Figure 2.35: Setup of the user study. Two Sun SPOTs sensor nodes are connected to a Smart Gateway through a sync-based Device Driver (left). Their functionality is exposed through a RESTful Web API. Two other Sun SPOTs are accessible through a WS-*, SOAP + WSDL interface deployed in an application server (right).

was part of an assignment in the Distributed Systems course at ETH Zurich[4].

To get and parse the RESTful sensor response, participants used the HTTP and JSON libraries, which are already available on Android phones. To perform the WS-* request, the external *kSoap2* library [252] was used. Averaging over the submissions, the programs had 105 lines of code for the WS-* implementation ($SD = 50.19$, where SD is the Standard Deviation), opposed to 98 lines of code for the REST implementation ($SD = 48.31$).

The two coding tasks were solved in teams of two or three members who were able to freely decide how to split up the tasks amongst team members, which models industry practice. The coding tasks were successfully completed by all teams within two weeks. To ensure that every team member had understood the solution of each task, individual reports describing the design and architecture choices had to be submitted. Additionally, every student had to submit a structured questionnaire about both technologies and a voluntary feedback form on the learning process. Students were informed that answers in the feedback form were not part of the assignment, and that responses would be used in a research study. To encourage them to give honest answers about the amount of effort invested in solving both coding tasks, perception, and attitudes towards both technologies, entries in the feedback form were made anonymously. Table 2.4 summarizes the data collection sources.

Demographics: The participants were from ETH Zurich, in their third or fourth year of Bachelor studies. Teams were formed of two or three members. They were taught both technologies for the first time, in a short introduction during the tutorial class. 89% reported not having had any previous knowledge of WS-* and 62% none of REST. From the 35% that had already used REST before the course, half reported that they had

[4]The assignment is available online: `guinard.org/phd/study-assignment.pdf`

previously been unaware that the technology they used actually was based on the REST architecture. 5% (2 students) had programmed WS-* applications.

Additional Tasks: Subsequent tasks involved creating visualization mechanisms for the retrieved data, both locally and through cloud visualization solutions.

Data Source	**Type**	**N**
RESTful and WS-* Applications	Team	25
Structured Questionnaire	Individual	69
Feedback Questionnaire	Anonymous	37

Table 2.4: Data was collected from different programming tasks and questionnaires.

2.5.2 Results

Here, we present our results on the perceived differences, learning curve, and suitability of the technologies for smart things related use-cases.

Perceived Differences

Using the structured questionnaire, we collected qualitative data on the perceived advantages of both technologies with respect to each other (see Table 2.5). While REST was perceived to be "*very easy to understand, learn, and implement,*" "*lightweight and scalable*", WS-* "*allows for more complex operations,*" provides higher security, and a better level of abstraction.

REST ($N = 69$)	#
Easy to understand, learn, and implement	36
Lightweight	27
Easy to use for clients	25
More scalable	21
No libraries required	17
Accessible in browser and bookmarkable	14
Reuses HTTP functionality (e.g., caching)	10
WS-* ($N = 69$)	#
WSDL allows to publish a WS-* interface	31
Allows for more complex operations	24
Offers better security	19
Provides higher level of abstraction	11
Has more features	10

Table 2.5: Participants felt that WS-* provides better abstraction, but REST is easy to learn and use.

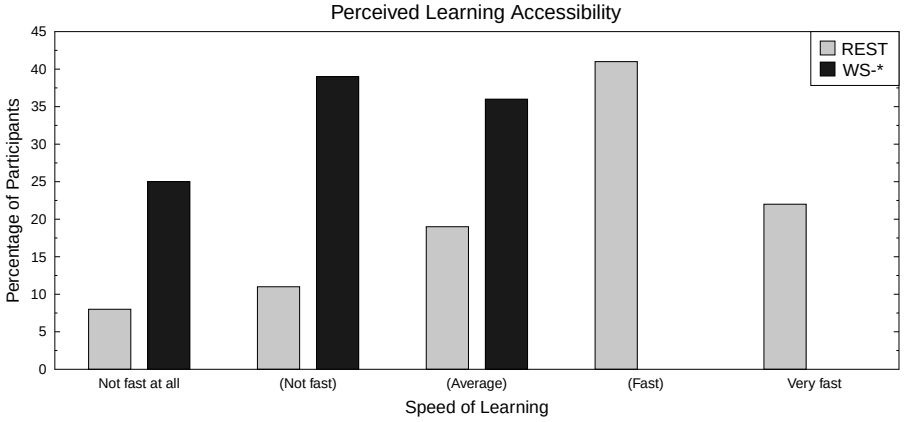


Figure 2.36: A majority of participants reported that REST as *fast* or *very fast* to learn and WS-* as *not fast* or *average*.

Learning Curve

In the feedback form, we asked participants to rate on a 5 point Likert scale how easy and how fast it was to learn each technology (1=not easy at all, ..., 5=very easy). As shown in Figure 2.37, 70% rated REST "easy" or "very easy" to learn. WS-* services, on the other hand, were perceived to be more complex: only 11% respondents rated them easy to learn. Event if all participants were required to learn and explain the concepts of both technologies, compare their advantages, analyze their suitability and explain design decisions, we restricted the sample to participants who reported to have worked on programming both REST and WS-* assignments within their teams (N=19) to avoid bias. We then applied a paired two sample t-test to compare the learning curve for REST and WS-*. Our results show that REST (with an average $M = 3.85$ and a Standard Deviation $SD = 1.09$) was reported to be significantly easier to learn than WS-* ($M = 2.50, SD = 1.10, t(19) = 4.23, p < 0.001$. Similarly, REST ($M = 3.43, SD = 1.09$) was perceived to be significantly faster to learn than WS-* ($M = 2.21, SD = 0.80, t(13) = -3.46, p = 0.002$).

Furthermore, in the feedback form, we collected qualitative data on the challenges of learning both technologies, asking the participants: "What were the challenges you encountered in learning each of the technologies? Which one was faster and easier to learn?". Nine participants explained that REST was easier and faster to learn because RESTful Web services are based on technologies, such as HTTP and HTML, which are well-known to most tech-savvy people: "*Everybody who is using a browser already knows a little about [REST].*"

WS-* was perceived to be overly complicated: "*REST is easy and WS-* is just a complicated mess.*" Reasons for such strong statements were the complexity of extracting useful information out of the WSDL and SOAP files (mentioned by 8), as well as the number and poor documentation of parameters for a SOAP call. The lack of clear documentation was perceived as a problem for REST as well: Seven participants said that further request

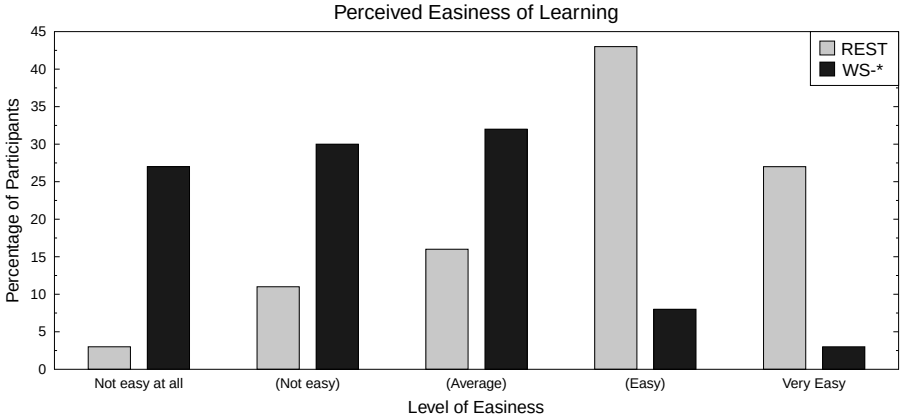


Figure 2.37: Participants reported REST as easier to learn than WS-*.

examples, alongside with the traditional documentation (e.g., Javadoc) for both REST and WS-*, would be very useful. Eight participants explicitly mentioned that they had had previous experience with REST during their spare time. This illustrates the accessibility and appeal of RESTful Web services, and it positions them as an ideal candidate for smart things APIs in terms of lowering the entry barrier for creating applications. In the feedback form, 25 participants said that REST made it easier to understand what services the sensor nodes offered. Eight participants explained this by the fact that, for REST, an HTML interface was provided. This emphasizes that RESTful smart things should offer an HTML representation by default. Seven participants found WS-* easier for this matter. They noted that a WSDL file was not necessarily easy to read, but they liked the fact that it was "*standard*".

Suitability for Use-Cases

In the feedback form, we asked participants to rate on a Likert scale (1=WS-*, ..., 5=REST) which one of the two technologies they would recommend in specific scenarios. REST was considered more suitable than WS-* for IoT applications running on embedded devices and mobile phones (see Figure 2.38). The one sample t-test confirmed that the sample average was statistically higher than the neutral 3, and therefore inclined towards REST. This was the case both for embedded devices ($M = 3.86, SD = 1.03, t(36) = 5.10, p < 0.001$), and for mobile phone applications ($M = 3.51, SD = 1.12, t(36) = 2.79, p = 0.004$). For business applications, however, a higher preference for WS-* was stated but the preference was not emphasized enough to be statistically significant ($M = 2.67, SD = 1.33, t(36) = -1.48, p = 0.07$).

General Use-Cases We asked our participants to discuss the general use-cases for which each technology appeared suitable. When asked: "For what kind of applications is REST suitable?", 23 people mentioned that REST was well adapted for simple applications offer-

ing limited and atomic functionality: "*for applications where you only need create/read-/update and delete [operations]*". 8 participants also advised the use of REST when security is not a core requirement of the application: "*Applications where no higher security level than the one of HTTP[s] is needed*". This is supported by the fact that the WS-* security specification offers more levels of security than the use of HTTPS and SSL in REST [160, 116]. 6 participants suggested that REST was more adapted for user-targeted applications: "*[...] for applications that present received content directly to the user*". Along these lines, 14 users said that REST was more adapted for Web applications or applications requiring to integrate Web content: "*[for] Web Mashups, REST services compose easily*".

We then asked: "For what kind of applications is WS-* more suitable?". Twenty participants mentioned that WS-* was more adapted for secure applications: "*applications that require extended security features, where SSL is not enough*". 16 participants suggested to use WS-* when strong contracts on the message formats were required, often referring to the use of WSDL files: "*with WS-* [...] specifications can easily be written in a standard machine-readable format (WSDL, XSD)*".

REST (N=37)	#
For simple applications, with atomic functionality	23
For Web applications and Mashups	14
If security is not a core requirement	8
For user-centered applications	6
For heterogeneous environments	6
WS-* (N=37)	#
For secure applications	20
When contracts on message formats are needed	16

Table 2.6: REST was perceived to be more suited for simple applications, and WS-* for applications where security is important.

For smart things Both WS-* and RESTful Web Services were not primarily designed to run on embedded devices and mobile phones but rather on business or Web servers. Thus, assessing the suitability of the two approaches for devices with limited capabilities is relevant.

As shown in the first part of Figure 2.38, for providing services on embedded devices, 66% of the participants suggested that REST was either "adapted" to "very-adapted". When asked to elaborate on their answers, 6 participants suggested that for heterogeneous environments REST was more suitable: "*for simple application, running on several clients (PC, iPhone, Android) [...]*". 7 participants said that REST was adapted for embedded and mobile devices because it was more lightweight and better suited for such devices in general: "*the mapping of a sensor network to the REST verbs is natural [...]*". To confirm this, we investigated the size of the application packages for both approaches. The average footprint of the REST application was 17.46 kB while the WS-* application

had a size of 83.27 kB on average. The difference here is mainly due to the necessity to include the *kSoap2* library with every WS-* application. These results confirm earlier performance and footprint evaluations [39, 214, 81].

Smart Home Applications We then went into more specific use-cases, asking: "*Imagine you want to deploy a sensor network in your home. Which technology would you use and why?*". Sixty-two respondents recommended REST to deploy a sensor network in the home, 5 recommended WS-*, and 2 were undecided. Twenty-four participants justified the choice of REST by invoking its simplicity both in terms of use and development: "*REST [...] requires less effort to be set up*", "*Easier to use REST, especially in connection with a Web interface*". Eight participants said that REST is more lightweight, which is important in a home environment populated by heterogeneous home appliances. Interestingly enough, 14 participants mentioned that in home environments there are very little concerns about security and thus, the advanced security features of WS-* were not required: "*I would not care if my neighbor can read these values*", "*The information isn't really sensitive*".

For Mobile Phones Since the mobile phone is a key interaction device for creating IoT applications, we asked the participants to assess the suitability of each platform for creating mobile phone clients to smart things. As shown in the second part of Figure 2.38, 53% of the participants would use REST, 16% would use WS-* and 32% were undecided. They explained these contrasted results by the fact that mobile phones are getting very powerful. 7 participants explained that the amount of data to be processed was smaller with REST which was perceived as an important fact for mobile clients. Interestingly enough, some participants considered the customers of mobile platforms to have different requirements: "*I would use REST, since customers prefer speed and fun over security for smaller devices*". The lack of native WS-* support on Android (which natively supports HTTP) and the required use of external libraries was also mentioned as a decision factor as REST calls can be simply implemented only using the native HTTP libraries.

For Business Applications The results are much more inclined towards WS-* when considering "business" applications. As shown in the third part of Figure 2.38, the majority of our participants (52%) would choose WS-* and 24% REST for servicing business applications. Twenty-one (out of 69, see Table 2.4) justify their decision by the security needs of enterprise applications: "*I would rely on the more secure WS-* technology*". Eighteen participants talk about the better described service contracts when using WS-*: "*I propose WS-* because we could use WSDL and XSD to agree on a well-specified interface early on [...]*". Amongst the participants suggesting the use of REST, 10 justify their decision for its simplicity and 10 for its better scalability.

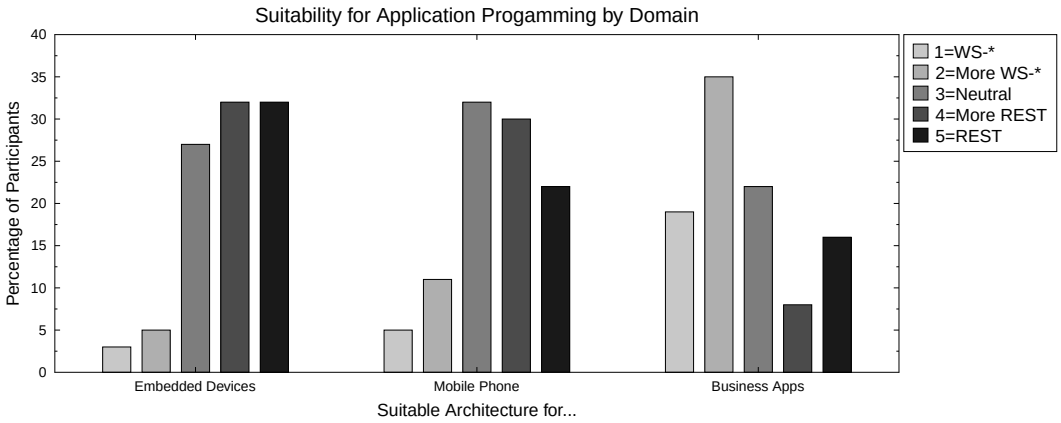


Figure 2.38: Participants reported that REST is better suited for Internet of Things applications, involving mobile and embedded devices and WS-* fit better to the constraints of business applications.

2.6 Discussion and Summary

A central concern in the Internet of Things and thus in the Web of Things is the interoperability between smart objects and existing standards and applications. Two service-oriented approaches are currently at the center of research and development: REST and WS-*. Decisions on which approach to adopt have important consequences for an IoT system and should be made carefully. In this section, we complement existing studies on performance metrics with an evaluation of the developers' preferences, and IoT programming experiences with REST and WS-*. In the context of the presented study, the results show that REST stands out as the favorite service architecture for IoT applications. Future studies should conduct a long-term assessment of the developers' experience, beyond the initial phase of getting started with the technologies. Furthermore, work could be done to compare the experience of advanced, not just novice developers, possibly within industry projects. However, the more experienced the developers are, the more they are likely to develop a bias towards one or the other technology. We summarize the decision criteria used by developers in our study and devise guidelines in Table 2.7.

Requirement	**REST**	**WS-***	**Justification**
Mobile & Embedded	+	-	Lightweight, IP/HTTP support
Ease of use	++	-	Easy to learn
Foster third-party adoption	++	-	Easy to prototype
Scalability	++	+	Web mechanisms
Web integration	+++	+	Web is RESTful
Business	+	++	QoS & security
Service contracts	+	++	WSDL
Adv. security	-	+++	WS-Security

Table 2.7: Guidelines for choosing a service architecture for IoT platforms.

While our results confirm several other research projects that take a more performance-centric approach [39, 214, 81], they contradict several industry trends. In home and industrial automation, standards such as UPnP, DLNA or DPWS expose their services using WS-* standards. One of the reasons for this, also noted by participants, is the lack of formal service contracts (such as WSDL and SOAP) for RESTful services. This is an arguable point as Web experts [160] already illustrated how a well-designed RESTful interface combined with HTTP content negotiation results in a service contract similar to what WS-* offers [160, 116], with no overhead and more adapted to services of smart things [81]. Yet, this illustrates an important weakness of RESTful Web services identified by participants: RESTful Web services are a relatively *fuzzy* concept. Even if the basics of REST are very simple, the lack of a clear stakeholder managing "standard" RESTful architectures is subject to many (wrong) interpretations of the concept. Until this is improved, resources such as [50, 160] profile themselves as de-facto standards.

In cases with strong security requirements, WS-* has a competitive advantage [160, 116]. The WS-Security standard offers a greater number of options than HTTPS (TLS/SSL) such as encryption and authentication beyond the communication channel, endpoint-to-endpoint. In theory, these could also be implemented for RESTful Web services. However, the lack of standard Web support of these techniques would result in tight coupling between the secured things and their clients. Nevertheless, in the context of smart things, it is also important to realize that WS-Security standards are much more resource intensive than those of HTTPS and, thus, rarely fit resource-constrained devices.

Furthermore it is important to consider *how accessible smart things should be*. Participants identified that RESTful Web services represent the most straightforward and simple way of achieving a global network of smart things because RESTful Web services seamlessly integrate with the Web. This goes along the lines of recent developments, such as 6LoWPAN [148] and the IPSO alliance [244], CoRE [242] and CoAP [176], or the Web of Things Architecture presented in this thesis, where smart things are increasingly becoming part and leveraging the infrastructure of the Internet and the Web.

Lastly, it is worth noting that while the decision of adopting a WS-* or RESTful architecture for a smart things platform is important, bridges between the two architectures can be created with some efforts [43, 75, 108], for instance through the use of application layer gateways or Smart Gateways drivers (see Section 2.1.2) as we suggested in [77].

2.7 Related Work

In this section we discuss previous research related to the Web of Things Architecture. It is structured according to the layers of Web of Things Architecture beginning with work related to creating a global network of smart things (Device Accessibility Layer) and ending with ways of composing the services of smart things in an easy and accessible manner (Composition Layer).

2.7.1 Device Accessibility Layer

Linking the Web and physical objects is an attractive idea that has already been in the mind of researchers for years. Early approaches started by attaching physical tokens (such as barcodes) to objects to direct the user to pages on the Web containing information about the objects [202]. These pages were first served by static Web Servers on mainframes, then by early gateway system that enabled low-power devices to be part of wider networks [174]. The Cooltown project pioneered this area of the physical Web by associating pages and URIs to people, places and things [110] and implementing scenarios where this information could by physically discovered by scanning infrared tags in the environment. Similarly, the SPREAD physical/spatial computing model [30] consisted in spreading, with the help of wireless technologies, information that could then be retrieved by mobile users, pointing them to contextually related Web resources. The key idea of these works was to provide a virtual counterpart of the physical objects on the Web. URIs to Web pages were scanned by users e.g., using mobile devices and directed them to online representations of real things (e.g., containing status of appliances on HTML pages or user manuals).

A number of projects proposed solutions to expose the functionality of smart things through APIs in order to build applications upon real-world devices. Among them, JINI, UPnP, DNLA, etc. The advent of WS-* Web Services (SOAP, WSDL, etc.) led to a number of works towards deploying them on embedded devices and sensor networks [157, 36, 129]. DPWS is a subset of the WS-* standards that allows minimal interaction with web services running on embedded devices. DPWS specifies a protocol for seamless interaction with the services offered by different embedded devices. DPWS is aligned (and for the most part compatible) with WS-* technologies. The various specifications DPWS include support for messaging, service discovery, service description, and eventing for resource-constrained devices.

We joined this research and proposed a middleware that bridges the gap between DPWS embedded-devices and enterprise applications, taking care of cross-cutting concerns such as search, service composition, dynamic provisioning of services and data storage [183, 36]. However, in Section 2.5 we discussed the shortcomings of the approach in the case of the WoT.

Several systems (or middleware) for integration of smart things with the Web have been proposed such as SenseWeb [67, 127], the Global Sensor Network (GSN) [1], Pachube [266], ThingSpeak [243], Sen.se [233], ThingWorx [274] or Sensor.Network [84]. These offer platforms for people to share their sensory readings using Web services to transmit data onto a central server and thus cover several layers of the Web of Things Architecture. However, these approaches are focused on building central repositories on top of which services can be built.

The idea to push the Web and the Internet as close as possible to smart things was explored early on in the Internet0 project [65]. Gershenfeld and Cohen understood early on the importance of having a single protocol for smart things and proposed adapting

Internet protocols to embedded devices.

With advances in computing technology, tiny Web servers can be embedded in most devices [40, 99, 41] and thus not only Internet but also Web technologies can be pushed to smart things. Hence, in the Device Accessibility Layer, we propose using the Web and its technologies as the integration backbone of smart things. We build upon works towards the Web integration of wireless sensor networks such as [126, 39, 37] and extend them by systematically applying the constrains of RESTful architectures to smart things and discussing the concept of Smart Gateways to integrate non-IP enabled devices [78]. Furthermore, we complement this work by proposing the three other layers that streamline the development of applications using smart things to what can be done today with Web 2.0 mashups.

2.7.2 Findability Layer

Search has become such a central commodity that it is hard to imagine the Web without search engines. The findability of information [144], people and places has become a central concern of most information architectures [145]. However, while search in the Web of documents is already significantly advanced, searching the real-world remains one of the biggest open challenges for the Web of Things to materialize.

When considering the Web of Things beyond micro use-cases such as home or factory automation, focusing on macro use-cases where billions of things are available and connected to the Web, then discovery simply by browsing HTML pages with hyperlinks becomes difficult. Searching for things is significantly more complicated than searching for documents [166], as things are tightly bound to contextual information such as location, are often moving from one context to another and their HTML representations are less keywords-rich than traditional Web pages.

As a consequence, several researchers have been looking at specific ways of describing smart things and domain-specific standards have been proposed: SensorML [18] is a standard XML language that can be used to describe sensor network applications and devices. Similarly, the Extended Environments Markup Language (EEML) [232] is a language for describing sensor data in digitally enhanced environments.

Not specific to Wireless Sensor Networks and thus closer to the concerns of this thesis, DPWS [103] proposes a *device metadata language* [227] which offers a semantic description of what a real-world device is, by what company it was produced and what it has to offer in terms of functionality. DPWS metadata is often embedded in a WSDL (Webservice Description Language) file which in turns contains the interface description or API of the offered WS-* Webservices. Based on DPWS we proposed the SOCRADES Lookup and Search Infrastructure [80].

The metadata offered by these languages is the basis of our proposed STM model. However, these formats are not well supported and understood on the Web and for some parts

overlap what HTTP already has to offer. As a consequence, they do not leverage the search infrastructure already in place on the Web (e.g., existing search engines).

Closer to Web languages, researchers have been proposing the use of domain-specific ontologies to support semi-automated mashups of smart things [199, 109]. Vermeulen et al. used this approach to implement mashups of tagged physical objects with services on the Web [198]. While the expressive power of these approaches is significant so is their complexity. Our goal is to enable users to search the real-world and machines to understand the basics of smart things (e.g., in order to generate building-blocks of mashups). As a consequence we would like to consider descriptions that can be implemented in a lightweight manner with a constrained but sufficient descriptive power so that they can be understood by a vast number of existing infrastructure services.

SA-REST [119] proposes a lightweight alternative language that can be used to describe RESTful services sharing similarities to what the WSDL language can describe for WS-* services. This is also the proposal of the WADL (Web Applications Description Language) [86] or of some extensions in the WSDL 2.0 language. However, SA-REST goes beyond these languages as it proposes to also support the composition of these services in semi-automatic mashups [177]. Unfortunately, languages like SA-REST are not widely understood on the Web, for instance they are not processed by search engines. The importance of relying on mechanisms that can be well-understood by search engines for a realistic deployment of a service lookup service was discussed by Song et al. [181].

Kopecky et al. propose to make descriptions of RESTful Web services directly embedded in HTML representations [111]. They propose the hRESTs microformat. Note that it is sometimes criticized by the Web community because most of the data the hRESTs microformat encapsulate can also be retrieved by crawling Web architectures that respect the REST constraints (and in particular the *connecteness* constraint). However, when deploying services directly on smart things (without Smart Gateways) this format can be interesting as it avoids the great number of HTTP calls required by crawlers. Researchers such as Alarcon et al. discuss ways and languages to gather more valuable information when crawling RESTful services [7].

Our approach is to propose the STM model inspired by these description languages but we focus on a small set of properties that are sufficient to allow the findability of smart things and their integration into tools such as Physical Mashups editors. We further implement the model using a set of standard microformats as these are well supported and broadly understood by search engines. Yet, relying on existing search engines does not fully leverage smart things and does not really take into account their specific requirements such as mobility or their strong ties with physical locations. As a consequence a body of research is related to real-time search engines for the Internet and the Web of Things.

As explained in the previous section, several platforms such as SenseWeb [67, 127], Pachube or Sensor.Network [84] were proposed. Building on top of Pachube, Kamilaris et al. propose "bridging the Mobile Web and the WoT in Urban Environments" [107] and offer a mobile search engine using location information to retrieve nearby data services.

Similarly, we explored using the concept of proximity to dynamically search for services offered by smart things [64]. However, these approaches are based on a centralized data lookup infrastructure and do not fully leverage the distributed nature of smart things.

In the world of business services, a crucial challenge for SOA developers and process designers is to find adequate services for solving a particular task [31]. Discovering enterprise services often implies manually querying a number of registries, such as Universal Description and Discovery and Integration (UDDI) registries, and the results depend largely on the quality of the data within that registry. While such an approach is adequate for a rarely changing set of large-scale services, the same is insufficient for the requirements of the service offered by smart things. Registering a service with one or more UDDIs is rather complex (which is also why UDDIs are rarely used in practice), and does not comply with the usage minimization of the devices' limited resources. Furthermore extensive description information is necessary [142], while the smart things can only report basic information about themselves and the services they host. Trying to reduce the complexity of registration and discovery, different research directions have been followed in order to provide alternatives or extensions of the UDDI standard [31, 181]. However, also these do not take into account the particular requirements of real-world services.

Römer et al. survey search engines for the real-world [166] and present their own engine in [151] to search for real-world dynamic conditions such as "finding the most quiet place in a city". In their approach, probabilistic models are used to determine which actual sensors to contact for a particular query. Frank et al. study the use of a distributed query infrastructure composed of mobile nodes that can be used to search for the location of real-world objects [58, 59]. A concept that inspired us to implement an application with RFID enabled mobile phones on top of the EPC network infrastructure as published in [69] and described in Section 4.3.4.

These engines and applications focus on leveraging the dynamic conditions of smart things which is a very promising approach to create real-world computational engines. Our target with the Findability Layer, however, is slightly different. Indeed, we enable the search not for particular real-world situations but for services provided by smart things corresponding to user queries. This supports developers, tech-savvies and end-users in finding the services required to create composite applications such as Physical Mashups.

Haodong et al. took such an approach and propose Snoogle [89]. This search engine can be used to find a particular mobile object or a list of objects that are likely to serve the requested service. It uses information retrieval techniques to maintain indexes of keywords corresponding to smart things. These indexes are managed by so called *Index Points* that are local to the smart things (e.g., one Index Point per room). On top of these local Index Points, a single mediator is maintaining an aggregate view of the whole network. A similar approach is taken by Yap et al. in MAX [213]. However, unlike in Snoogle where the smart things push information to the Index Point, in MAX metadata (keywords) are pulled from the infrastructure and the nodes upon queries. These approaches represent powerful real-world search engines but their integration to global networks such as the World Wide Web was not addressed.

Closer to our work, in a theoretical paper, Stirbu proposes leveraging the distributed infrastructure of smart things to achieve a discovery system on the Web [185]. The author considers the idea of devices registering themselves to a Registry Service through a simple HTTP `POST` call. However, the device has to post its full description to the Registry Service that publishes it in an Atom feed. This is a rather strong coupling between the device and the Registry. We avoid this by proposing an architecture inspired from Stirbu's proposal but rather consider the smart things as passive actors that just submit their root URIs either directly or through users. The actual semantic metadata extraction process is handled by the infrastructure. Just as search engines crawl pages of information without hard constraints on those pages, our LLDUs (with the help of a STM Translation Service) crawl the smart things to extract relevant metadata. While we clearly do not pretend offering a comprehensive way to describe and search for all types of smart things on the Web, our approach leverages the lightweight infrastructure put in place at the Device Accessibility Layer of the Web of Things Architecture and extends it with LLDUs that are used to retrieve metadata and offer localized service lookup queries. Furthermore, thanks to the combination of RESTful Web APIs respecting the REST constraints and different implementations of the STM model (e.g., microformats or RDFa[4]), we have a flexible and rather loosely-coupled way of crawling metadata without strong requirements on the smart things themselves.

2.7.3 Sharing Layer

In recent years, the world experiences a renewed trend towards sharing physical resources in all kinds of domains [17]. More specifically, in the Internet of Things domain, data sharing was identified as one of the key enablers [197, 15]. As a consequence several research platforms such as SenseWeb [67, 127], the Global Sensor Networks (GSN) [1], the SOCRADES Integration Infrastructure [36] or SensorBase [28] appeared. These early approaches inspired research in the field but did not use Web standards which requires additional bridges to re-use the data they hold on the Web.

The recent trend to adopt Web technologies and in particular RESTful architectures has influenced platforms such as Pachube or Sensor.Network [84] which propose a solution by providing a central platform for people to share their sensor data. However, these approaches are based on a centralized data repository to which the data is pushed and do not allow authorized and authenticated direct interaction with smart things as we enable it with the Social Access Controller architecture. More importantly, unlike SAC, these platforms are based on their own access control lists which are hard to scale, maintain and manage.

Vasquez [197] introduced the notion collaboration between social networks and smart objects. Furthermore, several research projects have been focusing on leveraging the value of social graphs from existing social networks to share smart things. In [15] Blackstock et al. provide a survey of Social Web of Things projects. Several projects explore the use of Twitter as a publishing and sharing mechanism for smart things [295]. As an example, the

S-Sensors project [13] specifically looks at the use of Twitter as a messaging and sharing mechanism for Wireless Sensor Networks. The SenseShare [188] project allows users to share sensor data with their friends. It also allows owners to apply different filters to the data before sharing it.

While SenseShare was a source of inspiration for the SAC architecture, it presents some shortcomings. Similarly to Pachube or Sensor.Network, SenseShare acts as a data store between the sensors and the clients. SenseShare and S-Sensors further allow sharing the data coming from sensors but do not support direct interactions with the sensors. As an example, one can't enable switching on/off devices by close relatives. Similarly, a Web-enabled Hi-Fi system couldn't enable songs to be played remotely through a RESTful interface which access is managed by the sharing system.

Furthermore, SenseShare or S-Sensors require the use of Facebook, respectively Twitter. Such a tight coupling with a single external service whose contract (API and allowed accesses) is subject to change over time, is problematic. It is also restrictive as it prevents from using a more adapted social network for a specific use-case. As an example, LinkedIn might be more adapted for a B2C (Business to Consumer) or B2B (Business to Business) sharing of smart things. This led us to the interoperability requirement of SAC, which supports different social networks, and enables users to control which one to use for each smart thing.

Recent work, published after the SAC project [71], also leverages recent standards such as OAuth, OpenID and OpenSocial to offer an interoperable solution to real-world data sharing. The SENSE-ATION [178] project enables sharing sensory data coming from mobile phones. Its focus, however, is to offer this information to the developers of applications hosted on social networks (e.g., OpenSocial Gadgets). Similarly, Parimpu [156] offers a platform inside which applications based on sensor streams can be created and shared with Twitter contacts.

2.7.4 Composition Layer

The idea of physical devices that can be composed together and with their environment to create new applications has been long dreamed of by pioneers such as Mark Weiser [203] and often refined since then for instance in the vision of heterogeneous homes [5].

Key in these vision is the notion of end-users being able to create these simple composite applications on their own. Implementations of this vision appeared thanks the evolution of opportunistic programming in which developers have access to tool-chains and programming languages helping them to iterate more rapidly, creating small prototypes with a very limited amount of code [20, 90]. This evolution was further fostered by the developments of Web technologies and languages considered as relatively easy to use and develop upon [163].

Following these developments, the idea of enabling opportunistic applications by mixing the physical world (i.e., smart things) and the Web appeared [110]. Later, these

opportunistic applications were influenced by the concepts of Web 2.0 mashups studied for instance in [215, 216], where end-users are empowered to create simple composite applications on the Web. Wilde proposed the notion of *Physical Mashups* as "[...] new applications using this unified view of the Web of today and tomorrows Web of Things". In [76] we proposed an experimentation and implementation of Physical Mashups and refined the notion together with Trifa and Wilde in [81].

Furthermore, in [163], Roelands et al. discuss the notion of "Do-it-Yourself" in the space of the Internet of Things. They define the concept of "Smart Composables Internet of Things" and explain how physical mashups can contribute to end-user re-usability of smart things.

Several researchers explored ways of easily combining physical objects and Web technologies to create ad-hoc applications. Vermeulen et al. proposed to let users link physical object to composite services on the Web using RFID tags [198]. Similarly, Vasquez and Lopez-de-Inpina explored several simple applications where end-users could combine physical devices with services on the Web [197].

Rather than considering single use-case, we propose an architecture that supports several types of mashups. In this space, researchers identified the mobile phone as being a key platform for Physical Mashups [136], thanks to its ubiquitous Web access. Brodt and Nicklas [22] present an architecture for creating mashups on mobile phones using JavaScript and HTML as well as a mashup server where wrappers for each service are implemented. Mikkonen et al. propose a mashup framework running on embedded devices [140]. The architecture of these solutions targets the use of mobile devices as smart things, in our approach we would like to support other types of smart things and enable their dynamic discovery.

Several projects explore more generic solutions where different smart objects can be composed to create new applications. Vermeulen et al. propose the use of Semantic Web Technologies to enable for semi-automated mashups between the physical and digital world [199]. This approach is promising but taking an automated approach slightly differs from the level of flexibility envisioned in Physical Mashups were people can *re-wire* the physical world [90] themselves, using simple compositions and rules.

Carboni and Zanarini propose the concept of Hyperpipes inspired from the Unix pipes [27]. Hyperpipes are defined using a relatively simple Domain Specific Language based on the concept of sink and source objects that can be connected together. As an example, hyperpipes can be used to redirect the screen output of a laptop to a board. The authors further create a mobile application from which the smart things can be piped together.

We take a similar approach by proposing an extended language set (DSL) for Physical Mashups offering more possibilities and slightly more complex mashups. Furthermore, rather than proposing a single mashup editor we build the framework as a service featuring a RESTful API that can be used to create and run Physical Mashups and Physical Mashup editors.

2.8 Summary

In this chapter, we presented our Web of Things Architecture and its four layers. Rather than being a strictly layered architecture we suggest it should be taken as a set of architectural guidelines that help facilitate, brick by brick, the integration of smart things into Web applications.

In the Device Accessibility Layer, we propose to push the Internet and the Web down to the smart things themselves. We explain how a Resource Oriented architectural approach can be used to model the services smart things have to offer and provide them through a uniform API for the real-world based on REST. Furthermore, we discuss the need for smart things to push events rather than being constantly polled and propose a solution based on the upcoming developments of the Web such as HTML5 WebSockets.

For devices that cannot connect to the Internet and offer their services through a Web server, we propose the concept of Smart Gateways which act as reverse proxies that can be dynamically extended to support new kinds of smart things and proprietary or lower-level protocols. We also provide an evaluation of the differences in terms of performance between an end-to-end HTTP approach and a synchronization-based Smart Gateway mediated approach. With this, we illustrate how a Smart Gateway helps scaling deployments and applications.

In the Findability Layer we propose a set of metadata that covers the most important data required to enable searching for smart things on the Web and to automate processes such as simple UI rendering or the automatic generating of mashup building-blocks for smart things. We implement this model by using microformats combined with the crawling of RESTful APIs. We also extend the network of Web-enabled smart things and Smart Gateway with the concept of Local Lookup and Discovery Units that enable the registration and indexing of smart things. Furthermore, they allow mashup developers and users to formulate several types of local and contextual queries that help them finding the right services offered by smart things.

In the Sharing Layer we emphasize on the importance of having an authentication and sharing mechanism for smart things. Instead of creating anonymous access control lists we leverage social networks and create a proxy called Social Access Controller that bridges social networks and the WoT, implementing a Social Web of Things in which owners of smart things can share their devices with friends, colleagues or relatives. This proxy can be deployed at several places in the network and manages both the access to things and the authentication on social networks through their Web connectors.

In the Composition Layer we adapt an existing mashup framework to Web-enabled smart things and illustrate how this process is made straightforward thanks to the use of Web standards. We further introduce the Physical Mashups Framework that was specifically designed to manage the life-cycle of Physical Mashups.

Finally, we consider a WS-* alternative architecture. We analyze the body of research comparing WS-* and RESTful architecture for smart things and complement these eval-

uations by a qualitative evaluation describing the experience of developers implementing a prototype using both technologies. We conclude that WS-* services have advantages for applications requiring complex service contracts or with high security requirements. However, when considering ease of use, ease of learning, Web integration and fostering public innovation, the RESTful approach seems more adapted.

In the next two chapters, the presented architecture is applied and evaluated in two concrete domains, Wireless Sensor Networks and automatic identification networks.

Chapter 3

Bringing Wireless Sensor and Actuator Networks to the Web

Contents

In the last decade, a tremendous progress in the field of embedded systems has given birth to a myriad of tiny computers, to which all kinds of environmental sensors (e.g., temperature, humidity, vibration, radioactivity, electricity, etc.) can be attached. By interconnecting these devices using low-power wireless communication, a whole new world of possible applications is unveiled. Networks of physically distributed computers, usually called Wireless Sensor Networks (WSN), are valuable tools for monitoring or *sensing* the physical world [6]. Unfortunately, due to the lack of standards most projects in this field

are based on different software and hardware platforms [161] and a common application layer is still lacking [6]. Within such an heterogeneous ecosystem of devices, application development still requires skills and time [146]. Moreover, for each new deployment a large amount of work must be devoted to re-implement basic functions and application-specific user interfaces, which is a waste of resources that could be used by developers to focus on the application logic. Ideally, developers should be able to quickly build applications on top of WSNs.

Several researchers are actively working towards this goal. The advent of IP technologies for WSNs [40, 99] combined with the creation of global consortia such as the IPSO are showing a trend towards using the Internet and its TCP/IP (v4 and/or v6) protocols as the transport protocol of choice for WSNs.

On top of the Internet layer, we join other researchers [41, 214, 33, 39, 126] and propose the use of Web protocols and RESTful architectures as the application layer of WSNs [79]. In this chapter we systematically apply the Web of Things Architecture to two sensing platforms: a general purpose sensing platform and an energy sensing platform. Our aim is to illustrate how adopting a Web approach can enable access to a set of tools that allow creating applications on top of WSNs by recombining ready-made building-blocks, just like with LEGO bricks.

This chapter is structured as follows. We begin by describing our experiments with Web-enabling the Sun SPOT platform. Similarly, we then describe how we Web-enabled an off-the-shelf energy monitoring platform [83, 76, 204]. We describe an open-source Smart Gateway designed to accommodate most smart meters nodes. We further propose applications on top of this platforms and discuss a pilot deployment.

Finally, we demonstrate how, thanks to the other layers, the Composition Layer of the Web of Things Architecture can be leveraged to easily enable the creation of applications on top of WSNs.

3.1 WoT General Purpose Sensing Platform

The Sun SPOT platform [273] is particularly suited for the rapid prototyping of WSN applications because it features a full Java Mobile Edition stack. The version of the Sun SPOT we used for the presented implementation and evaluation has the following main hardware characteristics[1] :

- ARM920T 32 bits CPU with 180 MHz
- 512 kB of RAM
- 4 MB of flash storage

[1]It is worth noting that this is a rather powerful WSN platform when compared to traditional 8 bits platforms. However, it depicts the current trend towards more powerful low-power sensor nodes such as the RN-131 from Roving Networks [269] or the open-source Fly-Port from Openpicus [258].

- an IEEE 802.15.4 radio communication module
- a mini USB connector

The Sun SPOT kits feature a *base-station* node which is a USB module that has IEEE 802.15.4 transport capabilities and can be attached to a computer that will act as a proxy for incoming, non-IEEE 802.15.4 communication.

In this section we describe our two implementations of the Device Accessibility Layer for this platform, once using Smart Gateways [137] and once with end-to-end HTTP connectivity [79, 155] and compare both approaches in an evaluation [81]. We further illustrate how implementing the Findability Layer for the Sun SPOTs allows for searching for them and can be leveraged to create dynamic user interfaces [94] and to allow dynamic WSN integration into mashup tools [4].

3.1.1 Device Accessibility Layer with End-to-End HTTP

In order to empirically test the potential advantages of the Web and HTTP principles, we implemented a RESTful architecture on the Sun SPOT Java sensor nodes. The architecture is composed of two components: an embedded software stack directly deployed on the sensor nodes that implements the HTTP protocol and a transport Protocol Translation component used to forward the incoming HTTP request to the right node over the IEEE 802.15.4 transport protocol supported by the Sun SPOTs.

Embedded Software Stack

Integration at a lower-level is also facilitated by using RESTful interactions with devices [173, 79]. Based upon this, we implemented an embedded HTTP server[2] directly on the Sun SPOT nodes as shown in Figure 3.1. The embedded Web server natively supports the four main operations of the HTTP protocol: `GET`, `POST`, `PUT`, `DELETE`. The HTTP server is deployed on each sensor node, making it an independent and autonomous device. Each Sun SPOT offers a number of sensors (light, temperature, acceleration, etc.), a number of actuators (digital outputs, leds, etc.) and a number of internal components (radio, battery). These, including the Sun SPOTs themselves, are the *resources* of our RESTful architecture. Resources are organized in a tree hierarchy and each of them implements or inherits the four verbs.

Requests for services (i.e., verbs on resources) are formulated using a standard URI. For instance, typing a URI such as `/spot1/sensors/temperature` in a browser requests the resource `temperature` of the resource `sensor` of `spot1` with the verb `GET`. The request is routed by the `RequestDispatcher` to the correct resource as shown in Figure 3.1 on which it invokes the `doGet()` operation. The resource then reads the current temperature using the native Sun SPOT API and sends it to the `Formatter`. While this component can

[2]Based on the NanoHTTPD project: `http://elonen.iki.fi/code/nanohttpd`.

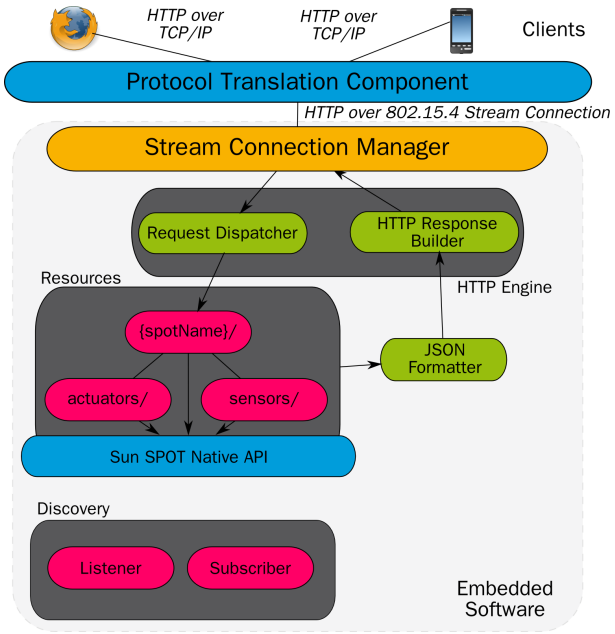


Figure 3.1: Embedded software stack directly deployed on the Sun SPOTs in order for them to natively understand the HTTP protocol. Incoming requests go through the Stream Connection Manager via the Protocol Translation component and are then sent to the Request Dispatcher. This component reads the URI and redirects the request to implementation of the corresponding resource. The resource calls the native API and sends the returned data to the JSON Formatter. The data is converted to JSON and sent back to clients.

support various formats, we decided to use JSON since it is more adapted (than XML for instance) to devices with limited capabilities both because the amount of data transfered is reduced and the parsers require less resources [212]. The JSON data resulting from the call for temperature is shown in Figure 3.3. This data is finally wrapped into an HTTP packet and sent further to the protocol translation component. Alternatively the results can be distributed asynchronously to a URI (Web-hooks) when the values overcome a certain threshold configurable through the RESTful API as well.

HTTP Callbacks (Web Hooks) As mentioned in Chapter 2, for monitoring applications the polling model is sub-optimal. Thus, the nodes can be controlled (e.g., turning LEDs on, enabling the digital outputs, etc.) using synchronous HTTP calls (client pull) as explained before, but can also be monitored by subscribing to notification over an HTTP Callback. For example, a subscription to a feed is done by creating a new rule on a sensor resource and `POST`ing a threshold (e.g., > 100) to `/spot1/sensors/light/rules` together with an HTTP callback.

In response, every time the threshold is reached, the node `POST`s a JSON message to the callback URI. Alternatively, the same mechanism can be used to publish data as an Atom feed to an external AtomPub server. This allows for thousands of clients to monitor a single sensor by outsourcing the data delivery to an intermediate, more powerful server.

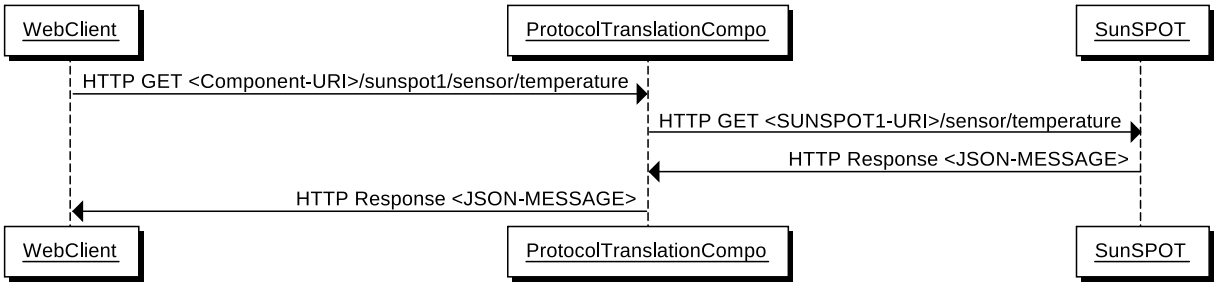


Figure 3.2: Sequence diagram of a protocol translation for the Sun SPOTs. The incoming HTTP over TCP/IP request is routed to the correct node as an HTTP request over the Sun SPOT GCF radiostream protocol (using IEEE 802.15.4).

Protocol Translation Component Since the Sun SPOTs do not natively support WiFi or Ethernet communication, a translation component that forwards requests from traditional Web requests over TCP/IP to the correct Sun SPOT over the IEEE 802.15.4 stream connection protocol is needed.

As shown in Figure 3.2, when receiving an HTTP request from the Internet/Intranet, the translation component reads the request URI and maps it to one of the registered nodes. In case the node is busy, it also serves as a buffer by queuing requests and resubmitting them later.

To allow flexible mashups, we wanted the nodes to be mobile, traveling from base-station to base-station, which requires a dynamic discovery process to find new nodes and register their basic information (the MAC address, a short description, their URI). This process is carried out by a `DiscoveryComponent`, which broadcasts invitation messages on a regular basis on a dedicated port. On their side, the nodes listen to this port and can decide to subscribe to the broadcasting gateway. Then, the `ProtocolTranslationComponent` registers the node's address and is ready to redirect them to the newly registered node.

Management User Interface The Sun SPOT embedded HTTP server we created only offers JSON representations of the resources. While the served JSON messages provide links to related resources and thus holds the connectedness constraint of RESTful architectures, JSON messages are not the best way for human clients to discover the resources. In the Device Accessibility Layer of the Web of Things Architecture we suggested systematically providing HTML representations of resources to allow user-friendly browsability.

Hence, the `ResourceExplorer` component offers to users (e.g., mashup builders) a Web user interface to browse the available services as shown in Figure 3.3. Just as they would navigate on Web sites, they can explore the device hierarchy and test services by clicking on the link structure reflecting the hierarchy of the physical world (e.g., a temperature sensor is the *child* of a sensor node). The explorer dynamically adapts its content to the available devices and is implemented as an AJAX (HTML and JavaScript) application designed to minimizes the connections to the nodes while looking for a service.

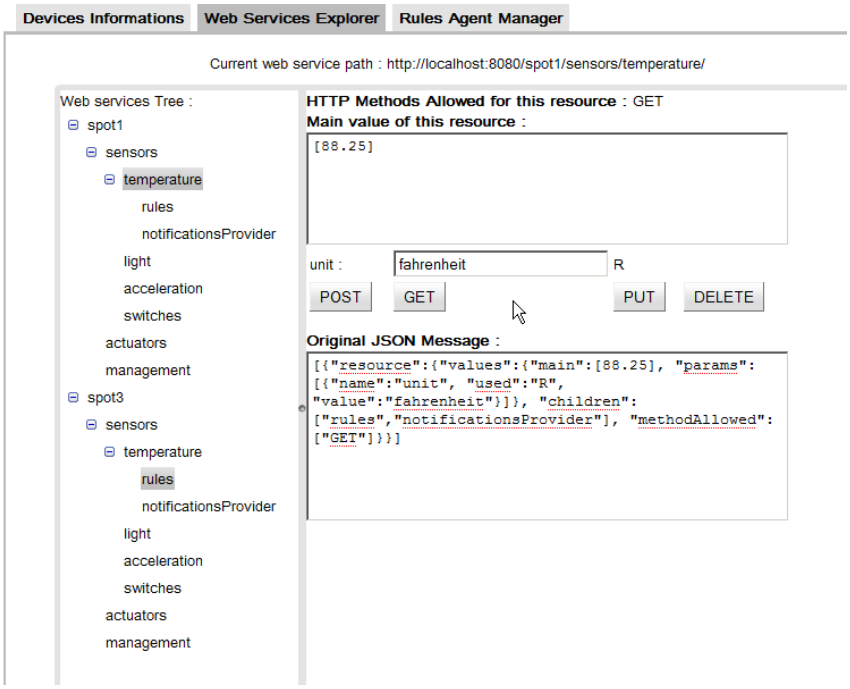


Figure 3.3: Using the AJAX `ResourceExplorer`, users can explore directly the resources provided by the device. This example shows how users can get the current temperature resource values and the formalism of the JSON response.

Device Accessibility Layer with Smart Gateway

In order to be able to asses the differences between end-to-end HTTP communication and communication managed by a Smart Gateway, we implemented a Sun SPOT `DeviceDriver`. There are basically two differences between the Sun SPOT `DeviceDriver` and the end-to-end HTTP stack proposed before.

First, rather than using HTTP end-to-end, the `DeviceDriver` uses the Sun SPOT native communication API (over an IEEE 802.15.4 physical link) and acts an application layer gateway. From a Web client point of view there is no real difference since requests are still formulated with HTTP. Indeed, the `DeviceDriver` takes care of the unmarshalling and marshaling of requests from one application protocol (HTTP) to the other (JSON over IEEE 802.15.4 using the native Sun SPOT Java communication API).

Secondly, rather than relaying every incoming request to the Sun SPOTs, the `DeviceDriver` is based on a caching architecture explained below.

System Overview The Sun SPOT `DeviceDriver` is a software component that can be deployed in the Smart Gateway framework described in Section 2.1.2. To offer a RESTful Web API for the Sun SPOTs functionality, the `DeviceDriver` uses a Web server component provided by the Smart Gateway framework. The resources of the API are shown in Figure 3.4.

The `DeviceDriver` takes care of the operations related to the Sun SPOT specific com-

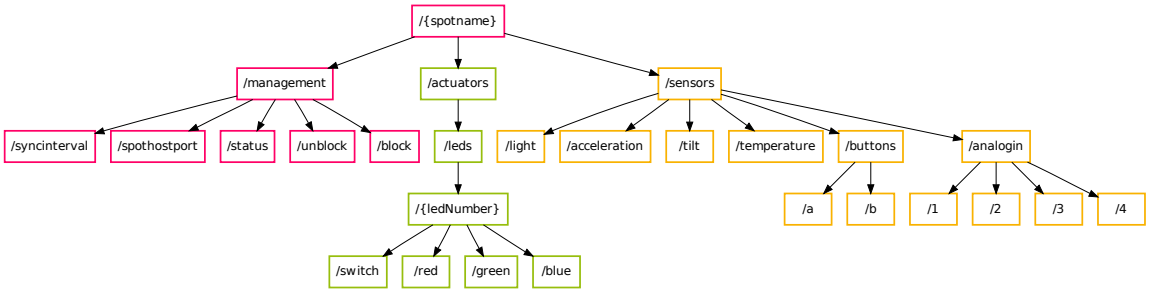


Figure 3.4: Resources of the Sun SPOT RESTful API as offered by the `DeviceDriver` through the Smart Gateway.

munication protocol and API. It further manages the synchronization of the Sun SPOT resources' representations. Indeed, the Smart Gateway has a local copy of the devices' resources and minimizes the actual communication between devices and the Web by locally caching the status of devices. The details of the caching and request process are shown in Figure 3.5. First, a synchronization thread is started by the `DeviceDriver`. After an initial synchronization request, this thread will issue a new request after each reply. A direct consequence of this architecture is that, unlike in the case of the HTTP end-to-end architecture, clients communicate with a `DeviceDriver` (through the Smart Gateway Web server) and not with the Sun SPOT itself.

This has the advantage of fully decoupling the Sun SPOTs from the outside world. Indeed, the device only needs to send an update packet to the Smart Gateway with a frequency short enough to ensure the validity of data. These data are then made available by the Smart Gateway to clients either as HTML or JSON representations. Moreover, the Smart Gateway also offers an Atom representation of the data that can be used to fetch historic data or aggregates of several smart things into a single feed.

On the other hand, unlike the end-to-end HTTP architecture which has the advantage of always returning the most recent sensor readings, the staleness of the retrieved data in the case of the synchronization-scheme will depend on the frequency of updates that can be retrieved from the Sun SPOT.

Software Architecture The Sun SPOT `DeviceDriver` is implemented as an OSGi bundle [137] which allows to dynamically load it in the Smart Gateway framework presented in the Device Accessibility Layer. Here, we provide a summary of the most important classes of the bundle in order to better understand the design of a typical Web of Things `DeviceDriver`. The relationships between these classes is show in Figure 3.6.

SpotManager This class manages the discovered sensor nodes. It launches a thread that will discover the nodes and assign them to communication ports. It further manages the life-cycle of these nodes makes sure that Sun SPOTs to which contact hasn't been successfully established during the last few synchronization rounds get removed.

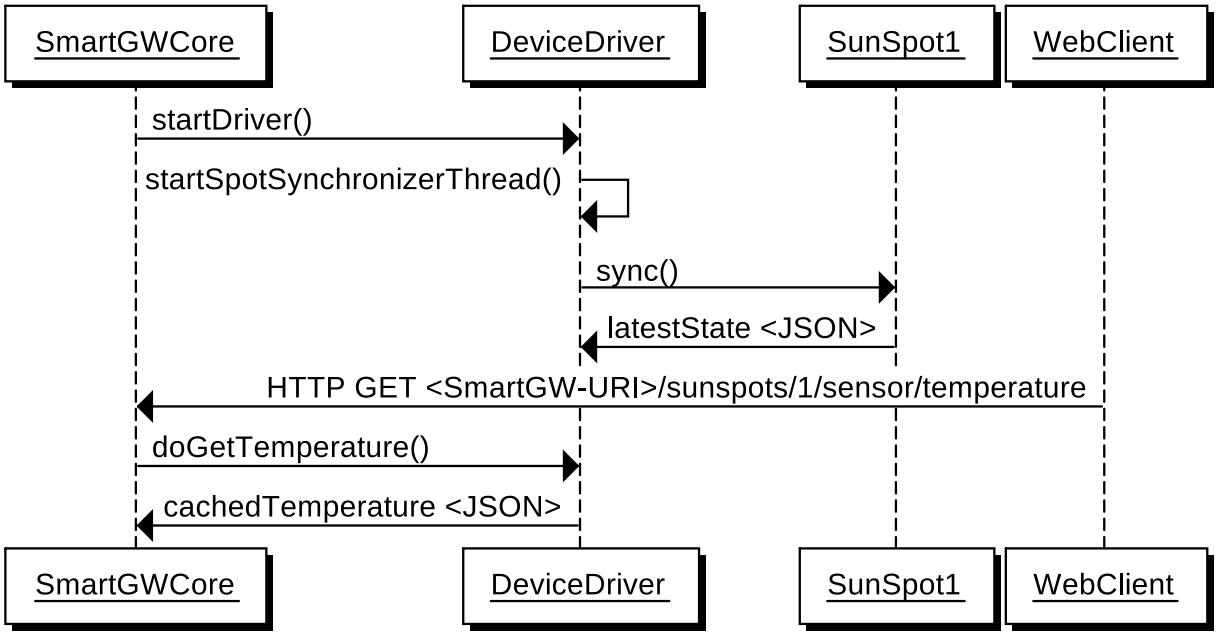


Figure 3.5: Sequence diagram of a request for the temperature of Sun SPOT 1 when using the Smart Gateway synchronization-based architecture. The Smart Gateway serves data to the clients that was cached by the `DeviceDriver` during the last synchronization round.

SpotRepresentation An instance of this class represent the current state of a Sun SPOT, including the state of all its sensors and actuators. It is provisioned and refreshed after each synchronization round through the `SpotManager` class.

SpotCommunicationBackend This class is the proxy between the Sun SPOTs and the `DeviceDriver`. It listens to incoming communication and informs the `SpotManager` about newly discovered nodes. Furthermore, an `SpotSynchronizerThread` instance is started for each new node and manages the initial handshake between the device and the `DeviceDriver`.

SpotSynchronizerThread This is the core of the `DeviceDriver`. Each instance of this thread is in charge of synchronizing a Sun SPOT data on a regular basis, and provisioning the corresponding `SpotRepresentation`.

Quantitative Evaluation

In Section 2.5 we evaluated in qualitative terms what the experience of developing applications on top of the RESTful Web-enabled Sun SPOTs was like for developers. We compared this experience with developing on top of WS-* enabled Sun SPOTs and came to the conclusion that REST was easier to use. Here, we would like to assess the technical feasibility of the approach, evaluating if the RESTful Web-enabled Sun SPOTs perform well enough to implement concrete use-cases. Furthermore, we compare the end-to-end HTTP with the Smart Gateway mediated approach to better quantify the differences and measure the benefits of adding a Smart Gateway to the architecture.

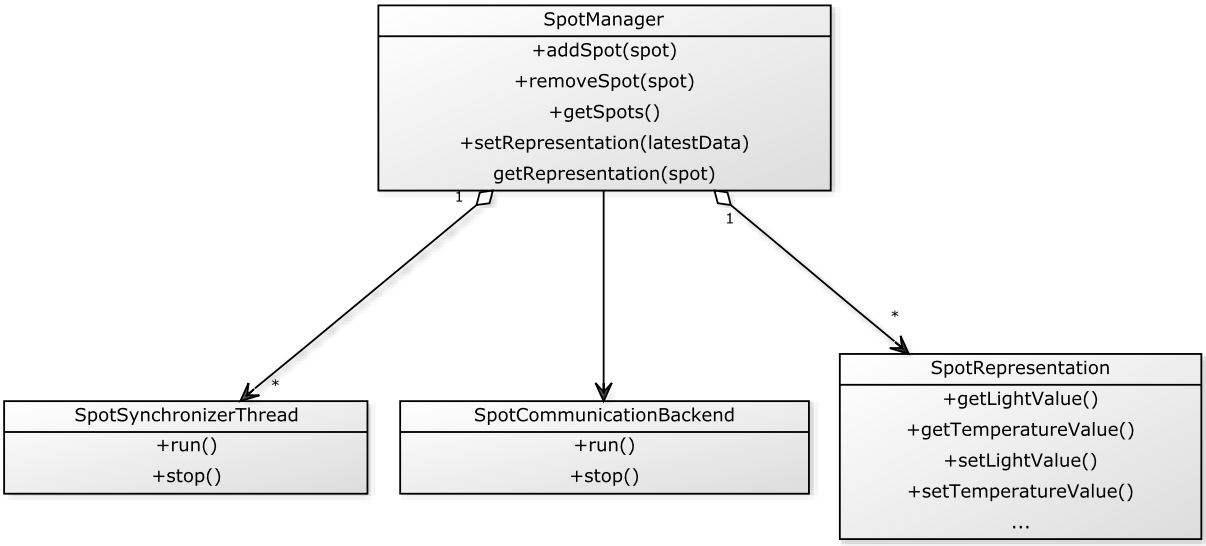


Figure 3.6: Simplified class diagram of the Sun SPOT `DeviceDriver` architecture. The `SpotManager` is the link between the discovery of new Sun SPOTs (handled by the `SpotCommunicationBackend`) and the `SpotRepresentation` regularly synchronized by the `SpotSynchronizerThread`

.

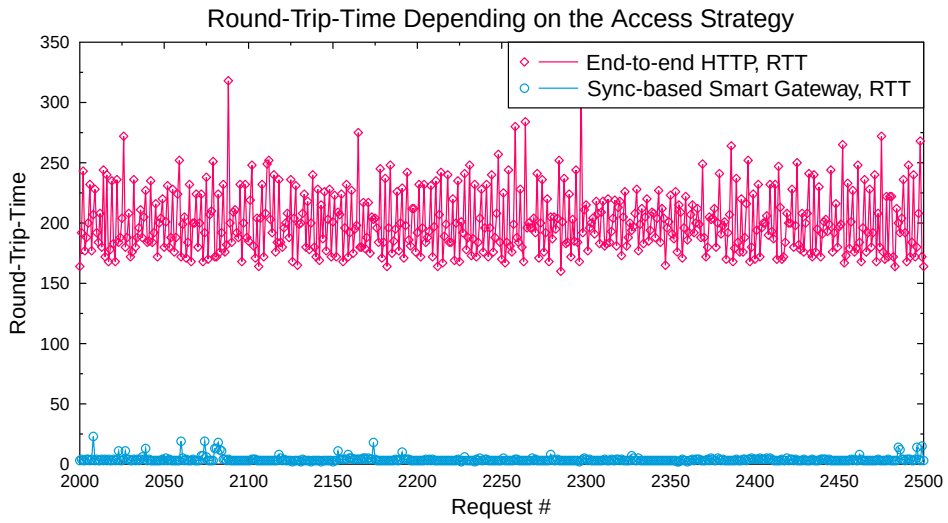


Figure 3.7: Round-trip time for 500 consecutive `GET` requests on a Sun SPOT node when using the end-to-end HTTP implementation and the sync-based Smart Gateway implementation (detailed in Figure 3.8).

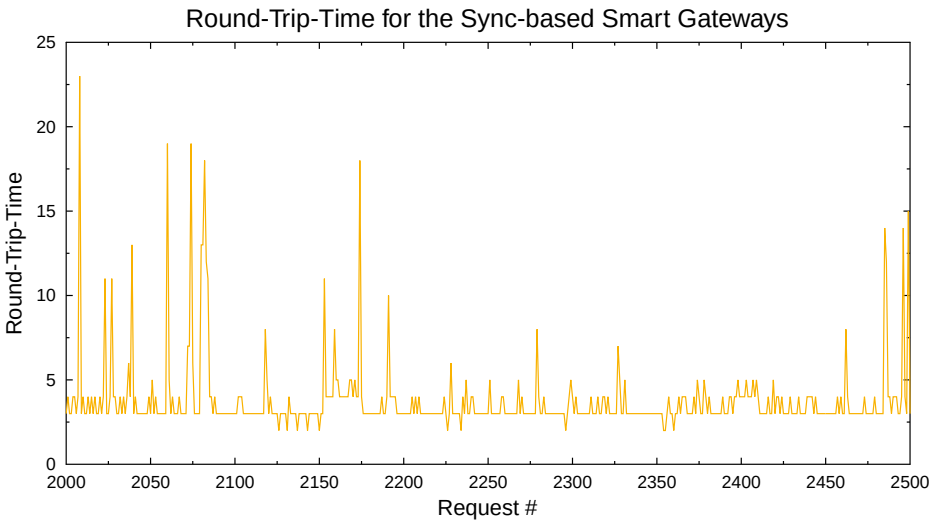


Figure 3.8: Details of the Round-trip time for 500 consecutive `GET` requests on a Sun SPOT node when using the sync-based Smart Gateway implementation.

End-to-End HTTP vs Sync-based Smart Gateway We implemented a simple scenario where a user issues a `GET` request to read the current light sensor value of a Sun SPOT located one radio hop away from a Smart Gateway. We compare the two different architectures described before and show the round-trip time (RTT) for each request in Figure 3.7 and Figure 3.8.

First, each request is routed through the Protocol Translation component to the embedded HTTP server running on the remote Sun SPOT and executed there. For this case, the average round-trip time for 7'000 consecutive requests is 205 milliseconds ($SD = 127.8ms, min = 96ms, max = 8500ms$). The upper graph in Figure 3.7 shows the results for request 2000 to request 2500.

In the second case, we use the `DeviceDriver` on a Smart Gateway and the synchronization-based architecture. The Smart Gateway software is running on a Linux Ubuntu Intel dual-core PC 2.4 GHz with 2 GB of RAM. In this case the average round-trip time was 4.2 ms ($SD = 3.7ms, min = 2ms, max = 111ms$). The results for 500 consecutive requests are shown in Figure 3.8.

The results for both approaches are summarized in Figure 3.7. The fact that these results are far better than the evaluation of the HTTP end-to-end architecture is not surprising. Indeed, in this case each request is served from the cache on the Smart Gateway without direct communication between the client and the Sun SPOT.

However, the trade-off of the sync-based approach is the staleness of the retrieved data which will depend on the frequency of updates sent by the Sun SPOT. Figure 3.9 shows the data age for requests 2000 to 2500. In the case of the HTTP end-to-end architecture this corresponds to the processing and propagation time. In the case of the sync-based Smart Gateway the age will grow until a successful synchronization.

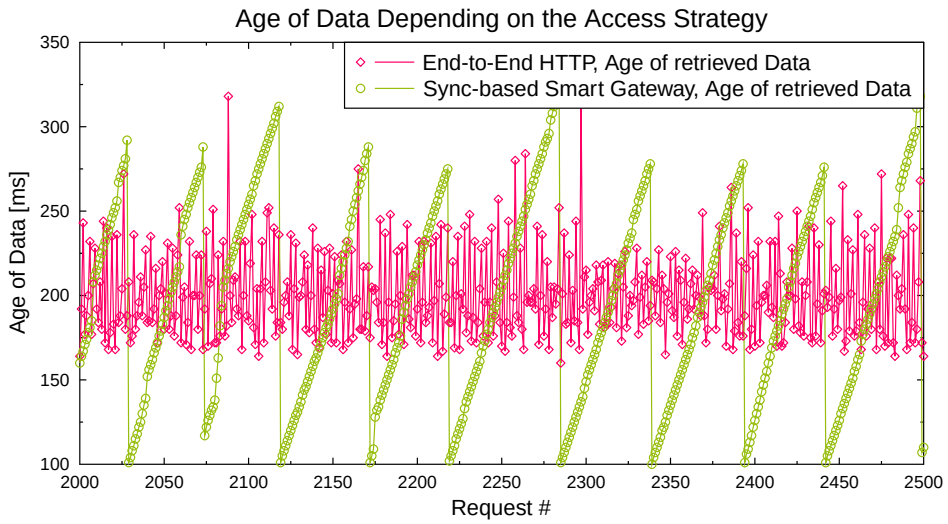


Figure 3.9: Age of the data for each request. In the HTTP end-to-end architecture the data age equals the message processing and propagation time. In the sync-based Smart Gateway the data age grows until a synchronization successfully happened.

Evaluating Concurrency One of the most prevalent advantages of taking the sync-based Smart Gateway approach is to decouple the clients and the actual sensors which greatly improves the scalability of the architecture in terms of concurrency. This is an important point since in the Web of Things vision, smart things are openly accessible on the Web and thus accessed by multiple clients building all kinds of applications.

In this last quantitative evaluation, we want to asses the difference between both approaches in terms of concurrent requests. Similarly to the case exposed before, the Smart Gateway software is running on a Linux Ubuntu Intel dual-core PC 2.4 GHz with 2 GB of RAM. The Web server used for this implementation is based on the Noelios Restlet Engine of Restlet 1.1.7 [268].

Figure 3.10 shows the results when having up to 100 concurrent clients running 100 requests. Not surprisingly, the sync-based Smart Gateway approach scales much better as the success rate of the end-to-end HTTP version drops as soon as the number of concurrent clients is reaching 30.

While the success rate of the sync-based Smart Gateway also drops to 50% as soon as we are reaching 40 concurrent clients, it remains stable and requests are still being served. Furthermore, this strongly depends on the Web server that is being used and high-performance caches that can easily scale with a really important number of concurrent HTTP requests are common nowadays. This means that using a synchronization-based mechanism, thousands of HTTP clients can retrieve simultaneously sensor data from a single device with low response times, while still preserving the freshness of the data collected under a reasonable bound for many applications. This will not hold true for non-cacheable (write) requests that must be sent to devices (e.g., turn on LEDs, change

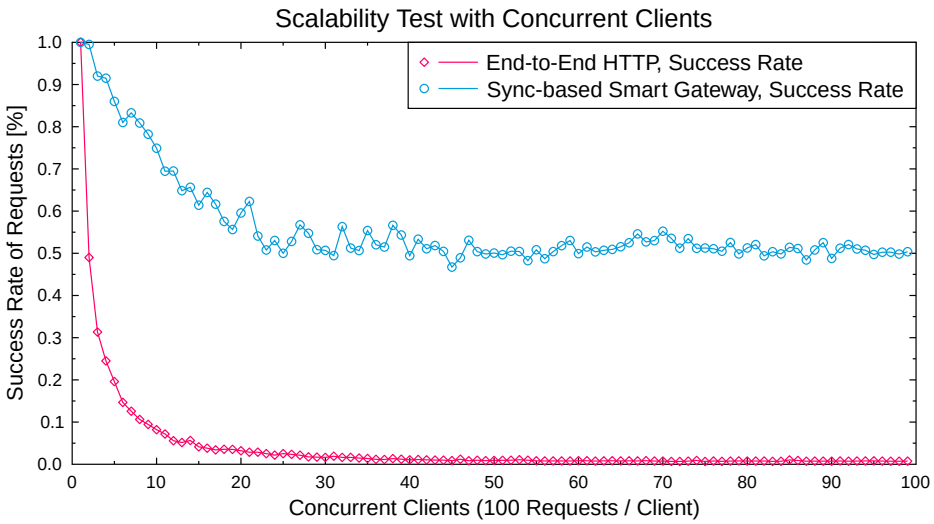


Figure 3.10: Comparing the scalability in terms of concurrent requests for the end-to-end HTTP and the syn-based Smart Gateway version. Each client runs a 100 requests and the clients are concurrently started. The Smart Gateway version scales much better.

application state). As many distributed monitoring applications are usually read-only during their operation (sensors collect data, but users cannot change their status), our architecture exhibits a good scalability level. This enables new types of applications where physical sensors can be shared with thousands of users with little impact on the latency and data staleness.

3.1.2 Findability Layer

To demonstrate the suitability of the Findability Layer presented in Section 2.2, we apply the STM Model to describe the metadata of the Sun SPOTs using microformats. Then, we illustrate the benefits of the approach with two use-cases of the metadata.

Microformats

First, the HTML representation of the Sun SPOTs is modified to embed metadata using the compound of microformats described in Section 2.2.1.

In particular, the `hProduct` microformat is used to describe the static product properties of the STM model as shown in Listing 3.1. The `hCard` microformat is used to describe the manufacturer and owner of the object. The static services properties are described using the `hRESTs` microformat.

For the dynamic properties, these are described with `hCard` which covers the relative location properties. We add the optional `geo` microformat to `hCard` in order to support the absolute location properties (i.e., latitude and longitude). Finally, the QoS dynamic

properties are expressed using the `hReview` microformat. As mentioned before, since these properties are not provided by the Sun SPOT itself they are inherited from the LLDU to which the device is bound. This also allows for dynamic updates of the properties.

```
<span class="hproduct">
  <span class="fn">Sun SPOT Embedded Development Platform</
     span>
  <span class="identifier">,
    <span class="type">epc</span> assigned EPC number:
    <span class="value">urn:epc:id:gid:2808.64085.88828</span>
  </span>
  <span class="category"><a href="http://www.webofthings.com/
     tags/wsn" rel="tag">
  Wireless Sensor Nodes</a></span>
  <span class="brand">Oracle Corp.</span>
  <span class="description">
  This is a Sun SPOT Embedded Development sensor node,
     offering a RESTful Web API through the Web of Things
     Smart Gateways.
  </span>
  <img alt="photo of the Sun SPOT"
  src="http://www.sunspotworld.com/images/spot-reflect.png"
     class="photo"/>
  <a href="http://www.sunspotworld.com/" class="URL">
  More information about this device.</a>
  <span class="price">400$</span>
</span>
```

Listing 3.1: Snippet of the `hProduct` microformat used to describe the product metadata of the Sun SPOTs.

Integration with InfraWoT

The first benefit of the metadata is the fact that search engines such as Google understand them and thus will render adapted results when searching for a smart thing.

Moreover, upon the discovery of a Sun SPOT by a LLDU, the metadata can also be used for indexing purposes and richer keywords are extracted to serve future queries. The data can also be used to render more adapted pages when browsing Sun SPOT on the LLDU. Figure 3.11 shows the results of discovering a Sun SPOT when using the LLDU Query Web User Interface. The metadata is used to render detailed lists of offered resources and services, a map of where the smart things are located and QoS information about the services.

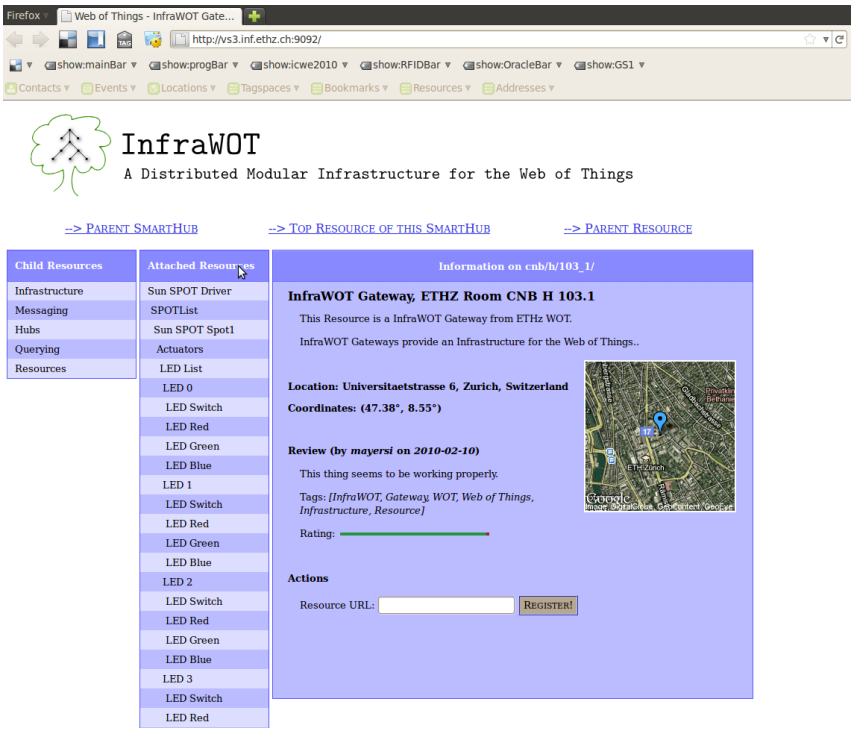


Figure 3.11: Result of discovering a metadata annotated Sun SPOT as seen in the LLDU Query Web User Interface. The Sun SPOT description page is rendered with a map and detailed information about the device and its status.

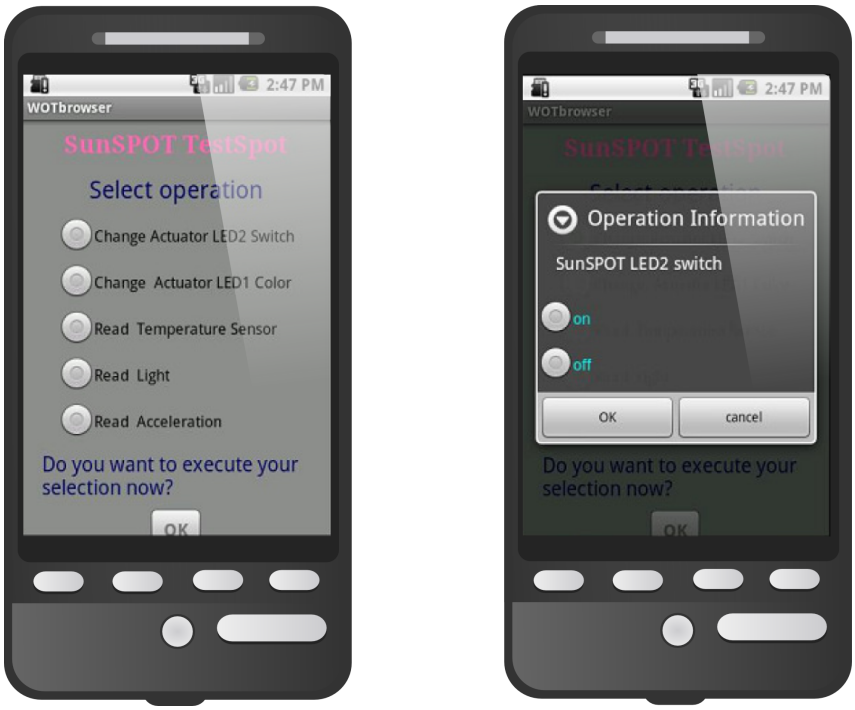


Figure 3.12: Result of discovering a metadata annotated Sun SPOT using the Android WoT Browser. The view on the left gives access to all the resources. As an example, the view on the right can be used to turn the Sun SPOT LED2 on or off.

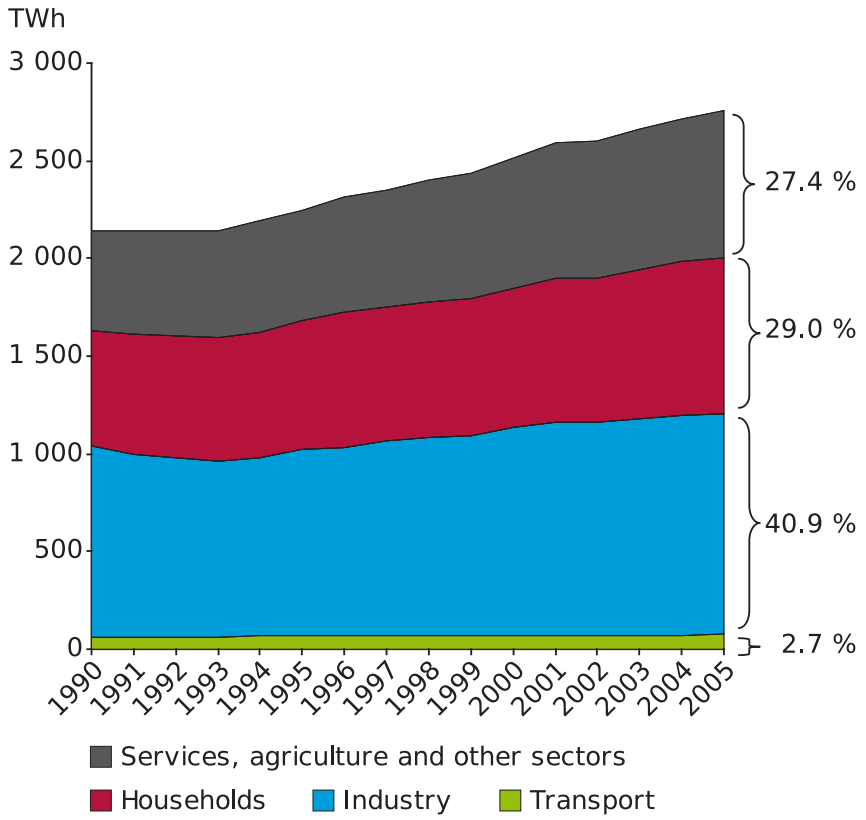


Figure 3.13: This graph depicts the cumulated electricity consumption in the European Union, split by sectors. The proportion of energy consumed by households has been constantly increasing since 1990. (Original source: `www.eea.europa.eu/data-and-maps`)

Rendering Mobile UIs for the Spots

Finally, we created the *Android WoT Browser* [94], an mobile application that scans QR codes containing the root URI of a smart thing from which it fetches the metadata and can dynamically create UIs for operating the smart thing. As an example, Figure 3.12 is the UI generated for the Sun SPOT. It offers a native application from which the main functionality of the Sun SPOT can be called and used. It is worth noting however, that the UIs that can be generated are limited to simple interactions with resources as they rely on the metadata available in the microformats which are not covering fully automated machine to machine interactions for more complex services.

3.2 WoT Smart Metering

A major burden for people, who want to save energy at home, is for them to identify how much energy is consumed by different appliances. How much does a computer consume in operation / when it is powered off? Is the consumption of an energy-saving lamp significantly lower in the long run than a standard lightbulb? Such questions are key to understand where energy can be saved by individuals. Studies have shown that the lack of feedback about their consumption is a major barrier for individuals to save

energy [29]. This is especially important since residential and commercial buildings are major consumers of energetic resources. In the European union, the residential sector accounts for 29% of the overall electricity consumption, a number that has been constantly increasing since 1990 as shown in Figure 3.13 [231]. A similar pattern can be observed in the consumption of households in the U.S. [220]. Hence, to help users identify how their electricity usage relates to difference devices or action, there is a need for systems that give them instantaneous feedback about their consumption [53, 2]. Studies have empirically shown [152] that introducing instantaneous consumption feedback in households helps them to reduce their consumption by 5% to 10%.

Currently available off-the-shelf products that depict the energy consumption in near real time are helpful, but do not fully meet the user needs as they have a high usage barrier and often require complex installations [60]. Furthermore, they are not able to offer the most promising feedback since they lack the ability to provide an appliance-specific break down of the energy consumption and are not able to compare the consumption of individual devices in an appealing manner on a central screen [53]. More importantly in the context of this thesis, *they do not offer open APIs, which makes developing applications on top of them highly complex* and does not foster public innovation [206].

In this section, we present and discuss a system for increasing energy awareness in domestic and office environments built using commercially available smart power outlets named Plogg [262]. We illustrate how adopting the approaches presented in the Web of Things Architecture can help to overcome the above-described limitations. Compared to other solutions, the resulting system is simple to install and does not require any modifications of the wiring, which in many houses in Europe is difficult to access. By providing a Smart Gateway taking care of the network communication and configuration we simplify the deployments such systems in real-world environments. Furthermore, RESTful Web API enables easy interoperability with other applications that can be built on top of the system. Moreover, we present a Web user interface that allow for users to monitor, compare, and control the electricity consumption of devices at home.

This section is organized as follow: First, we briefly describe the sensor nodes that we used for our implementation. Then, we describe the way these nodes were made part of the Web taking the Smart Gateway approach described in the Device Accessibility Layer. Finally, in the evaluation section we present a pilot deployment of the system and discuss the results. Parts of this section are joint work with Markus Weiss published in [83, 76, 205, 204].

3.2.1 Implementing the Device Accessibility Layer

The overall architecture of the system is shown in Figure 3.14 and is composed of 5 main levels. First, at the device level are the appliances we want to monitor and control through the system (e.g., a fridge, a TV). At the second level each of these appliances is then plugged into a sensor node. Then, a Smart Gateway discovers the sensor nodes

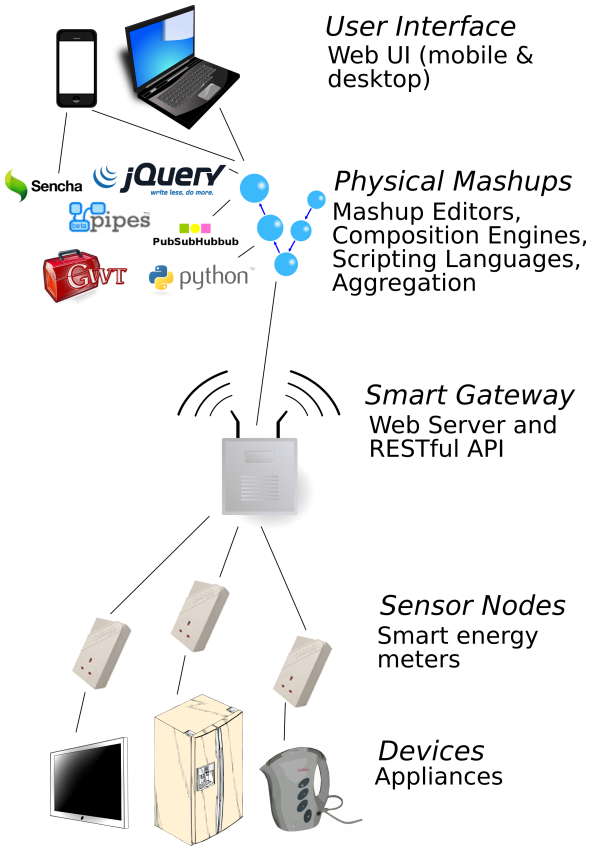


Figure 3.14: Appliances connected to Ploggs communicate with a Smart Gateway offering the Ploggs' functionality as a RESTful Web API. Physical Mashups are then built on top of the API.

and makes them available on the Web through a RESTful API. At the mashup level the sensor nodes' services are composed together to create an energy monitoring and control application, using Web scripting languages or composition tools. Finally, this application is made available through a Web User Interface in a Web browser (e.g., on a mobile phone, a desktop computer, a tablet PC, etc.)

Energy Sensor Nodes

Our system is based on the off-the-shelf commercial Plogg sensor nodes. The Ploggs are a combination of an electricity meter plug and a data logger. Furthermore, they offer a Bluetooth and Zigbee interface to retrieve the current or logged consumption. These factors make them especially suited for appliance level monitoring. Unfortunately, the provided software is limited and does not offer active monitoring. For developing applications on top of the Ploggs a Windows DLL can be purchased. As a consequence, the platforms that can communicate with the Ploggs are limited to the Microsoft Windows platforms and integrating them in other environments (e.g., with mobile phones) requires a lot of expertise.

To overcome these limitations, we apply the architecture described in the Device Accessibility Layer. Since the Ploggs do not have TCP/IP communication capabilities, we built a Smart Gateway for the Ploggs as shown in Figure 3.14.

A Smart Gateway for Smart Meters

As described in the Device Accessibility Layer, a Smart Gateway is a component that abstracts smart things specific protocols and makes the functionality of the smart things available through a Web RESTful API.

Our first experience designing a Smart Gateway for the Plogg is based on the Windows DLL for Bluetooth Ploggs [76]. This C++ gateway first discovers the Ploggs by scanning the environment for Bluetooth devices. The next step is to make their functionality available as RESTful resources. A small footprint Web server (Mongoose [256]) is used to enable access to the Ploggs' functionalities over the Web. This is done by mapping URIs to native requests of the Plogg Bluetooth API through the DLL.

While it worked as expected, we identified a number of shortcomings of this approach: First, by using the Windows DLL we require the Smart Gateway to run in a Microsoft Windows environment. This is a rather important issue. Indeed, in order to simplify its deployment, we envision the Smart Gateway software to be deployed on existing infrastructure nodes such as Wireless routers or Network Attached Storage devices and the vast majority of these devices does not run a Windows operating system but rather variations of Linux/Unix.

Furthermore, building the Smart Gateway on top of a vendor-specific DLL prevents the framework from being used for other types of smart meters. As a consequence, we designed

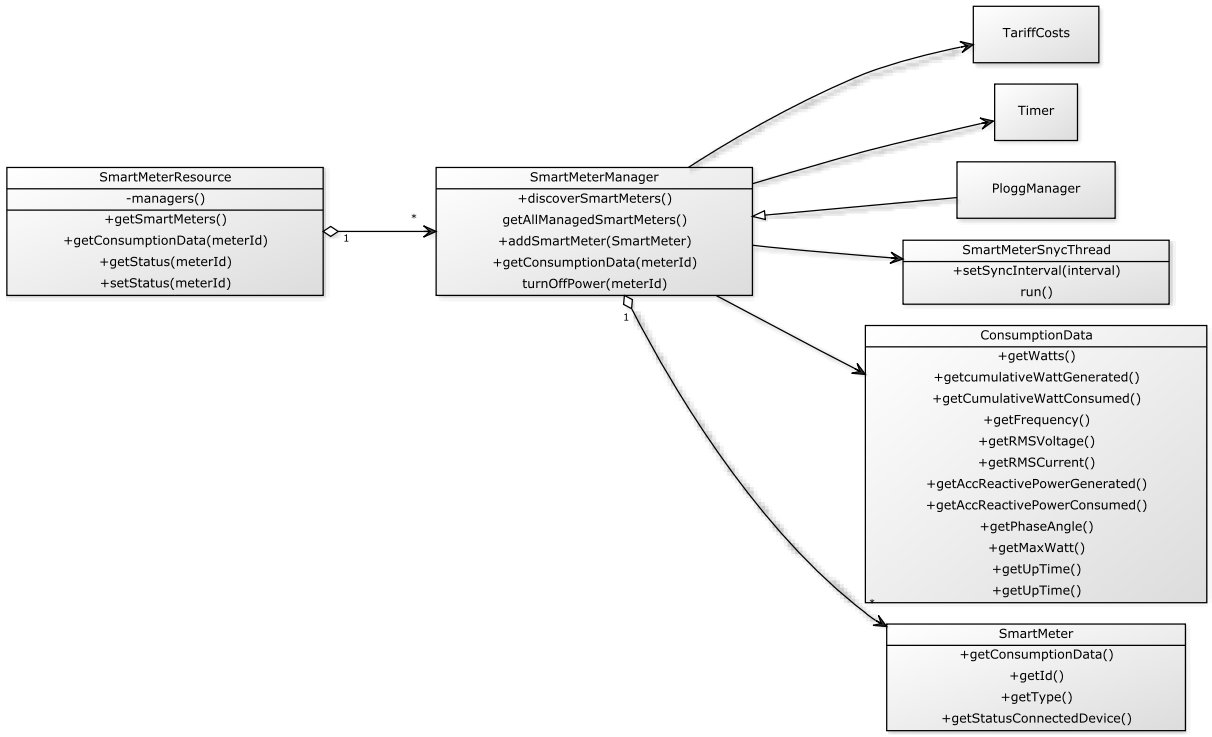


Figure 3.15: Simplified class diagram of the Webnergy framework. Concrete smart meters (e.g., Ploggs) have to extend the `SmartMeterManager` class. The Web representation of the smart meter data is taken care of by the `SmartMeterResource`.

and implemented a second Smart Gateway framework for smart meters that does not require a specific OS and that can be easily extended to support other types of smart meters.

This Smart Gateway framework dedicated to smart meters has been open-sourced as the *Webnergy* project and, to date, is used in a dozen of external research projects [281].

System Architecture The Webnergy framework is a software that acts as a reverse proxy between Web clients and smart meters. A simplified view of framework's architecture is shown in Figure 3.15. It is mainly composed of the following classes:

`SmartMeterManager` This class abstracts the functionality of most smart meters, it has to be extended to meet the constraints of each particular smart meter platform. We implemented a version of the `SmartMeterManager` for the Ploggs. Note that this class represents a group of smart meters rather than a single node. For instance the `PloggManager` manages all the Ploggs sensor nodes that are within reach.

`SmartMeter` This represents a single instance of a sensor node of a particular smart meter platform.

`ConsumptionData` This is an abstraction of the data a smart meter can return. It is the data clients will retrieve through the Web API.

`SmartMeterSyncThread` As for the Sun SPOTs, the Webnergy framework is based on

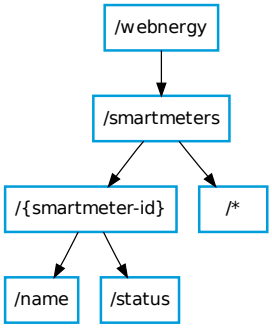


Figure 3.16: RESTful Web API of the Webnergy framework. The functionality of these resources is listed in Table 3.1.

a synchronization strategy. The framework uses the `SmartMeterSyncThread` to synchronize each sensor node of each smart meter platform on a regular basis. As a consequence, the data being served to the clients through the Web API is cached data. The synchronization time depends on the smart meter platform and can be set in the framework.

`SmartMeterResource` This class is in charge of managing the Web API of the supported smart meter platforms. It binds URIs to the resources of the `SmartMeter`s and `SmartMeter Manager`s and is in charge of translating the `ConsumptionData` into the correct representation format.

RESTful Web API The API of the Webnergy framework is relatively simple as shown in Figure 3.16. It is worth noting that all `SmartMeter`s platforms are represented in the same way. All resources can be represented in three formats: JSON, XML and HTML.

The description of the functionality the resources encapsulate is shown in Table 3.1. Note that the smart meters identifiers are unique identifiers generated by the framework based on the hardware address of a particular meter node (e.g., Bluetooth or Zigbee ID).

Implementation To facilitate its portability, the Webnergy framework was implemented as a Java application. As direct consequence is that it can be easily bundled as an OSGi component in order to be integrated to the Smart Gateway architecture we described in Section 2.1.2. When added to the Smart Gateway architecture, it serves as a `DeviceDriver` framework for the integration of smart meters.

3.2.2 Applications

Thanks to the RESTful API offered for the Ploggs through the Webnergy Smart Gateway, creating a compelling user interface (UI) becomes quite easy: all the interface needs to support is an HTTP client library. Since there is ubiquitous support for HTTP across

URI	HTTP Method	Description
/webnergy	GET	Index page
/smartmeters	GET	Lists all the available smart-meters with IDs and names
/smartmeters/*	GET	Shows the consumptions of all smart meters
/smartmeters/smartmeter-id	GET	Lists the consumption data for a particular meter node
/smartmeters/smartmeter-id/name	GET	Gets the human-readable name of a meter node
/smartmeters/smartmeter-id/name	PUT	Sets the human-readable name of a meter node
/smartmeters/smartmeter-id/status	GET	Displays the current status of the meter node and thus of the attached device (on/off)
/smartmeters/smartmeter-id/status	PUT on/off	Switches device on or off

Table 3.1: Resources of the Webnergy RESTful Web API.

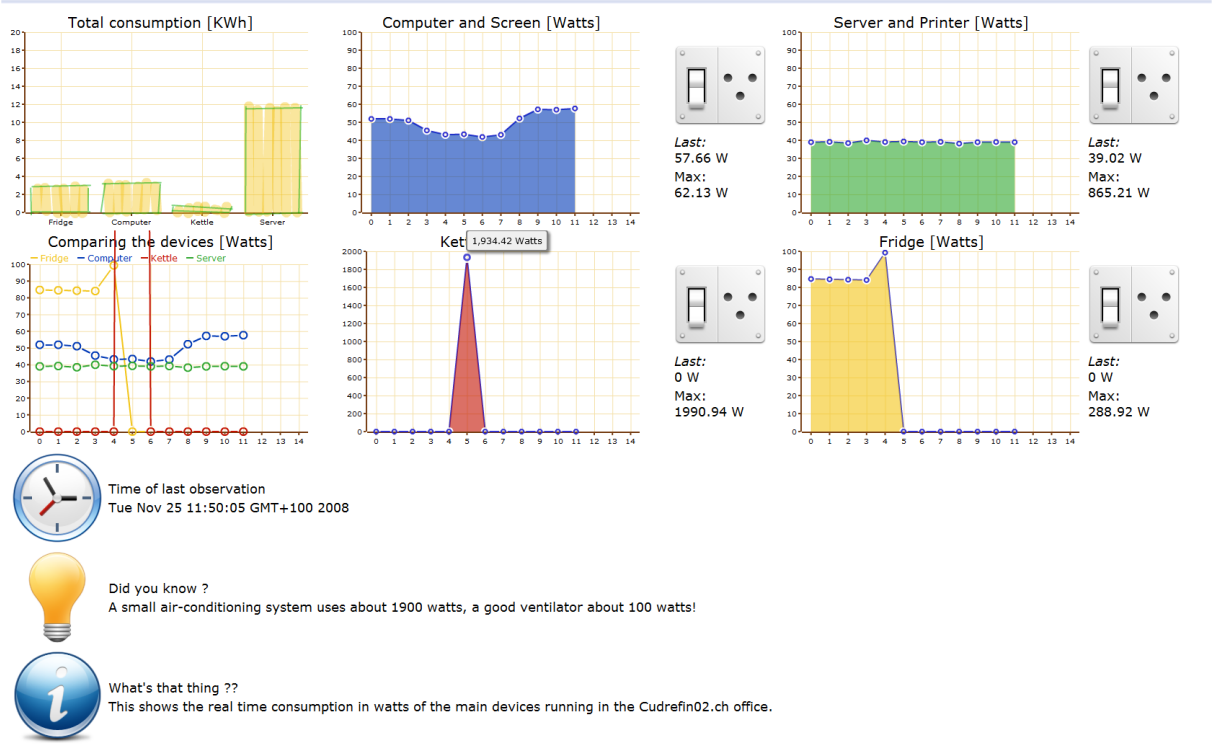


Figure 3.17: The monitoring and control Web user interface for the Ploggs shows the consumption of each connected appliance. The switch icons can be used to power on / off the devices. It is implemented as a simple JavaScript + HTML page calling the Ploggs' RESTful Web API.

all programming and scripting languages, this allows developers to choose literally any language to build applications on top of the smart metering infrastructure. Thanks also to the abstraction of devices behind Smart Gateways, there is no need for the chosen language to support Bluetooth, Zigbee or the protocol a particular smart meter uses.

We illustrate this with a JavaScript Web UI that allows users to monitor and control the consumption of the attached devices from a standard web browser.

Web User Interface

The Web UI was designed to be attractive, easily-accessible, and to display real-time data rather than snapshots. It is a dynamic Web site which was easily built on top of the RESTful API offered by the Smart Gateway. The implementation was made in JavaScript using the Google Web Toolkit (GWT) [239]. To get the consumption data, the UI simply calls the gateway URL every few seconds and feeds the JSON Ploggs' results to interactive JavaScript graph widgets. As shown in Figure 3.17, the resulting interface offers six real-time and interactive visualization widgets. It is dynamically created depending on the number and names of the discovered Ploggs. The four graphs on the right side provide detailed information about the current consumption of all the appliances in the vicinity of the gateway. The two remaining graphs show the total consumption (kWh) and respectively a comparison (on the same scale) of all the running appliances. A switch icon next to the graphs enables users to switch on and power off the devices via their Plogg directly from the Web. Finally, the lower part of the UI provides guidance that shows users effective measures to decrease their energy consumption.

It is worth noting that creating such an interface by directly connecting to the Ploggs would not have been feasible. Indeed, widely popular web languages such as JavaScript do not offer support for Personal Area Network (PAN) protocols such as Bluetooth or Zigbee. However, thanks to the RESTful API, connecting to the smart meters is reduced to being able to call a URI and parse JSON messages, which Web languages can do out-of-the-box.

3.2.3 Qualitative Evaluation

The Web UI and the Smart Gateway were released for public use as a packaged project named *Energie Visible* and are available on the Web for free [228]. The software is a *download and run* application. At the time of writing, it was being used by a dozen of (mostly tech-savvy) people around the globe to monitor the energy consumption of their households. In order to evaluate the suitability of the system to provide feedback in a real-world environment, Energie Visible was permanently deployed at the Cudrefin02 headquarters as described next. Then, we report on formative feedbacks on the usability of the system and the measures people applied due to the increased energy consumption awareness. However, since there existed no fine-grained data on the energy consumption

before our deployment, we cannot quantify the energy savings that were achieved with our system. Furthermore, it is in general difficult to quantify savings as direct effect of a system, since such real world deployments contain numerous side effects that cannot be controlled or kept constant. Thereafter, we provide insights from the developer perspective gathered from experts who developed on top of our RESTful system after we released the open-source code.

Pilot Deployment at Cudrefin02 The prototype was deployed at Cudrefin02, a private swiss foundation active in the field of sustainability and has been running reliably since November 2009. Regarding the given setting and the goal to raise consumption awareness, our prototype had to fulfill certain requirements. Due to the fixed setting in the office, the fact that continuous operation should be ensured and users were not technological affine, the prototype had to be easy to install and simple to use. Hence, the UI had to be developed in an attractive and easily accessible way (no additional software to learn or install) that allows both, staff and visitors, to become aware of the electricity consumption of appliances running at the headquarters.

From a feedback perspective, the breakdown of the entire energy consumption, e.g., for specific rooms, appliances, or times of the day, is a powerful way of establishing a direct link between action and effect. This considerably improves the intensity of reflection and interpretation of a measure or omission [53]. However, besides providing this possibility to apportion the entire consumption to single devices, it is important to enable users gathering feedback frequently and in real-time. This allows for users to relate feedback to a certain behavior or device usage [2] and thus take effective measures regarding the energy consumption. Continuous feedback has thereby been proven to be most effective [195]. In addition, only feedback that is at hand when needed is able to satisfy users' spontaneous curiosity.

The system is deployed in the ground-floor office of the headquarters. There, the Ploggs are used to monitor the energy consumption of various devices such as a fridge, a kettle, two printers, a file-server, and computers including their screens. The Smart Gateway software is deployed on a small embedded computer consuming about 20 Watts, but also used as a file-server and multimedia station. At startup, the Smart Gateway discovers all Ploggs within communication distance and queries their consumption values every 20 seconds. The frequency of calls is the observed time needed to establish the Bluetooth communication with each plogg and get the data returned to the Smart Gateway. As a consequence, the system achieves only near real-time measurements. However, in case of a direct request the corresponding Plogg is queried right away and the value can be gathered in about 2 seconds. It is worth noting that according to further experiments this relatively slow communication time is due to the Bluetooth stack used by the Ploggs and can be improved at the firmware level.

During the whole time a large display in the shop-window of the office encouraged people passing by to experiment with the system. The staff used the system to monitor the consumption and remotely control the appliances by browsing to the web UI on their

desktop computer.

User Formative Feedback

An exhaustive analysis of the system's impact on the electricity consumption of the users is outside of the scope of this thesis and is discussed in [204, 135]. Nevertheless, to better understand the usage of the system and the feasibility of real-world Web-based smart meters, here we report on user feedback. This section is based on guided interviews with the foundation staff after eight month of operation and verified based on the logged data.

The aim of the deployment was to raise consumption transparency to help visitors as well as members of the staff to better understand how much energy different devices throughout the office consume in operation and in standby. At the beginning, staff members had to get used to the system and started exploring the energy consumption of different devices. By experimenting with the kettle and the different printers, for example, they learned that the amount of water heated up as well as the type of printer being used has a high impact on the energy consumption. Staff members also started instantly comparing how much energy their office desks consumed. After an initial period of about a month, the staff members' initial curiosity was satisfied and they thus reported they started using the system less. However, they then looked more into details such as the standby consumption of different devices, the accumulated consumption over time, and once identified where electricity was wasted, took effective measures to conserve energy. For instance, they identified that the computer screens, even when completely turned off, still consumed a considerable amount of energy and that even when not used for several consecutive days, the file-server kept consuming about 30 Watts. To prevent this waste of energy, concrete technical and behavioral measures were implemented. In order to avoid residual electricity usage such as with the computer screens, all the appliances within the headquarters except for the fridge are now connected to a central power switch. As a policy, the last person leaving the office is now in charge of turning off the central power switch. Moreover, the computers and the file-server are now shutdown every evening. The information item at the bottom of the Web UI, which displays contextual hints, was also seen as very helpful. However, the staff reported that it would be better if the hints related to the current consumption. During the whole time the system has been demonstrated to visitors. They especially liked to remotely turn devices on or off and the possibility to see the accumulated consumption together with the monetary cost caused by the device.

Our formative evaluation hints that the prototypes' functionality is well suited to increase the consumption transparency and helps users to save energy. We realized that in the exploratory phase people like to interact with the system and are looking for a simple and fun way to identify the electricity consumption of different devices. Later, it becomes important to provide functionality that constitutes added value to keep users motivated. Therefore, it is not sufficient to provide feedback on the current consumption on appliance level, but also account for how much energy has been consumed over time. Thus, the system allowed users to identify sinks where energy was wasted. In addition, we identified

it is important to relate the consumed energy to the incurred cost to draw conclusions and take effective measures. Furthermore, the suitability of a Web UI was confirmed as none of the users had problems understanding and using the Web page.

The Developer Perspective

The main goal for offering a Web layer on top of the Ploggs is to illustrate the ease development of applications on top of an otherwise closed system. The Web-enabled smart power outlets thus offer a platform for the fast prototyping of energy awareness-related demonstrations and applications.

Since the release of Energie Visible on the web, several development teams have asked for using our software to build new prototypes upon. As a consequence, the Webnergy project was open-sourced [281] and the Smart Gateways used in several projects, ranging from personal applications to commercial demonstrators. We followed two students (external to the project) who built applications on top of the Ploggs Smart Gateway and report on their feedback here. This, unstructured data complements the structured developers-evaluation presented in Section 2.5.

For the first developer, the idea was to build a new mobile energy monitoring application based on the iPhone and communicating with the Ploggs. The final application is shown in Figure 3.18. The application offers three main functionalities. First it lets users get an aggregation of the consumption of all monitored devices in their environment. Then, a summary of the consumption of each discovered device (i.e., discovered Plogg) is presented in a list view. From this list, users can access the detailed information for each Plogg such as the consumption of the connected device over time and its expected electricity cost per year.

In the second case, the goal was to demonstrate the use of a browser-based JavaScript Mashup editor with real-world services [149]. According to interviews we conducted with the students, they in particular highlighted the ease of use of a Web "API" versus a custom "API". For the iPhone application a native API to Bluetooth did not exist at that time, but like for almost any platform an HTTP (and JSON) library was available. One of the developer mentioned a learning curve for REST, but emphasized the fact that it was still rather simple and that once it was learned the same principles could be used to interact with a large number of services, languages, and possibly soon also with smart things. The students finally outlined the direct integration to HTML and Web browsers as one of the most prevalent benefits.

They explained how this significantly eases the development on the vendor's side, since applications can be built on languages for which a plethora of libraries and frameworks are available. Furthermore, the use of popular languages makes it easier to find adequate developers. This also unveils the possibility for external developers to create small web applications and plug-ins on top of smart meters.

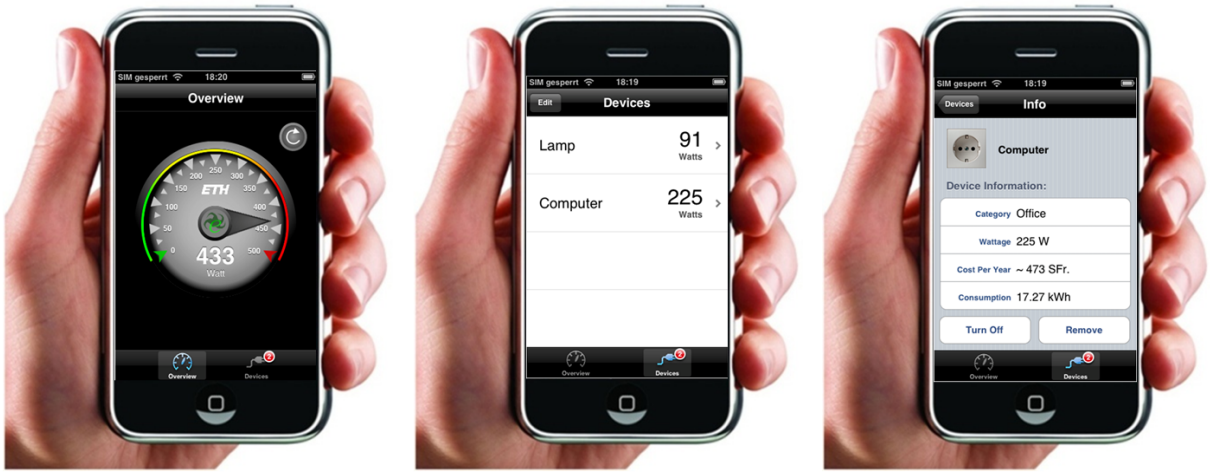


Figure 3.18: The mobile user interface built using the Ploggs RESTful API. First, the aggregated consumption is displayed. Then, from a list of connected devices the user can select a particular device and get details about its consumption over time. (Source [204])

3.3 Sharing Layer

Thanks to the implementation of the Device Accessibility Layer and the Findability Layer for the Sun SPOTs and the Ploggs, they can be shared using the SAC architecture described in Chapter 2.3.

We demonstrate this by sharing the Sun SPOTs Smart Gateway: First, the Smart Gateway needs to be registered with the SAC server. This is done through the SAC API (see Section 2.3) with a `POST` on `<SAC-BASE-URI>/gateways` with the following parameters (as a URL-encoded form):

`gatewayBaseUri` The base URI of the Smart Gateway.

`subUri` The sub-URI from which the sharing should be enabled, here `/sunspots`. This can be useful to share just one type of smart things on a Smart Gateway that manages several (e.g., sharing just a particular Sun SPOT).

`description` A human-readable description of the Smart Gateway.

`username` The username part of the credentials that should be used to get access to the Smart Gateway.

`password` The password part of the credentials that should be used to get access to the Smart Gateway.

`userId` A list of SAC internal users (they need to be currently authenticated) that should be able to share resources managed by this Smart Gateway. The list of currently authenticated users and their identifiers can be retrieved using the `/users/loggedIn` resource.

Once added, resources from the Smart Gateway can be shared with trusted connections using the API. For instance, sharing the `temperature` resource of a Sun SPOT requires

a `POST` on `/gateways/{gatewayBaseUrl}/shares` and the following parameters:

`resourceSubUri` URI of the resource that should be shared, here `/sunspots/spot1/temperature`.

`userId` SAC user sharing the resource (needs to be currently authenticated).

`networkUserId` SAC identifier of a trusted connection on a social network which can be retrieved using the `/users/{userId}/friends` resource.

`networkUserName` Social network identifier of the trusted connection to share with.

Once the sharing process is finished, a message will be posted directly to the trusted connection through the social network connector. The message contains a URI that can be used to access the shared smart thing resource through SAC. In the case of our example, the URI looks like `http://vswot.inf.ethz.ch:8091/gateways/vswot.inf.ethz.ch:8081/resources/sunspots/Spot1/sensors/temperature` where `vswot.inf.ethz.ch:8081` is the Smart Gateway base-URI and `/sunspots/Spot1/sensors/temperature` the relative URI of the shared temperature sensor. Finally, it is worth noting that SAC will grant access to this resource to the trusted connection if and only if she was successfully authenticated on her social network.

One of the very interesting advantages of modeling the Sun SPOTs and Ploggs with a RESTful architecture and in particular of the connectedness constraint, is that is allows the SAC to discover the resources and sub-resources the owner can share for a given Smart Gateway. This is achieved using the crawling algorithm introduced in Section 2.2.1 where the the resource and sub-resources are extracted by following links in the HTML representation. Furthermore, the operations one can execute on resources are identified by calling the HTTP method `OPTION` for each resource. This returns the methods supported for a particular URI, e.g., `PUT`, `POST`, `GET`, etc.

To illustrate this process, consider an owner who wants to share the RESTful Ploggs. The user gives the credentials to the Ploggs Smart Gateway alongside with its base URI, i.e., `<BASE-URI>/smartmeter`. The crawling engine will browse that page and detect the links to the sub-resources of the Ploggs such as: `/smartmeters/lamp` and `/smartmeters/lamp/status`.

For each resource, the crawler will also retrieves the HTTP methods it supports. For example, the `lamp` resource only supports `GET` whereas the `status` resource also supports `PUT` to switch on or off the device. The result of this process is a list of resources that can be shared. Figure 3.19 shows the example of such a list after crawling the Sun SPOTs Smart Gateway with the Friends and Things Web application.

This approach is valuable because it only requires for smart things to be based on a truly RESTful architecture with no additional constraints on the embedded metadata. However, from the figure, it appears clearly that this is not the most intuitive type of representation that can be brought to users. More tailored representations, with better descriptions of the resources, can be extracted from implementations of the STM model

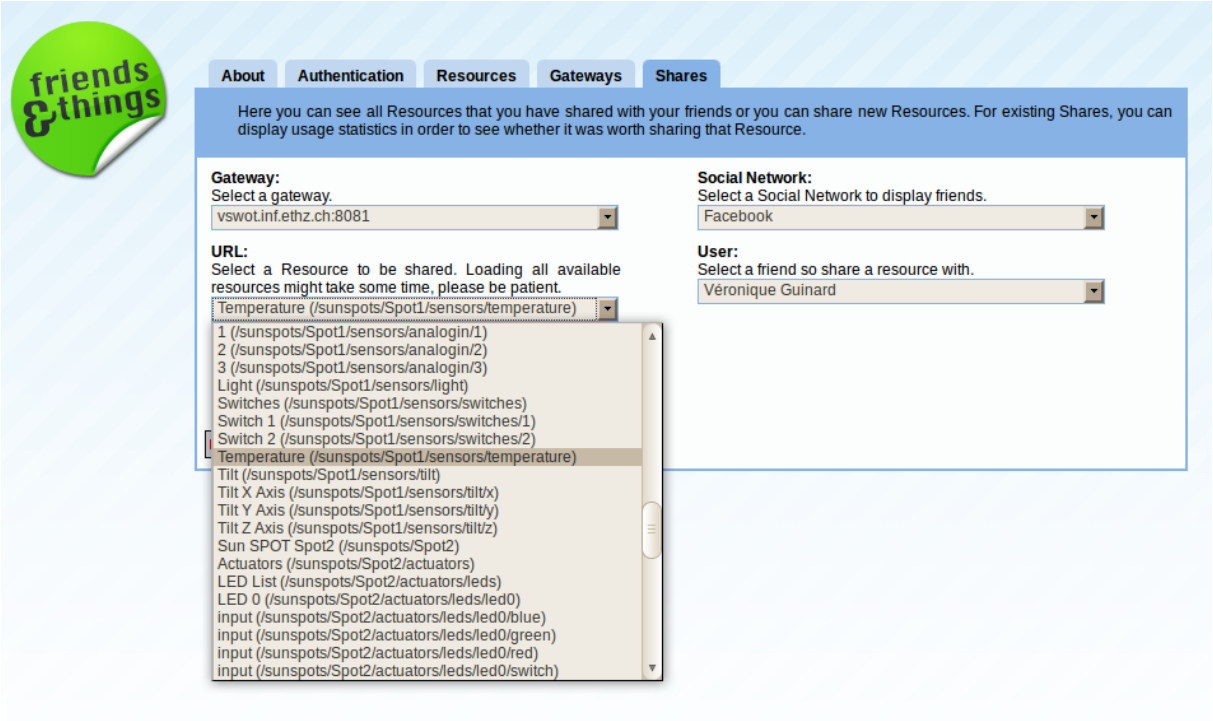


Figure 3.19: Screenshot of the Friends and Things Web user interface (built on top of the SAC API) after crawling the resources of a Sun SPOTs Smart Gateway. All resources are extracted and ready to be shared with trusted connections.

as explained in Section 3.1.2.

3.3.1 Quantitative Evaluation

To quantify the overhead of using a Social Access Controller we evaluated our implementation when sharing the Sun SPOTs. We setup a scenario with two computers. Similarly to the evaluations proposed before, the Smart Gateway software is running on a server featuring Linux Ubuntu with an Intel dual-core PC 2.4 GHz with 2 GB of RAM and the Noelios Restlet Engine Web server (Restlet 1.1.7 [268]). The Sun SPOTs sync-based Device Driver is deployed in the Smart Gateway and a Sun SPOT node is bound to this driver through an IEEE 802.15.4 wireless link.

Additionally, a laptop client is located one hop away from the server and both are connected through a Gigabit Ethernet link. The laptop client runs a Linux Ubuntu OS and is an Intel dual-core PC 2.67 GHz with 4 GB of RAM.

After sharing the Sun SPOT as described before, we test the overhead of accessing it through SAC. The authentication and authorization process is using Facebook based on our `FacebookConnector` and accessing the URI shown in the first line of Listing 3.2, i.e., requesting the light sensor of the Sun SPOT through the SAC authentication and authorization proxy. An extract of the actual HTTP request is shown in Listing 3.2. It is worth noting that in order to be able to monitor the requests and results from an external

tool, we do not use an encrypted channel. However, when deploying a SAC, encrypted HTTP communication should be systematically used to avoid attackers intercepting the social network authentication keys.

```
1 GET /gateways/vswot.inf.ethz.ch:8081/resources/sunspots/Spot1/
     sensors/light HTTP/1.1
2 Host: vswot.inf.ethz.ch:8091
3 [...]
4 Keep-Alive: 115
5 Cookie: FacebookConnect.sessionKey=2.AQA7DlNqyVIhqZWE
     .3600.1311098400.0-1417076934;
6 FacebookConnect.userId=1417076934; FacebookConnect.loggedIn=1
```

Listing 3.2: HTTP request used to access the light sensor of a Sun SPOT through SAC. The request contains a cookie with the Facebook session keys that will be used to check whether the user is authenticated and authorized to access the resource.

We run 1000 sequential requests and back our data with several iterations of the 1000 test runs. An extract of results can be seen in Figure 3.20 where they are compared with the same requests when directly requesting the Sun SPOT resource, without going through SAC. Going through SAC, the requests have an average RTT of 218 ms ($min = 204, max = 830, SD = 24$). Without SAC we observed an average RTT of 9 ms ($min = 6, max = 40, SD = 2$). As a consequence we can conclude that in this setup SAC generates an overhead of about 200 ms.

The overhead is reasonable for most applications we envision. However, these results can easily be explained since most of the RTT is due to the latency while contacting Facebook (located 14 hops away from the SAC server) to authenticate the user using OAuth. Indeed, in our implementation SAC will contact the social network and check the authentication before each request. Hence, the overhead could be greatly reduced by caching the authentication keys on the SAC server side. Rather than storing them by session (which does not respect the *Stateless Interactions* constraint of REST), these keys could be stored for each user and invalidated after a certain time (sometimes specified by the social network). However, such a cache is a trade-off since systematically controlling the keys offers a better level of security.

For requests intensive applications (e.g., real-time monitoring applications) where the latency might play a role, the SAC server can be used for the initial subscription after which the content can be delivered by the sensor or Smart Gateway directly to the client through a secure Web push channel such as the one implemented by the tPusher service over secure WebSockets [125] (see Section 2.1.3) or simply through a secure (HTTPS) HTTP Callback mechanism.

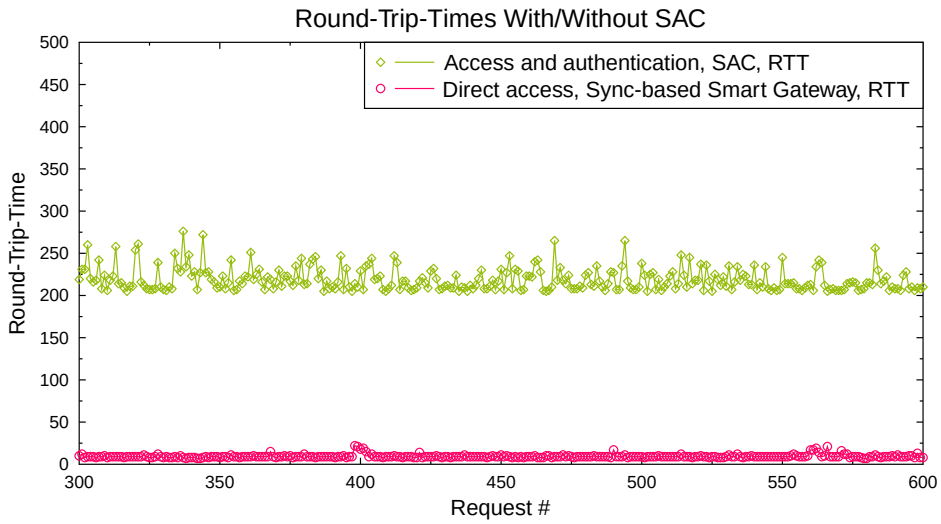


Figure 3.20: Round-trip time for 300 consecutive requests for the `light` resource of a Sun SPOT with or without authentication through the SAC server. The authentication process has an overhead of a about 200 ms per request.

3.4 Composition Layer: Cross-Device Physical Mashups

This section applies the idea of *Physical Mashups* as described in the Composition Layer to wireless sensor networks. Following the three levels of mashability introduced in the Composition Layer: *Manual Mashup Development*, *Widget Based Mashup Development* and *End-User Development with Mashup Editors*, we propose, design and implement three Physical Mashups combining the RESTful Sun SPOTs, the RESTful Ploggs and virtual services.

3.4.1 The Ambient Meter

This first prototype is joint work with Vlad Trifa and was published in [76, 190]. It demonstrates how real-world services provided by physical devices can be combined together thanks to their RESTful Web APIs. Hence, it can be classified as a Manual Mashup Development. However, it further illustrates how the smart things themselves can communicate with one another to create new applications thanks to the ubiquitous availability of the HTTP protocol.

The Ambient Meter is a mobile device that displays the level of energy consumption of the place it is currently located in by changing its color. It can be taken from one place to the other and adapts to the place it monitors automatically, without the need for human intervention. Depending on the total amount of energy consumed in the room it is located in, the Ambient Meter changes its color from very green (i.e., the amount of energy consumed in the room is low) to very red (i.e., a lot of energy is currently consumed

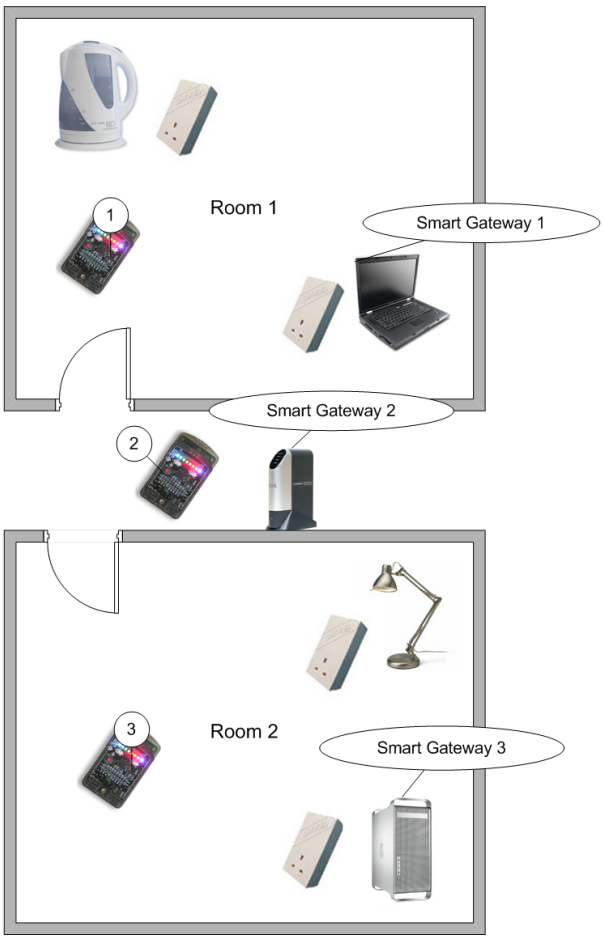


Figure 3.21: Demonstration settings of the Ambient Meter. Every 5 seconds the Ambient Meter (implemented with a RESTful Sun SPOT) polls a URI for an assessment of the energy consumption observed by all the Ploggs the Smart Gateways discovered.

in this place).

The Ambient Meter is built with our implementation of the end-to-end HTTP Sun SPOT and it uses the Ploggs for energy monitoring as well as the Ploggs Smart Gateway deployed alongside with an LLDU (Local Lookup and Discovery Unit) for resolution of its current location. Every 5 seconds, the Ambient Meter will poll the following URI using the GET method on `http://localhost/webnergy/smartmeters/*.json`. When the meter is located in Room 1, as shown in Figure 3.21 (step 1) it is bound to the Smart Gateway 1, meaning that localhost in this context is bound to the address of Smart Gateway 1. Thus, the result of the call is going to be the JSON representation of the energy consumption of all the Ploggs discovered by the Ploggs' Smart Gateway 1. Placed in the hallway, the Ambient Meter binds itself to Smart Gateway 2. Using the same URI as before it will get the energy consumption of all the devices monitored. Again, the same process occurs in Room 2, where the Ambient Meter gets the load of the desktop computer and the lamp.

Integrating all the real-world devices of this prototype would have been rather time consuming if the Smart Gateway, the Ploggs and the Sun SPOTs were only offering their native (proprietary) APIs. Thanks to the RESTful approach the integration work was reduced to building a simple Web mashup, where all the services are invoked by means of simple and lightweight HTTP requests.

3.4.2 With Clickscript

This second prototype illustrates how physical mashups can be created by end-users as well. In this use-case we use a RESTful Sun SPOT to actuate a RESTful Plogg when reaching a certain temperature. To implement this use-case we use the ClickScript Mashups Editor [149]. As shown in Figure 3.22 we use the adapted version of the Clickscript presented in Section 2.4.3.

In this simple Physical Mashup, the room temperature is obtained through the Sun SPOTs. This can be either obtained by regularly polling `/spot1/sensors/temperature` which requires an additional loop building-block or by subscribing, through WebSockets and the tPusher service, to a new rule on the Sun SPOT. If this temperature exceeds 35 degrees, the Ploggs will be actuated and the attached fan will be turned on.

Since Clickscript was written using purely Web languages (JavaScript and HTML) it cannot use resources based on the low-level proprietary service protocols of the Ploggs and the Sun SPOTs. However, it can easily access RESTful services. Thus, it is straightforward to create Clickscript building-blocks representing the two devices. This is done by adding a small JavaScript snippet to the Clickscript environment. The snippet used to integrate the temperature sensor of the Sun SPOTs is shown in Listing 3.3. The simplicity of this code illustrates well the ease of integration of smart things to Web tools and languages once they offer their services through a Web API.

```
1 //This is the Sun Spot temperature component
2 csComponentContainer.push({
```

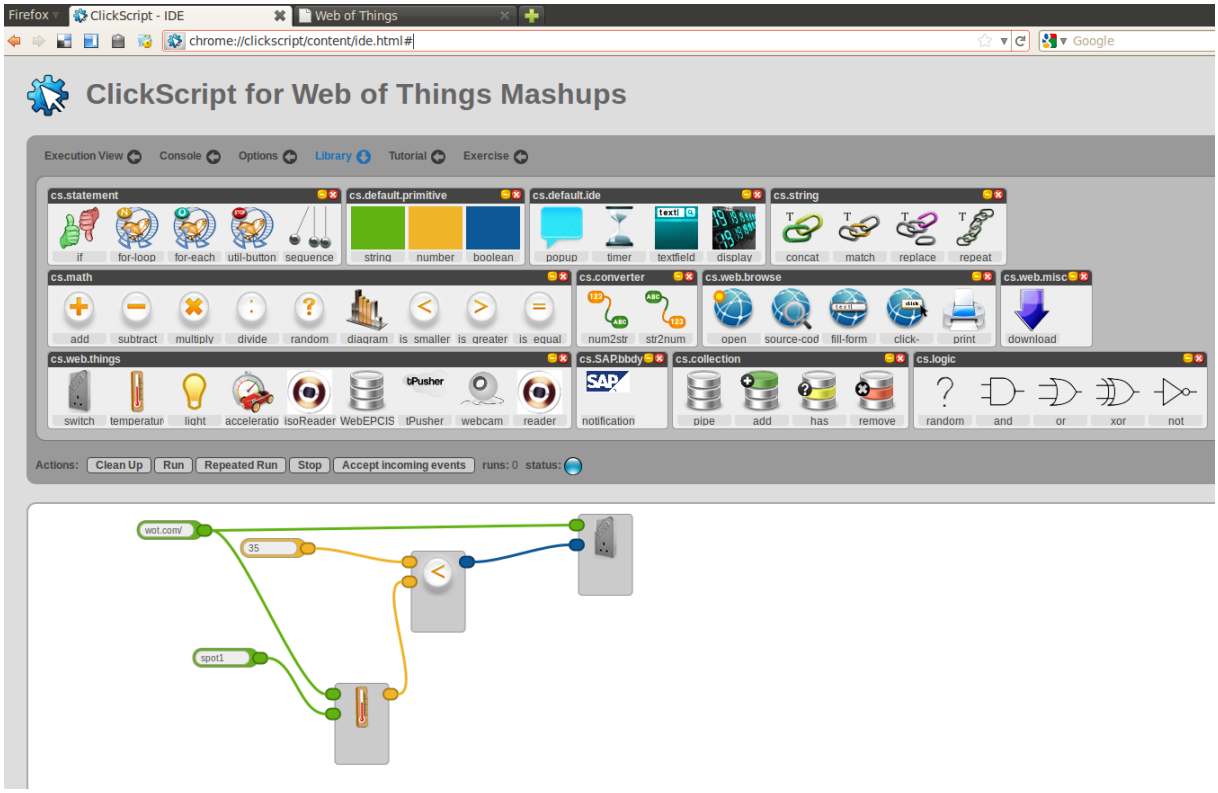


Figure 3.22: A simple Physical Mashup using the adapted version of the Clickscript Mashup Editor. The temperature sensor of the RESTful Sun SPOTs is used to actuate a RESTful Plogg.

```
3     name : "cs.web.things.temperature",
4     description : "Get the temperature from a RESTful Sun SPOT
          ",
5     inputs :
6     [
7     {
8         name: "IP",
9         type: "cs.type.String"
10    },
11    {
12        name: "Name",
13        type: "cs.type.String"
14    }
15    ],
16    outputs:
17    [
18    {
19        name: "Temperature",
20        type: "cs.type.Number"
21    }
22    ],
23    image: "web/things/temperature.png",
```

```
24      exec : function(state){
25          this.setAsync();
26          var ip = state.inputs.item(0).getValue();
27          var name = state.inputs.item(1).getValue();
28          var aurl = "http://" + ip + "/sunspots/" + name + "/
                sensors/temperature";
29          var component = this;
30          $.ajax({
31              beforeSend: function(xhrObj){
32                  xhrObj.setRequestHeader("Accept","application/
                        json");
33              },
34              url: aurl,
35              type: "GET",
36              dataType: "json",
37              success: function(result){
38                  var temp = result.resource.getters[0].value
39                  // write this to output socket, expecting a
                        number
40                  state.outputs.item(0).setValue(temp)
41                  component.finishAsync();
42              },
43              error: function(msg){
44                  alert("Error on: "+aurl);
45              }
46          });
47      }
48  });
```

Listing 3.3: JavaScript code required to integrate temperature sensor of a RESTful Sun SPOT as a Clickscript building-block.

3.4.3 Energy-Aware Mobile Mashup Editor

In this second prototype, we use the API of the Physical Mashups Framework presented in Section 2.4.5 to build a mashup editor that helps people managing their home devices and create simple rules to optimize their energy consumption. It further allows them to automate their homes in an ad-hoc, flexible manner.

Rather than creating a desktop tool such as the editor presented before, we wanted the mashup editor to be able to run on mobile phones and tablet PCs. Indeed, several studies [112] over the past few years illustrated a clear trend for people to favor the use of their mobile devices while at home, a trend that was intensified thanks to the advent of tablet PCs such as the iPad or the Android Tablets.

In particular, mobile devices have been increasingly used to control smart home environments because of their portability and ubiquity [112]. Furthermore, by implementing

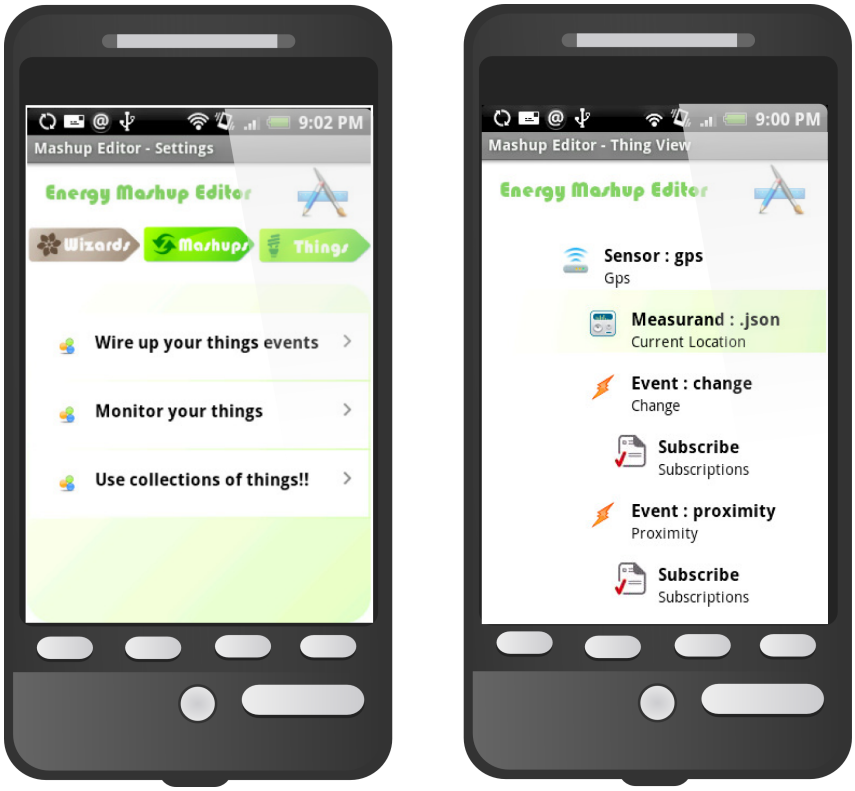


Figure 3.23: (Left) Home view of the Energy Mashup Editor. Users can choose three different wizards to create simple rules to automate their home. (Right) User interface for a smart thing once discovered by the Energy Mashup Editor through the Physical Mashups Framework. The smart thing can then be used in mashups.

the Device Accessibility Layer as described in [4] a mobile phone becomes both a control device and a RESTful smart thing offering sensing capabilities (e.g., location sensing).

Hence, we propose a mashup editor that allows managing smart homes implementing the Web of Things Architecture by creating simple rules and offering direct access to the appliances through a mobile phone interface. We present the Energy Mashup Editor which focuses on creating simple rules that help users optimize their home environment towards a better energy usage. We illustrate this by showing how the RESTful Sun SPOTs and RESTful Ploggs can be easily discovered by the framework and integrated in user-customized home automation rules. Parts of this section were published in [4, 68, 113].

Functional Overview

The home user interface of the Energy Mashup Editor is shown in Figure 3.23. The upper part offers access to three sub-interfaces: *Wizards, Mashups* and *Things* that we describe here.

Things: Discovery and Registration A click on *Things* provides access to the smart things management and discovery interface. One of the most important feature of the

Energy Mashup Editor is the ability to discover and register smart things based on their description metadata. The application supports smart things that are described according to the STM model proposed in Section 2.2.1.

The application offers two ways of registering smart things. First, a user can manually type the root URI of a smart thing, e.g., `/spot1` for a RESTful Sun SPOT. However, since typing URIs is rather cumbersome on most mobile devices, the Energy Mashup Editor also features a QR-code recognition module that can be used to extract a smart things' root URI from a barcode.

Once recognized, the URI is sent from the mobile phone to the Physical Mashups Framework where a STM Translation Service is used to extract resources and services from the retrieved smart things' description. The Physical Mashups Framework then returns the results to the mobile application where the extracted data is used to generate a new smart thing directly usable in mashups. The processed description metadata are then cached by the mobile application and accessible through the Things menu.

Figure 3.23 (right) shows the results of discovering an Android mobile phone implementing the Device Accessibility Layer and the Findability Layer of the Web of Things Architecture that we will call RESTful phone hereafter. In particular, this screen-shot shows the discovered sensor resource `/gps` which has three sub-resources. First (`/location`), it allows to retrieve the current location (*mesurand*) of the mobile phone using a polling approach. Then, two events can be subscribed to: `/change` and `/proximity` providing clients with changes, respectively proximity alerts through an HTTP Callback mechanism.

Wizards: Creating Mashups Rather than offering a box-and-pointer type of user interface which is challenging to use on devices with limited screen-size, the Energy Mashup Editor is based on the notion of wizards or assistants. In these wizards, the users are guided through a number of steps that help them creating simple rules.

The application features three types of wizards as shown in Figure 3.23 (left):

Wire Up Things Events This wizard can be used to create mashups triggered by real-world events. Once the users discovered a smart thing that offers event-based services, they can use it to create a mashup that will be activated by a push from the smart thing in question. As an example, a mashup can be triggered when a certain temperature level is recorded by a RESTful Sun SPOT. Figure 3.24 (left) shows the wizard when selecting a subscription to the proximity event of the RESTful phone. The form is generated dynamically based on the JSON message returned by the phone `location` resource. Note that the *Callback URL* parameter cannot be edited since the Physical Mashups Framework generates and manages callbacks automatically. Once subscribed to an event, the wizard will ask for selecting a triggered action, i.e., for specifying a building-block that should be used when reacting to the event. `TargetBlock`s (including previously discovered smart things) or `ProcessingBlock`s can be triggered. As an example, the `status` resource of a

Figure 3.24: Wizard to create a mashup based on information pushed by smart things (left). The form represents the parameters that have to be provided in order to subscribe to the RESTful phone GPS proximity event (right).

RESTful Plogg could be updated so that the device attached to the Plogg is turned off or on when the event is triggered.

Monitoring Things This wizard is used to pull information from a smart thing. This is especially suitable for smart things that do not support push mechanisms. The wizard first guides the user to selecting a previously discovered smart thing and selecting one of its resources. Once selected, the user is asked to specify a condition to be met to in order to trigger the execution of the next building-block as well as the polling frequency. As an example, Figure 3.24 (right) shows the form for selecting a condition on the `/gps/location` resource of the RESTful phone. Again, this form is dynamically generated using reflection [285] based on JSON message retrieved when requesting a `GET` on the resource URI. Finally, the user has to provide a `TargetBlock` or `ProcessingBlock` that should be triggered when the condition is met.

Collections of Things In order to create more complex mashups, the last wizard allows users to use collections of smart things (i.e., `RepositoryBlock`s) previously created with the Physical Mashups Framework. First, queries on these collections can be specified using the wizard. Then, the user is prompted to fill in the parameters of a `VisualizationBlock` where the queried data will be dynamically visualized using the Google Visualization API [238] (see Section 2.4.6).

Use-Cases

The Energy Mashup Editor was tested in a formative evaluation. While these tests clearly do not demonstrate the usability of the system in real home environments and with end-users, their purpose is to illustrate the typical use-cases that we envisioned when creating the editor. Furthermore, it gave us feedback and ideas for future directions for home automation mashups.

Methodology The evaluation was conducted in a laboratory at ETH. Five participants, 4 of them computer-science students, gave about 40 minutes of their time to build physical mashups. Five smart things were provided:

1. An Android-based mobile phone (HTC Hero) running the Energy Mashup Editor and used to create the physical mashups.
2. A virtual mobile phone which resources where accessible through a RESTful Web API and which location could be updated manually during the study, in a Wizard-of-Oz manner.
3. A fan table heater attached to a Plogg and Web-enabled through the Ploggs Smart Gateway.
4. A desktop lamp attached to another Plogg, Web-enabled through the same gateway as the heater.
5. A RESTful Sun SPOT implemented using the Sun SPOTs Smart Gateway.

Each of these devices was attached a QR-code containing the root URI of the device it was attached to. The five participants were given a small introduction on how to use the mobile application and were provided with four use-cases they should implement.

Tasks The exhaustive description of the use-cases is provided in [4]. Here we describe two of them to illustrate a concrete use of the Energy Mashup Editor.

The first use-case was testing the use of event-based mashups. The participants were asked to create a mashup that would save energy by starting the heater only when they were approaching home.

The task was solved by going through the following steps:

1. Use the `Things` menu and scan the QR-code of the mobile phone to discover it.
2. Similarly, scan the heater (connected to a Plogg).
3. Use the `Wire Up Things Events` wizard:
   (a) Select the discovered mobile phone
   (b) Browse to the GPS → Proximity → Subscription resource

(c) Subscribe to the resource and specify a radius (see Figure 3.24)

(d) Select the heater as the target building-block

(e) Browse to the Power Status → Status resources

(f) Select the desired value (on)

(g) Save and run the mashup (which exports it and runs it on the Physical Mashups Framework)

Technically, this mashup subscribes the Physical Mashups Framework to the proximity resource of the phone GPS. Whenever the phone is located within a certain radius (i.e., coming home) an HTTP `POST` request is sent to the Physical Mashups Framework. This event triggers the mashup which `POST`s "`on`" to the Plogg Smart Gateway, resulting in starting the attached heater.

In the second task, participants were asked to create another energy-saving rule that turns off the lights when the natural light level is high enough. The task was solved by going through the following steps:

1. Use the `Things` menu and scan the QR-code of the Sun SPOT
2. Similarly, scan the lamp (connected to a Plogg)
3. Select the `Monitoring Things` wizard:
   (a) Select the discovered Sun SPOT
   (b) Browse to the TestSpot → Light sensor resource
   (c) Provide a monitoring condition (e.g., light value > 200)
   (d) Select the lamp as the target building-block
   (e) Browse to the Power Status → Status resources
   (f) Select the desired value (off)
   (g) Save and run the mashup (which exports it and runs it on the Physical Mashups Framework)

In this case the Physical Mashups Framework will poll (`GET`) the light resource of the Sun SPOT on a regular basis and trigger a `PUT` "`status=on`" on the lamp (connected to a Plogg), resulting in cutting the power of the attached lamp.

The participants built two other use-cases in which the goal was to create a visualization of energy consumption data as well as to send an XMPP or Twitter message whenever the home temperature was reaching 30 degrees Celsius.

Formative Users Feedback Overall the users liked the idea of being able to re-wire their physical environment. For the 5 participants the idea of using a mobile phone was appealing because they always have it with them and are used to its interface. They

liked being able to scan objects but had some difficulties getting them recognized and suggested the use of other types of tags, more straightforward to read with a mobile phone such as NFC (Near Field Communication) tags. This confirms our study [201] where we evaluated several object identification techniques and concluded that NFC was the fastest and easiest to operate for end-users.

Besides a number of UI improvements [4], they suggested concrete use-cases. Two examples are the creation of an energy monitor that can compute and present the energy costs for a particular device. Another use-case was the control, configuration and visualization of data from laboratory instruments such as scales, preventing work-interrupts for getting this data from a desktop computer.

3.5 Related Work

Because of their ubiquity, Wireless Sensor and Actuator Networks are a the center of many recent Internet of Things applications. Hence, reducing the complexity of developing applications on these platforms [6] has interested several researchers over the last decade [65, 110].

However, WSN application development still requires specific skills and are time consuming [146]. Moreover, for each new deployment, a large amount of work must be devoted to re-implement basic functions and application-specific user interfaces, which is a waste of resources that could be used by developers to focus on the application logic. Ideally, developers should be able to quickly build applications on top of WSNs.

The advent of IP technologies for WSNs [40, 99] combined with the creation of global consortia such as the IPSO alliance fostered the idea of using the Internet as an integration bus to facilitate application development. In this space, researchers proposed and evaluated the use of WS-* services [92, 157, 103], in the SOCRADES project, we joined this research and explored the integration of WSNs to enterprise applications [36, 77].

Further researchers questioned the suitability of WS-* services because of their important needs in terms of network, processing and storage resources [214]. We shared these concerns and further argued that from a functional point of view an integration to true Web protocols would be highly relevant [79]. The idea of using Web protocols and in particular RESTful architectures specifically for WSNs had been proposed by Drytkiewicz et al. [39] and further explored by Luckenbach et al. [126] building upon early conceptual work from Kindberg, Barton et al. [110, 14]. These pioneering projects were showing the way towards Web integration of WSNs but they implemented the constraints of RESTful architectures only partially: They do not implement the connectedness constraint nor do they leverage mechanisms such as content-negotiation. This can be explained by the very prototypical state of the embedded Web in these early years.

Recent research benefits from the new perspectives of IPv6 for WSNs over IEEE 802.15.4 (also known as 6loWPAN) [99] and the development of very small-footprint Web servers

implementing full HTTP 1.1 stacks [42]. As conceptually suggested in [209, 185] we joined further recent projects in exploring a systematic implementation of REST as presented in this chapter and published in [79, 76, 81]. Building upon our work, Schor et al. proposed a native, application-gateway less, implementation of a RESTful architecture on WSN nodes [173].

While we share with these works the lower layers of the Web of Things Architecture (i.e., closely related to the Device Accessibility Layer we proposed), we take a more holistic approach and look at the integration issues from an application layer point of view. We are interested in the overall picture of integrating WSNs to the Web. We focus on the developer view-point and illustrate the benefits and architectures that can be leveraged once the sensor nodes are part of the Web. In this space, the sMAP [33] project, conducted in parallel to our work, looks at the bigger picture of WSN Web integration but does not address cross-cutting layers such as the Composition Layer or Sharing Layer.

WSNs have been used to monitor energy consumption in several projects [122, 179]. Similarly, the results of projects cited above could also be applied to the specifics of energy-aware WSNs. However, specifically looking at the use-case of smart meters, several research projects proposed the use of the Internet to facilitate their integration. Jiang et al. [105] looked at integrating custom smart meters directly to IP networks. In [173] the authors propose a user interface for creating simple energy-awareness rules on top of their RESTful architecture (e.g., switch off the light automatically). Our contribution here differs since we took the approach of adapting an off-the-shelf smart meter WSN platform (through the use of a Smart Gateway) and demonstrated the benefits by building Web applications that help end-users monitor and control their electricity consumption.

3.6 Discussion and Summary

In this chapter, we presented an implementation of the Web of Things Architecture for two different sensor networking platforms. First, with the Sun SPOTs, we illustrated how the Web of Things Architecture can be leveraged as a valid candidate for a WSN common application platform. Certainly, the presented approach is not the universal solution for every problem. Scenarios with specific requirements such as high performance real-time communication still benefit from tightly coupled systems based on traditional RPC-based approaches. However, for less constrained applications that are more oriented towards end-users and prototyping where ad-hoc interaction and serendipitous reuse are required, Web standards can simplify integration of WSN data and functionality.

In applications where the raw performance and battery life-time are critical, for example when nodes run on battery in large-scale and long-lived deployments, optimized protocols that minimize network connection and latency will remain the best option. However, when devices are connected to a power source and when sub-second latency can be tolerated, then the advantages of HTTP clearly outweigh the loss in performance and latency.

Based on our experience, we suggest that for many applications the drawbacks of Web architectures are largely offset by the simplification of the design, integration, and deployment processes [81]. Although HTTP introduces a communication overhead and increases average response latency, recent research has shown that this overhead is still small enough to enable most WSN scenarios [214, 157, 193]. Our study of end-to-end HTTP and Smart Gateway mediated implementation of the Sun SPOTs confirm these results and suggests that for sub-seconds WSN use-cases both approaches are useable.

Furthermore, we presented an easy to use WSN-based system for energy savings in home and office environments based on the Web of Things Architecture. Our contribution on the architecture level is twofold. Firstly, we extended the capabilities of the Ploggs by providing a Smart Gateway that discovers and integrates the available physical devices, and that provides a RESTful Web API that can be exploited by third-party applications. Secondly, we demonstrated how the approach eases application development and fosters interoperability of the system by showing how it was used for developing energy applications on Web-based and Mobile platforms.

Finally, beyond integration at the device level, we demonstrated how WSNs can directly benefit from the other layers of the Web of Things Architecture: the Findability Layer helps searching for WSN nodes directly on the Web, the Sharing Layer architecture offers an easy and straightforward mechanism to share all kinds of sensor related resources. With the Composition Layer we show that similarly to how Web 2.0 mashups have significantly lowered the entry barrier for the development of Web applications, these techniques and protocols can be used to lower the entry barrier for creating end-user applications that are using WSNs.

Chapter 4

Resource-Oriented RFID Networks

Contents

In Chapter 3, we introduced how wireless sensor networks and embedded computers can be integrated to the Web of Things. In this chapter, we look at the integration of every-day objects through Auto-ID (Automatic IDentification) [150]. Swartz identified 6 types of core Auto-ID techniques [186]: one and two-dimensional barcodes, OCR (Optical Character Recognition), magnetic stripes, automatic speech identification and RFID (Radio Frequency IDentification). More recently, new techniques such as image recognition have been added to the available technologies [158, 100].

Auto-ID technologies have long been used in ubiquitous computing to create a link between the physical and the digital world [203]. As an example researchers have been using one dimensional [3] or two dimensional barcodes [164, 23, 167] to create a link between every-day objects and mobile phones.

Amongst the Auto-ID technologies, RFID is particular since it allows to make every-day objects, that do not have intrinsic computing and communication capabilities, part of a wireless network. Moreover, in the EPC Network (Electronic Product Code) [169], tagged objects become part of an *Internet of Things*. Indeed, the RFID standards community has developed a number of wireless interfaces and software standards to provide interoperability across RFID deployments. The EPC Network is a set of standards [229] established by industrial key players as well as research institutions towards a uniform platform for tracking and discovering RFID tagged objects and goods [54, 55]. Fourteen standards are currently forming the EPC Network, addressing every step required from encoding data on RFID tags to reading them and sharing their traces.

From an industrial and real-world view-point, the EPC Network is an ideal IoT system. Indeed, most of its components, from the tags to the IT backend, are standardized which should considerably facilitate deployments. However, the EPC Network, as many other IoT infrastructures [161, 25], is hard and expensive to deploy, maintain and develop upon. Hence, the adoption of the software standards within the EPC Network has been slower than projected (e.g., [124]). Indeed, the deployment of RFID applications that implement the EPC Network standards often remains complex and cost-intensive mostly because they involve rather large and heterogeneous distributed systems. As a consequence, these systems are often only suitable for big corporations and large implementations and do not fit the limited resources of small to mid-size businesses and small scale applications both in terms of required skill-set and costs [172].

While there is most likely no universal solution to these problems, the success of the Web in bringing complex, distributed and heterogeneous systems together through the use of simple design patterns appears as a viable approach to address these challenges. Hence, in this chapter, we design and implement a Web of Things Architecture for the EPC Network. We discuss the pain-points of RFID applications that have made deployments challenging and describe how they can be addressed using solutions inspired from the Web and the Web of Things Architecture. The resulting component architecture is shown in Figure 4.1. It basically offers a cloud infrastructure featuring a number of adapters that make

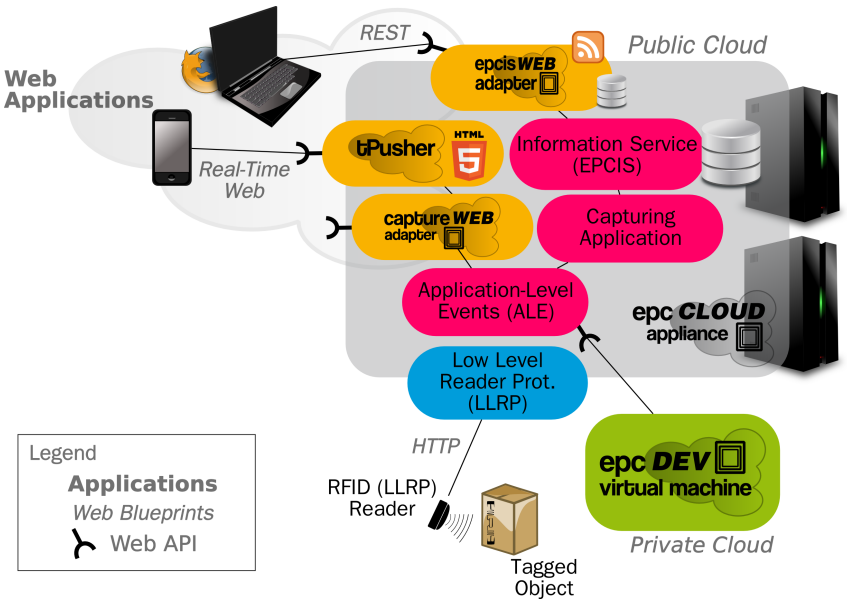


Figure 4.1: Overview of the EPC Cloud component architecture. Web components are added to the existing EPC Network components and packaged into virtual machines that can be deployed in public or private clouds. Web and mobile applications can be built on top of the new components' Web APIs.

standard EPC Network components integrated to the Web. On top of this infrastructure RFID applications can be easily created using standard Web languages and tools as shown in the leftmost part of Figure 4.1. This illustrates how, using RFID and leveraging our Web implementation of the EPC Network, objects without computing or communication capabilities can become part of the Web of Things simply by attaching or embedding RFID tags into them.

This chapter is based on work published in [74, 72, 75, 147] and is structured as follows. We begin by briefly describing the EPC Network, focusing on the standards that are especially important for our use-cases. We then look at three important pain points of EPC Network deployments. Then, for each pain point we propose a solution using the Web of Things Architecture and its components. First, we illustrate how cloud computing can foster the adoption of software implementations of the EPC standards. Then, we discuss how the Device Accessibility Layer and a REST architecture can be used to enable an easy access to EPC information systems and RFID readers. Finally, we demonstrate how the Web of Things Architecture enables developers and end-users to create physical mashups using data and devices of the EPC Network.

4.1 The EPC Network in a Nutshell

The EPC Network is an architectural framework for RFID applications based on 14 standards as show in Figure 4.3. These standards are used by several actors at several steps of the product life-cycle as illustrated in Figure 4.2. A comprehensive description

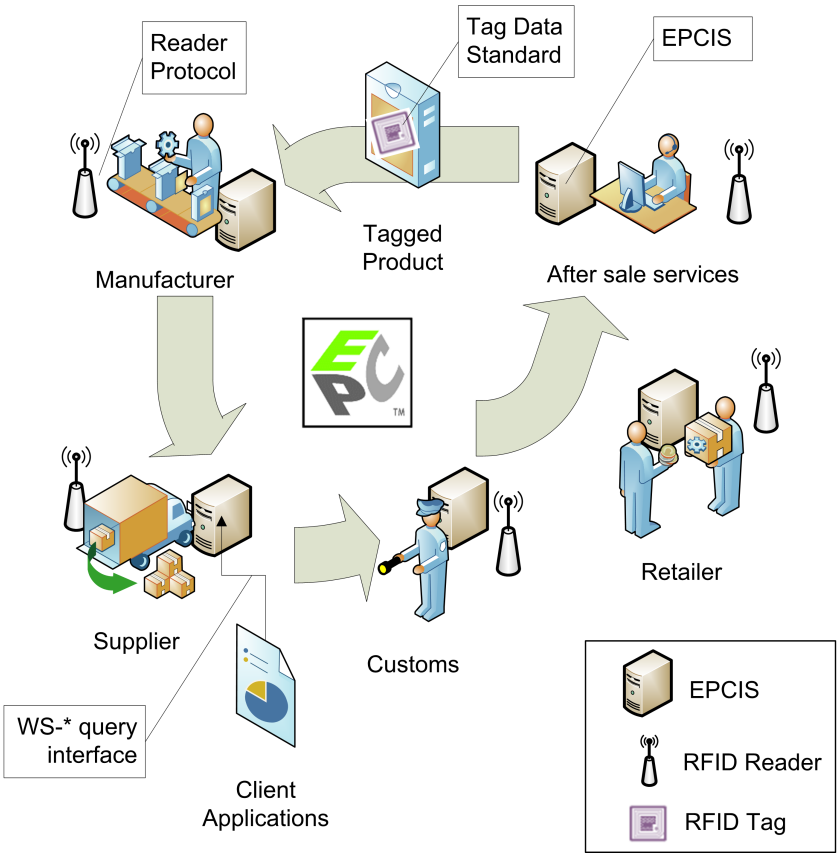


Figure 4.2: Simplified overview of the EPC Network. The standards help tracking and tracing a product through its life-cycle.

of how these standards are put together to create a track and trace network is provided in [55] or [189]. Here, we focus on a short description of the most relevant standards in the context of this chapter.

4.1.1 Identifying EPC Numbers

The Tag Data Standard [46] defines what an EPC number is and how it is encoded on the tags themselves. An EPC is a world wide unique number. The TDS supports nine encoding schemes for EPC numbers and addresses compatibility issues between them. These encodings specify numbers of a size between 96 and 202 bits which means that they offer at least 2^{96} unique identifiers. As a consequence, rather than identifying a product class, like the common barcode standards (UPC and EAN-13) do, it can be used to identify the instance of a product. An EPC number basically encodes three types of information: the manufacturer, the product class and a serial number. As an example in the following tag (represented in its URI form):
`urn:epc:id:gid:2808.64085.88828`
`2808` is the manufacturer ID, `64085` represents the type of product and `88828` an instance of the product.

EPC numbers can be potentially written onto any support. However, in the EPC Network, these number are commonly written on RFID tags, called EPC tags. The wireless communication used by these EPC tags is specified in the Tag Protocol standards. In particular the Class 1 Generation 2 UHF Air Interface Protocol Standard (also known as EPC Gen 2) [189] defines the current state of the art wireless protocol for tags to communicate over the 860 - 960 MHz frequency range.

4.1.2 Standards for Capturing EPC Events

The LLRP (Low Level Reader Protocol) standard [47] specifies how to communicate and configure standard RFID readers. Through the LLRP protocol clients can configure basic filtering and gather raw RFID data directly from the readers.

On top of the LLRP protocol, the ALE (Application Level Events) standard [45] specifies an interface that can be used by applications to obtain processed EPC events. The ALE offers processing in terms of filtering and aggregation capabilities.

On top of the ALE lies a custom components called Capturing Application [189]. Such an application has to be designed specifically for each deployment and basically maps the EPC events coming out of the ALE to events that are relevant in a business context, adding for instance the business steps or locations to events.

4.1.3 Sharing EPC Events

One of the primary goals of the EPC Network is to allow sharing observed EPC events. Thus, the network specifies a standardized server-side EPCIS [44], in charge of managing and offering access to traces of EPC events. Once added a business context by the Capturing Application, events are stored in an EPCIS together with contextual data. In particular, these data deliver information about:

- The *what*: what tagged products (EPCs) were read.
- The *when*: at what time the products were read.
- The *where*: where the products were read, in terms of Business Location (e.g., "Floor B").
- The *who*: what readers (Read Point) recorded this trace.
- The *which*: what was the business context (Business Step) recording the trace (e.g., "Shipping").

The goal of the EPCIS is to store these data to allow creating a global network where participants can gain a shared view of these EPC traces. As such, the EPCIS deals with historical data, allowing, for example, participants in a supply chain to share the business data produced by their EPC-tagged objects.

Technically speaking, a standard EPCIS is an application that offers three core features to client applications:

1. First it offers a way to capture, i.e., persist, EPC events.
2. Then, it offers an interface to query for EPC events.
3. Finally, it allows to subscribe to queries so that client applications can be informed whenever the result of a query changes.

There exist several concrete implementations of EPCISs on the market. Most of them are delivered by big software vendors such as IBM or SAP. However, the Fosstrak [56, 57] project offers a comprehensive, Java-based, open-source implementation of the EPCIS standard.

4.2 A Cloud-Based Virtual Infrastructure for the EPC Network

Bringing real-world data and smart things closer to the Web also facilitate the use of modern distributed architectures. In this section, we illustrate how virtualization and in particular cloud computing can greatly simplify the deployment and maintenance of standard-based RFID networks.

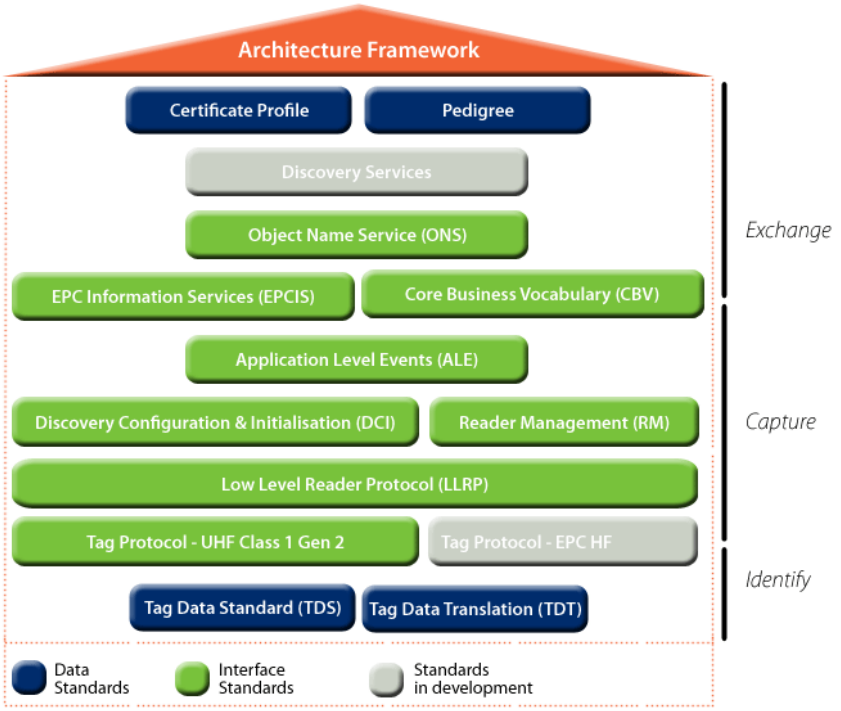


Figure 4.3: EPC Network Architectural framework composed of 14 standards (Source [189]).

4.2.1 Pain-Point: Complex Backend Deployment and Maintenance

Real-world, industrial IoT systems often encompass several relatively complex standards and their respective implementation is often scattered amongst a number of different software components [161, 25]. The EPC Network is no exception. Across all vendors of EPC Network stacks, the standards are implemented in several different software components often sold separately to form an EPC software stack. As an example the Fosstrak [57] open-source project [234] is implementing most of the EPC standards and requires the installation of 9 different software components in order to be able to run a end-to-end use-case from RFID tags to an IT system.

Additionally, a full Fosstrak installation also requires a compatible Java SDK, Apache Maven, a full MySQL database and an Apache Tomcat server, summing up the number of required software components to 13. As a consequence, a full EPC software stack is rather complex to install and deploy and *often requires software experts*, which becomes problematic especially when considering businesses for which IT is not a core concern (e.g., actors of the supply chain) or smaller businesses. The complexity is further increased by the maintenance work required by a number of different components and their respective updates and patches cycles. Hence, deploying and *maintaining IoT systems is time consuming* and accounts for a greater part of the system's overall software costs similarly to other IT systems [12].

Furthermore, the software components often need to be deployed on application servers running on dedicated hardware. For the Fosstrak stack, a Java Application Server (or

a least a servlet container such as Tomcat) is required and needs to be configured on a hardware server to handle the appropriate load and accesses. Similarly, the IoT embedded devices (e.g., RFID readers, sensor nodes, etc.) need to be deployed, maintained and configured. This induces *significant hardware costs and the need for hardware experts.*

4.2.2 Virtualization Blueprint

Reducing complex software installation is one value propositions [115] of virtualization platforms such as VMWare [280] or the open-source Virtual Box [279]. With these platforms, software stack can be installed once in a virtualized OS (operating system) called virtual machine or guest OS, and then shared to be deployed within minutes on any supported host machine running the virtualization platform. This significantly reduces the installation costs and required-skills.

In the IoT space, this benefit has been identified and is increasingly used in platforms such as the Instant Contiki virtual machine [225] which offers a complete development environment for WSNs (Wireless Sensor and Actuator Networks) ready to use within minutes [162]. However, the EPC Network still lacks such solutions. Hence, we virtualized an EPC software stack. The EPC Dev Virtual machine combines a Linux Ubuntu Operating System, with an Eclipse IDE (Integrated Development Environment), a source repository (Maven), as well as an Apache Tomcat container in which we deployed and configured the 9 remaining software components of Fosstrak. This means that the virtual machine can be used both as a development environment or as as a test server instance of the EPC software stack if installed on an appropriate server machine.

This cuts down the installation time of a full EPC software stack from several hours or days to a few minutes. It further fosters quick evaluation of a complete EPC software stack which can be of great help when assessing different implementations, developing proof of concept prototypes or enhancements of the EPC software stack.

4.2.3 Cloud Computing: Utility Computing Blueprint

While Virtualization significantly reduces the installation time, it does not solve the other issues of IoT deployments: software and hardware maintenance costs. However, recent developments in the Web 2.0 and especially the trend towards providing services on the Web rather than simply Web-pages, have materialized into a convergence of virtualization technologies and the distributed Web, leading to Cloud Computing.

Cloud Computing can take several forms under the umbrella of two big groups. *Private Clouds* are basically virtualized environments running locally as described in the previous section. *Public Clouds* are, on the other hand, virtualized environments running on remote machines. A Public Cloud can take many forms [217], in its *Utility Computing* form it basically proposes to further push the notion of virtualization by making the hardware

on which virtual machines run available as a virtual resource pool fully accessible, on-demand, on the Web. Amazon Web Services (AWS) [219] pioneered the space of Utility Computing followed by many others such as IBM, Microsoft, Rackspace and VMWare.

Recently, Cloud Computing has been increasingly used in conjunction with WSNs [25] as a way to reduce operative complexity. One of the important benefits of Cloud Computing is the fact that it allows businesses with limited resources (both financial and in terms of staff) to run an IT infrastructure corresponding to their needs and scale [184].

We experimentally applied the Utility Computing blueprint to the EPC software stack using the AWS platform and in particular the EC2 service. Amazon EC2 allows the creation and management of virtual machines (Amazon Machine Images, or AMIs) that can then be deployed on demand onto a pool of machines hosted, managed and configured by Amazon. We created a server-side AMI called EPC Cloud Appliance, based on Linux Ubuntu Public Cloud edition [276] and containing the 13 software components required by a full installation of Fosstrak as well as the three additional Web adapters we designed and will present in the following sections. A component view of the full appliance is shown in the upper-right part of Figure 4.1.

This concretely means that any company or research institution willing to deploy an EPC software stack can simply log onto AWS, look for the EPC Cloud AMI and select the type and number of remote servers it should be deployed on. Once the virtual servers are running (which typically takes less than 5 minutes), an RFID reader can be connected. If the reader does not offer a Web-management interface or a default configuration, the Fosstrak LLRP Commander and its Eclipse-based UI are available in the EPC Dev Virtual machine and can be used for configuring it. Then, the readers are described by accessing the configuration offered in the Web UI of the EPC Cloud Appliance. Once this is done, the cloud instance will contact the reader and start recording the EPC events.

A direct benefit of the approach is that the server-side hardware maintenance is delegated to the cloud provider which is often more cost-efficient for smaller businesses [217]. Furthermore it also offers better scaling capabilities as the company using the EPC Cloud AMI can deploy additional and more powerful instances within a few clicks from the Web front-end (or Web API) of AWS and will be charged only for the resources it actually uses.

4.3 Device Accessibility Layer

In this section we illustrate how the architecture and patterns proposed in the Device Accessibility Layer can be applied to the EPC Network and explain how this approach helps reducing the complexity of developing applications on top of the EPC Network.

4.3.1 Pain-Point: Complicated Applications Developments

The idea behind most commercial IoT deployments is the integration of real-world data to business systems or end-consumer applications. This requires to interface existing or new applications with the IoT infrastructure. Thanks to the recent advent of smart phones, companies are also increasingly willing to create mobile applications using IoT deployments.

In the case of the EPC network, the application integration point is the EPCIS standard. While the EPCIS provides a simple and lightweight HTTP interface for recording EPC events, its query interface is a standardized WS-* interface. In Section 2.5 we discussed and evaluated the importance of creating simples and easy to use APIs in order to foster public innovation. We concluded that WS-* applications have advantages such as sophisticated security features but are also complex systems with high entry barriers and require developer expertise in the domain which is often an issue when considering small to mid-size businesses [172]. Moreover, we showed that WS-* are often not well adapted to more light-weight and ad-hoc application scenarios such as mobile or Web applications.

This currently limits the scope of applications that can be built using EPC data. Indeed, track and trace applications are also relevant beyond the desktop. As an example, providing an out-of-the-box mobile access to EPC events is beneficial for many users such as mobile workers. Similarly, providing direct access to RFID traces to sensor and actuator networks enables those to react to RFID events. Finally, allowing light-weight Web applications (e.g., HTML, JavaScript, PHP, etc.) to directly access these data would enable the vast community of Web developers to create innovative applications using RFID traces.

To enable this type of applications, we propose to transform the EPC Network into a RESTful architecture and further introduce the notion of real-time Web, as described in the Device Accessibility Layer of the Web of Things Architecture. We first propose a RESTful architecture that offers a Web API complementing the standard WS-* interface of the EPCIS. Finally, we illustrate how the real-time Web can be used to push RFID events from RFID readers to the Web.

4.3.2 EPCIS Webadapter

In this section we describe the core architecture supporting a large-scale Web-enabling of the EPCIS features taking a top-down approach [74].

System Architecture

As mentioned before, in the EPCIS standard, most features are accessible through a WS-* interface. To specify the architecture of the EPCIS Webadapter we systematically

took the WS-* features listed in the standard [44] and applied the properties of Resource Oriented Architectures as described in the Device Accessibility Layer.

Addressability and Connectedness We first *identify the resources* an EPCIS should be composed of, i.e., we identify the actors of the system which are worth being uniquely addressed and linked to. We then make them *addressable* and inter-link the resources.

Looking at the EPCIS standard, we can extract a dozen resources. We focus here on the four main types:

1. Locations (called *Business locations* in the EPCIS standard): those are locations where events can occur, e.g.,:"C Floor, Building B72".
2. Readers, called *ReadPoints* in the standard: which are RFID readers registered in the EPCIS. Just as Business Locations, readers are usually represented as URIs: e.g., `urn:br:maxhavelaar:natal:shipyear:incoming` but can also be represented using free-form strings, e.g.,: *Reader Store Checkout*
3. Events: which are observations of RFID tags, at a Business Location by a specific reader at a particular time.
4. EPCs: which are Electronic Product Codes identifying products (e.g., `urn:epc:id:sgtin:618018.820712.2001`), types of products (e.g., `urn:epc:id:sgtin:618018.820712.*`) or companies (e.g., `urn:epc:id:sgtin:618018.*`).

We first define a hierarchical clustering of resources based on the following URI template: `/location/{businessLocation}/reader/{readPoint}/time/{eventTime}/event`. More concretely, this means that the users begin by accessing the Location resources. Accessing the URI `/location` with the `GET` method retrieves a list of all Locations currently registered in the EPCIS. From there, clients can navigate to a particular Location where they will find a list of all Readers at this place. From the Readers clients get access to Time resources which root is listing all the Times at which Events occurred. By selecting a Time the client finally accesses a list of Events.

Each event contains information like its type, event time, Business Location, EPCs, etc. If a client is only interested about one specific field of an Event, he can get this information by adding the desired information name as sub-path of the Event URI. For example `<EVENT-URI>/epcs` lists only all the EPCs that were part of that Event. The resulting tree structure is shown in Figure 4.4, and a sample Event in Figure 4.5.

Furthermore, to fulfill the connectedness constraint of REST architectures, all resources should be discoverable by browsing to facilitate the integration with the Web. Just as one can browse Web pages, one should be able to find RFID tagged objects and their traces by browsing. Each representation of resources should contain links to relevant resources such as parents, descendants or simply related resources.

Hence, to ensure the connectedness of the EPCIS Webadapter, each resource in the tree links to the resources below or to related resources. The links allow users to browse

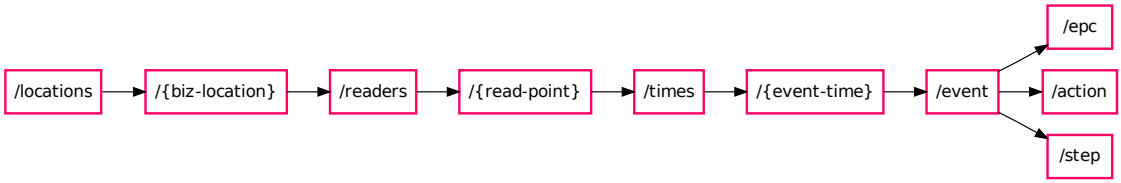


Figure 4.4: Hierarchical representation of the most important browsable EPCIS resources.

completely through the EPCIS Webadapter where links act as the motor. Every available action is deduced by the set of links included. This way, people can directly explore the EPCIS from any Web browser, simply by clicking on hyperlinks and without requiring any prior knowledge of the EPCIS standard.

To ensure that the browsable EPCIS interface does not become too complicated, we limit the number of available resources and parameters. For more complex queries we provide a second interface for which we map the EPCIS WS-* query interface to uniquely identifiable URIs. Each query parameter can be encoded and combined as a URI query parameter according to the following template `/eventquery/result?param1=value1&...&paramN=valueN`. Query parameters restrict the deduced result set of matching RFID events. The EPCIS Webadapter supports the building of such URIs with the help of an HTML form. If for example a product manager from Max Havelaar is interested in the events that were produced in Palmas, the following URI lists all events that occurred at this business location: `/eventquery/result?location=urn:br:maxhavelaar:palmas:productionsite`. To further limit possibly very long search results, the query URI can be more specific. The manager might be interested only about what happened on that production site on December 18th 2011, which corresponds to the following URI: `/eventquery/result?location=urn:br:maxhavelaar:palmas:productionsite&time=2009-11-04T00:00:00.000Z,2011-18-12T23:59:59.000Z` Figure 4.5 illustrates the HTML representation of this resource.

To keep the full connectedness of the EPCIS Webadapter, both the browsable and the query interface are inter-linked. For example, the EPC
`urn:epc:id:sgtin:0057000.123430.2025`
included in the event of Figure 4.5, is also a link to the query which asks the EPCIS for all events that contain this EPC.

By implementing the addressability property we allow greater interaction with EPCIS data on the Web. As an example, since queries are now encapsulated in URIs, we can simply bookmark them, exchange them in emails and consume them from JavaScript applications. Furthermore, by implementing the connectedness property we enable users to discover the EPCIS content in a simple but yet powerful manner.

RESTful Path ID	ID	*Unique Path ID to represent the Event*
Event Type	ObjectEvent	
Event Time	2009-11-04T10:21:39.000Z	
Time Zone Offset	+00:00	
Record Time	2010-02-26T15:04:59.000Z	
Business Location	urn:br:maxhavelaar:palmas:productionsite	
Read Point	urn:br:maxhavelaar:palmas:productionsite:outgoing	
Business Step	urn:epcglobal:epcis:bizstep:fmcg:shipping	
Action	OBSERVE	
EPC List		
EPC	urn:epc:id:sgtin:0057000.123430.2025	Company Prefix: 0057000 / Item Reference: 123430 / Serial Number: 2025 — *Serialized Global Trade Item Number*
EPC	urn:epc:id:sgtin:0057000.123430.2026	Company Prefix: 0057000 / Item Reference: 123430 / Serial Number: 2026 — *Serialized Global Trade Item Number*

Figure 4.5: HTML representation of an EPC event as rendered by a Web browser. Every entry is also a link to the sub-resources.

Uniform Interface Finally, in a ROA, the resources and their services should be accessible using a standard interface defining the mechanisms of interaction. We particularly focus on two aspects of the uniform interface here: the representation of resources, and the communication of errors.

Multiple Representation Formats A resource is representation agnostic and hence should offer several representations. The EPCIS Webadapter supports multiple output formats to represent a resource. Each resource first offers an HTML representation as shown in Figure 4.5 which is used by default for Web browser clients.

In addition to the HTML representation, each resource has also an XML and a JSON representation, which all contain the same information. The XML representation complies with the EPCIS standard and is intended to be used mainly for business integration. The JSON representation can be directly translated to JavaScript objects and is thus intended for mashups, mobile applications or embedded computers.

As introduced in the Device Accessibility Layer, the choice of what representation to use is left to clients who can request it through the HTTP *content negotiation* mechanism. Since content negotiation is built into the uniform interface, clients and servers have agreed-upon ways to exchange information about available resource representations, and the negotiation allows clients and servers to choose the representation that is the best fit for a given scenario.

A typical content-negotiation interaction with the EPCIS Webadapter looks as follows: the client begins with a `GET` request on `http://.../location`. It also sets the `Accept` header of the HTTP request to a weighted list of media types it can understand, for

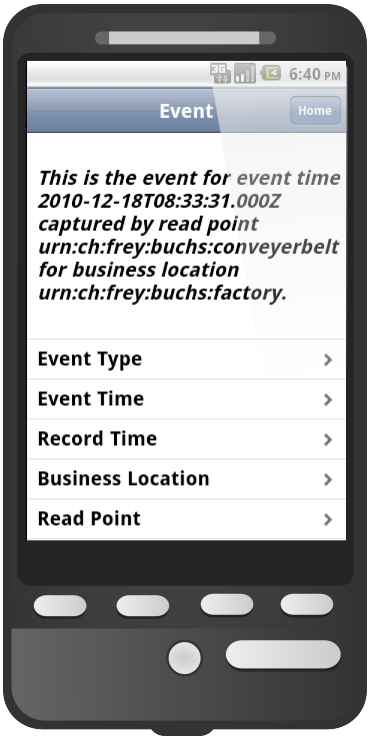


Figure 4.6: Customized HTML representation of an EPC event as appearing in the mobile Web browser of an Android Mobile phone.

example to: `application/json`, `application/xml;q=0.5`. The EPCIS Webadapter then tries to serve the best possible format it knows about and describes it in the `Content-Type` of the HTTP response. In this case it will serve the results in the JSON format as the client prefers it over XML (q=0.5).

Representations are also a very straightforward and powerful way of providing variations of formats that are adapted to the clients' platforms. As an example, Figure 4.6 is the HTML Web page a mobile user of the EPCIS Webadapter is served. Although it has the look and feel of a native mobile application, it is actually a simple HTML Web page featuring special CSS style-sheets and JavaScript code (based on the iUI framework [245]) that dynamically adapts the content to fit the form factor and interaction paradigms of a mobile phone.

When a client negotiates an HTML representation, the EPCIS Webadapter also detects whether the client should be served an HTML representation adapted to a mobile screen or simply to a desktop screen. For this, it uses the `User-Agent` standard HTTP header which contains a string describing the browser (or any other client) that was used for the request.

Error Codes The EPCIS standard [44] defines a number of exceptions that can occur while interacting with an EPCIS. HTTP offers a standard and universal way of communicating errors to clients by means of *status codes*. Thus, to enable clients and especially applications to make use of the exceptions defined in the EPCIS specification, the EPCIS

EPCIS Standard Exception	HTTP Status Code	HTTP Semantics
`SecurityException`	401	Unauthorized
`QueryParameter`	400	Bad request
`QueryTooLarge`	400	Bad request
`QueryTooComplex`	400	Bad request
`InvalidURI`	416	Requested range not satisfiable
`SubscriptionControls`	400	Bad request
`NoSuchName`	400	Bad request
`NoSuchSubscription`	400	Bad request
`DuplicateSubscription`	409	Conflict
`SubscribeNotPermitted`	401	Unauthorized
`ImplementationException`	501	Not Implemented

Table 4.1: Mapping EPCIS standard exceptions to standard HTTP status code. All other exceptions are mapped to *500: Internal Server Error*.

Webadapter maps the exceptions to HTTP status codes. As an example, the EPCIS exception: `SecurityException` is mapped to the HTTP status code: `401`, which semantics is `Unauthorized`, and returned to the client alongside with a message that describes what happened in a user-friendly textual way. Table 4.1 provides an overview of the exceptions to status-codes mappings.

Web-Enabling the Subscriptions

As mentioned before, standard EPCISs also offers an interface to subscribe to RFID events. Through a WS-* operation, clients can send a query along with an endpoint (i.e., a URI) and subscribe for updates. Every time the result of the query changes, an XML packet containing the new results is sent to the endpoint. While this mechanism is practical, it requires for clients to run a server with a tailored Web applications that listens to the endpoint and thus cannot be used by all users or cannot be directly integrated to a Web browser. To improve this, the EPCIS Webadapter offers a RESTful subscription interface and a Web feed of the updates.

For this, in the Device Accessibility Layer, we suggested the use of the Atom Syndication Format. Hence, in the EPCIS Webadapter, we propose an alternative interface for subscribing to RFID events using Atom as shown in the leftmost side of Figure 4.7. This way, end-users can formulate queries by browsing the EPCIS Webadapter and get updates in the Atom format which most browsers can understand and directly subscribe to. As an example a product manager can create a feed in order to be automatically notified in his browser or any feed reader whenever one of his products is ready to be shipped from the warehouse. More concretely, this results in sending an HTTP `PUT` request to `/eventquery/subscription?reader=urn:ch:migros:stgallen:warehouse:expedition&epc=urn:epc:id:sgtin:0057000.123430.*`, or, for a human client, clicking on the *subscribe* link present at the top of each HTML representation of query results.

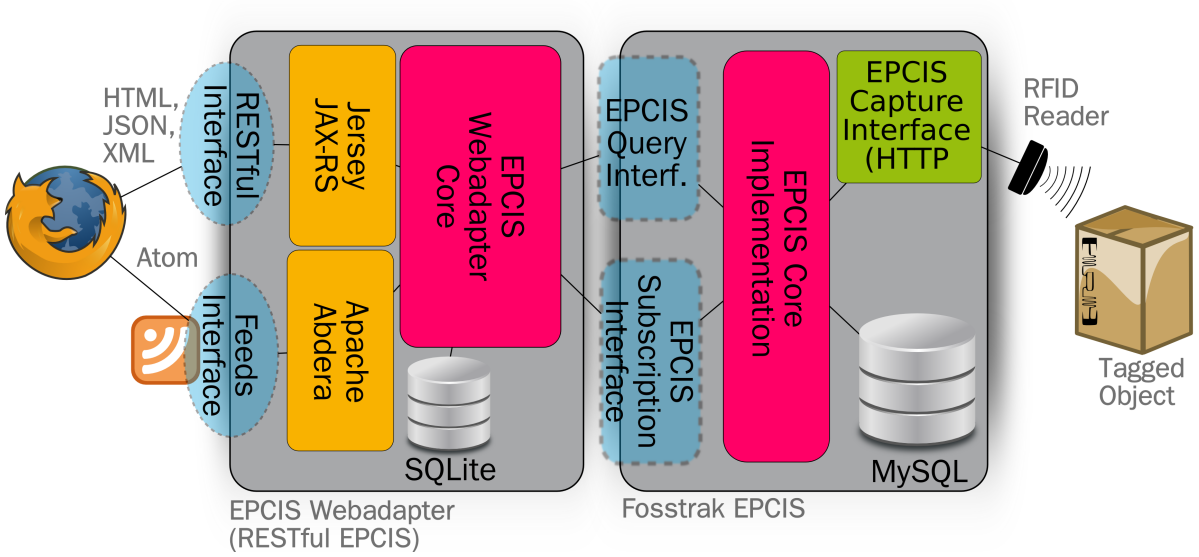


Figure 4.7: Architecture of the EPCIS Webadapter based on the Jersey RESTful framework and deployed on top of the Fosstrak EPCIS.

A product manager can then use the URI of the feed in order to send it to his most important customers for them to follow the goods progress as well. A simple but very useful interaction which would require a dedicated client to be developed and installed by each customer in the case of the WS-* based EPCIS.

From WS-* to REST: Integration Architecture

There are two ways of adding RESTful capabilities to a WS-* system. First, the RESTful architecture can be directly woven into the existing WS-* system. This may seems like a trivial solution at first, however the implementation of this solution is not entirely straightforward. While sharing a common goal, WS-* and REST are rooted on very different paradigms. Thus, cleanly weaving a REST architecture into the core of the WS-* system quite often requires the implementation of an alternate data model [75]. Having two data models for the same services ends up in architectures that are complex to maintain and evolve.

Smart Gateway An alternative integration pattern is to implement the concept of Smart Gateways presented in the Device Accessibility Layer of the Web of Things Architecture. In this case, a Smart Gateway is a software module that implements an external REST adapter making use of the WS-* interface. In this integration architecture, the Smart Gateway is used to translate RESTful requests into WS-* requests.

This allows for a cleaner, REST centric architecture and preserves the legacy WS-* system entirely intact. On the downside it hinders the performances of the RESTful Web API but the overhead can be minimized to a level acceptable for most applications as we will show in Section 4.6.

For the EPCIS Webadapter, we create an independent dedicated Smart Gateway, as it delivers a clear advantage in this case: it allows the EPCIS Webadapter to work on top of any standard EPCIS implementation.

The resulting architecture is shown in Figure 4.7. The EPCIS Webadapter is a module which core is using the EPCIS WS-* standard interface. Just as a Smart Gateway, it translates the incoming RESTful request into WS-* requests and returns results complying with the constraints of RESTful architectures. As shown on the left of the picture, the typical clients of the EPCIS Webadapter are different from the business applications traditionally connected to the EPCIS. The browser is the most prevalent of these clients. It can either directly access the data by browsing URIs or indirectly using scripting languages in Web pages.

Software Implementation

As shown in Figure 4.7, the core of the EPCIS Webadapter is based on the Jersey [249] framework. Jersey is a software framework for building RESTful applications. It is especially interesting since it complies with the JAX-RS [248] (JSR 311) standard for building RESTful Web services.

Jersey is responsible for managing the resources' representations and dispatching HTTP requests to the right resource depending on the request URI. When correctly dispatched to the EPCIS Webadapter Core, every request on the querying or browsing interface is translated to a WS-* request on the EPCIS. This makes the EPCIS Webadapter entirely decoupled from any particular implementation of an EPCIS. However, for our tests we used the Fosstrak EPCIS.

For the subscription interface we use Apache Abdera [221], an open-source implementation of an Atom-Pub server. Thus, every time a client subscribes to a query, the EPCIS Webadapter checks whether this feed already exists by checking the query parameters, in any order. If it is not the case it creates a query on the WS-* EPCIS and specifies the address of the newly created feed. As a consequence every update of the query is directly `POST`ed to the feed resource which creates a new entry using Abdera and stores it in an embedded SQLite database.

The EPCIS Webadapter core is packaged in a WAR (Web Application Archive) alongside with Jersey, Abdera and SQLite. As a consequence, like the Fosstrak EPCIS, it can be deployed to any Java compliant Web or Application Server. We tested it on both Glassfish [236] and Apache Tomcat [222].

4.3.3 Pushing from Readers to Web Clients

The second RESTful API meets the need for mobile or Web clients to access the raw data directly pushed by RFID readers through the LLRP and ALE protocols. The challenge here is that Web was mainly designed as a client-pull architecture, where clients

can explicitly request (pull) data and receive it as a response. This make the implementation of uses-cases where near real-time communication is required sub-optimal. As an example, a typical use-case is to push events that are being recorded by an RFID reader directly to a mobile browser application for monitoring purposes (see Section 4.5.2 for an implementation of this use-case).

For the EPC Network, we created two components as shown in Figure 4.1. The Capture App Webadapter acts as a multiplexer. It is a modular Web application which gets events from the ALE and redirects them to a number of RESTful Services (e.g., to the EPCIS Webadapter) for further processing. The services the application sends the events to can be configured through a RESTful interface on the Web as well, which allows to flexibly decide where RFID events should be routed to.

The second component is based on tPusher as presented in Section 2.1.3, which combines a RESTful API with a WebSocket and Comet server. Using a RESTful API, clients can subscribe to RFID event notifications for a particular reader by sending a `POST` request to a URI such as: `/t-pusher/reader/<READER-ID>`. This initiates a WebSocket connection with the server on which RFID events recorded by `READER-ID` will be pushed through the Capture App Webadapter.

4.3.4 Case-Study: EPC Find

To illustrate how implementing a Device Accessibility Layer for the EPC Network unveils new applications we describe and implement a mobile infrastructure on top of the EPCIS Webadapter as presented in [69]. The proposed infrastructure can be used to track and trace belongings.

Motivation

Losing something of great emotional or intrinsic (money or data!) value is often a shock. In this kind of situations we currently rely on lost property offices implemented and run by the travel business (airlines, train companies, coach services, etc.) or governmental organizations. In [128] it was identified that more than 400'000 items were lost in Switzerland in 2006. Amongst these less than 40% were recovered. In an era of high mobility, the solutions we rely on suffer from a number of problems. On the one hand, they lack dynamic information and compatibilities amongst the systems; on the other hand, they involve many intermediates and have high costs and no revenues for the institutions running them.

As for a number of systems, the existence of intermediates in the traditional approach decreases the efficiency and increases the costs. This fact was confirmed by interviews with experts at the Swiss National Railways (SBB), which run a large share of the Swiss lost property offices. Ideally, when Alice finds Bob's laptop, she should be able to report it directly to Bob. Of course this approach is not new and people have been enabling

this direct link for years using address tags providing contact details. This idea is rather straightforward however, we identify three main problems:

1. It reveals the owner's identity to everyone able to read the tag.
2. It requires manual updates: every time you change your address you need to change every name tag.
3. It denatures the object you tag by adding a relatively big label to it.
4. Besides his goodwill, there is no true incentive for the finder to return the found object.

Hence, the traditional approach to retrieve lost items can be enhanced by reducing the intermediates making it a more community-oriented process where finders are directly linked to owners. This improves the chances of recovery, simplifies the system and lowers costs. Combined with the use of mobile phones, RFID tags and EPC events, this can improve the dynamic information available to the owner of a lost item. For example, a consultant can know whether he simply left his laptop at home or whether it is lost and the incident needs to be reported to his company.

In our lost and found system, called EPCFind, we propose the prototype of a network for personal objects, that builds upon the EPCIS Webadapter.

Concept

With this prototype users interact using mobile devices to help them tracing and recovering users' belongings while on the run. In more concrete terms, with the *Distributed Tracing* approach, we can help an owner (Bob) getting dynamic information about where laptop might be located and in the *Community-Based Reporting* we help the finder (Alice) easily reporting the recovery of Bob's laptop while being on the move and without the need for intermediates.

Community-Based Reporting We propose to support a community of mobile phone users, which are able to communicate directly with the owners whenever they find an object. For this purpose, we use the EPCFind mobile software and wireless technologies. When Alice finds Bob's laptop, she can easily and quickly report the recovery by scanning the tag on the object. In order to do so, she uses the Report application of EPCFind, which connects to a central server and finds out about the object's owner. If Alice accepts it, the mobile application creates a trace of the recovery and reports it to Bob. Note that the system does not have to reveal Bob's identity. Instead, Bob uses the application shown in Figure 4.8 to directly contact Alice and arranges a way of sending the laptop back.

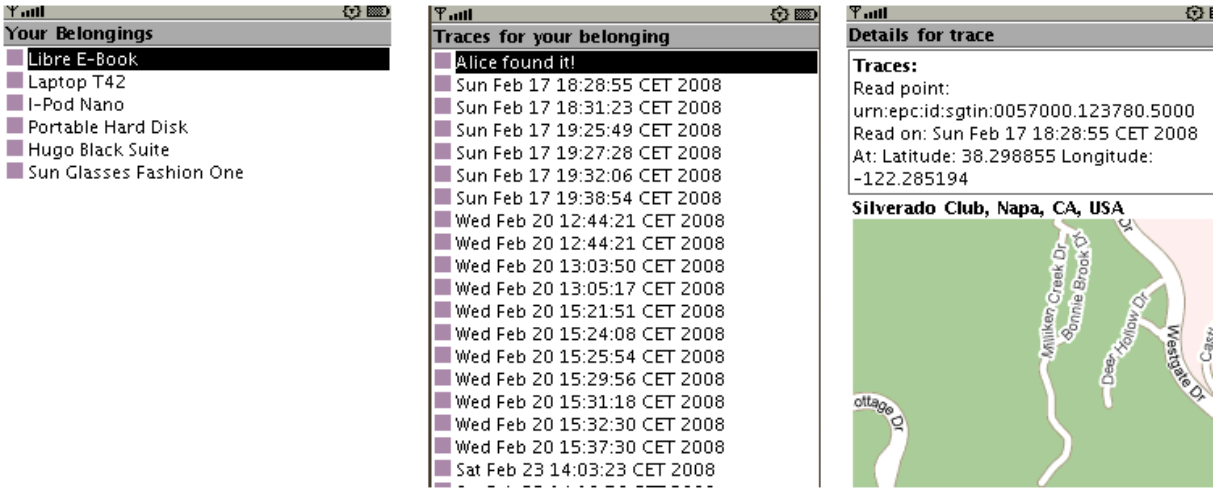


Figure 4.8: Mobile user interface of the owner. The screens offer: a list of the owner's belongings, the traces of the objects in the forms of EPC events, location details of the EPC events.

Distributed Track and Trace Community-based reporting fulfills the need for eliminating intermediates and eases the reporting process: Alice does not need to find the next lost property office and she can directly report the recovery to Bob using the EPCFind system. Yet, the system in this state does not resolve Bob's need for dynamic information: what if Bob's laptop was still at home? What if it got stolen and not simply lost? What if no one found it? To solve this issue, we extend our system with a distributed network of readers made available by the community. As proposed by Frank et al. [58, 59], we assume a network of readers formed by static (e.g. readers already in place in stores) and mobile devices (e.g. an RFID-enabled mobile phone). These distributed readers can silently (i.e., without explicit human interaction) register tagged objects in their vicinity. With EPCFind, Bob can use the application on his mobile phone to locate where his laptop was last *seen* by the distributed readers and make an appropriate decision based on this information (e.g., call the police, call his home, report the loss of his laptop to the company, etc.)

Similarly, the silent reporting can be used in order for Bob to register the presence of its own objects next to his mobile phone on a regular basis (e.g., while the phone and the laptop are on his desk at home, etc.). This approach reduces the privacy concerns inherent to the Distributed Tracing approach while not filtering the most valuable information in our case: when was the object last seen next to Bob?

Identifying Objects Core to the system is the notion of *selecting or scanning* physical objects using a mobile phone. The subject has been explored by several researchers already [201, 167]. As an example, Rukzio et al. identified touch using NFC tags as being a well-received interaction technique [167]. Hence, using NFC (Near Field Communication) seems quite natural for our application. However, because of its very limited range (i.e., touch), NFC needs a visible tag or zone of interaction. Beyond denaturing the object, it also concentrates the interaction metaphor on the tag rather than on the object.

Figure 4.9: Prototype of a mobile phone extended with a UHF EPC Gen2 RFID reader. This device can read RFID tags from a distance of about 30-50cm.

We change the interaction paradigm from identifying a tag representing the object, to identifying the object itself. This, we believe, can make the system rather easy and straightforward to use. Thus, we propose the use of EPC tags that can be read from a distance and without line of sight, thanks to the EPC Gen2 standard.

Leveraging the EPC Network The potential of the EPCFind system is relying on the size and contribution of the community. In more technical terms, a critical mass of mobile phones with RFID readers and tagged objects is required. Furthermore, the tagged objects need to disclose a number, which enables the unique identification of the object, unlike barcodes, which identify a type of product. Finally, as mentioned before, the reading range of the mobile readers needs to be greater than touch distance (ideally 20-30 cm). We propose using UHF tags, implementing the EPC Gen2 (Electronic Product Code tags, second generation) standard. These RF transponders fulfill both the need for a world-wide unique, instance-level ID and a greater read range. Furthermore, the planned deployment of the code on retail products would prevent from having to put tags on objects manually as they would already be tagged. It also permits to assign the ownership of an object at purchase time: freeing Bob from both having to tag his objects and registering them as his belongings. Indeed, by matching the EPC on the laptop with a unique number identifying Bob (e.g. his phone number, loyalty / credit card number, etc.), we can assign the ownership at the store, when Bob buys his laptop. The only drawback of this approach is the lack of UHF readers embedded in commercial mobile phones [208]. In order to overcome this problem for the implementation of EPCFind, we use three prototypes of Nokia E61i mobile phones as show in Figure 4.9. These phones are equipped with an UHF EPC RFID reader as a functional cover (i.e., the reader is integrated into the phone battery cover) potentially capable of reading up to 30-50 cm. From an human-computer-interaction viewpoint this is quite interesting improvement over

NFC mobile phones, since it slightly changes the way objects are identified, allowing users to identify objects as a whole rather than having to touch NFC tags [201].

Besides leveraging the unique numbers on tags (EPC Tag Standard), and the standardized reading of UHF tags (Reader Standard), the core of the EPCFind system is based on the EPCIS Webadapter used to hold information and traces of the tagged objects.

Implementation

The EPCFind software consists of two parts as shown in Figure 4.10: a mobile user interface and the backend built on top of the EPCIS Webadapter.

Mobile Application The mobile user interface is implemented using Java Mobile Edition [247] and needs to be installed on the mobile device of every member of the community. It is composed of three distinct `MIDlet`s, each representing one part of the application. The `Report MIDlet` is used by Alice to report the recovery of Bob's laptop. It activates the UHF RFID reader on the mobile phone and asks Alice to approach the phone to the object in order to identify it. It then reports the recovery to the EPCFind backend using either a WiFi or a GPRS connection. The `AutoReport MIDlet` is a process, which can run in the background in order to implement the distributed tracing. It activates the reader and reports an RFID event to the EPCIS Webadapter each time a (new) tag is in the scope of the reader. The `Find MIDlet` is the counterpart of the `Report` and `AutoReport MIDlet`s. It enables Bob to retrieve information about his belongings as shown in Figure 4.8. Using a unified interface, it offers access to two types of information: traces and recoveries. Traces are the events generated silently by the `AutoReport MIDlet`. They provide information about where an object was last seen, and thus allow deducing where it might be located. Recoveries are generated whenever a member of the community uses the `Report MIDlet` to signal the recovery of an object to its owner.

EPCFind Backend Traces of objects reported by the mobile software (`Report` and `AutoReport MIDlet`s) are stored in the Fosstrak EPCIS through the EPCIS Webadapter. The mobile software (`Find MIDlet`) also uses the RESTful Web API of the EPCIS Webadapter for queries of objects' traces.

The EPCFind backend software is used to implement and manage ownership of objects, i.e., it needs to know what belongs to Bob and control the information Bob and Alice can access.

Discussion on Privacy

As for a number of applications involving automated and pervasive tracking of objects or people, EPCFind raises some privacy issues. While, we do not pretend solving all of

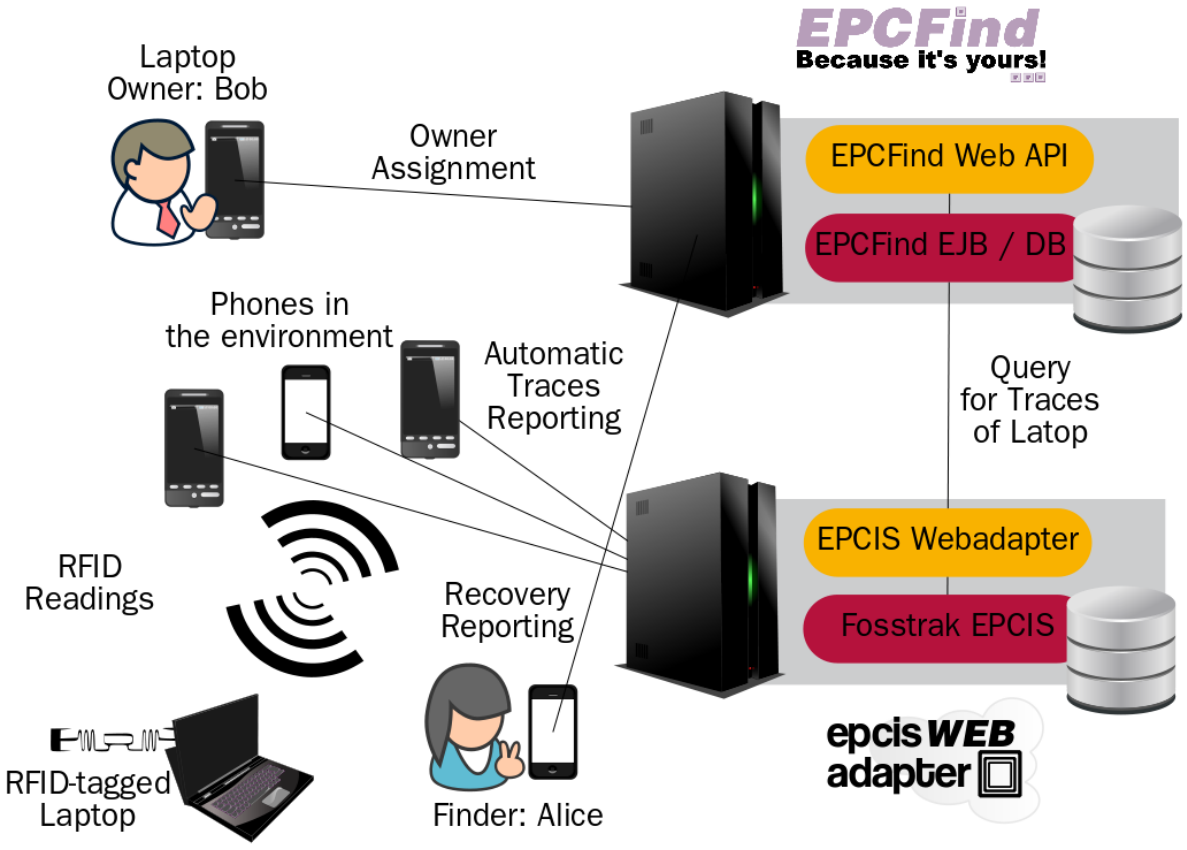


Figure 4.10: Architecture of the EPCFind prototype. Mobile phones in the environment communicate with the EPCIS Webadapter deployed on top of the Fosstrak EPCIS and record traces of tagged objects. Through the mobile EPCFind application communicating with the EPCFind backend and the EPCIS Webadapter, Bob can trace his lost laptop. It is eventually found by Alice who communicates it to the EPCFind backend and the EPCIS Webadapter.

them with our current implementation let us briefly identify flaws and discuss solutions based on other works in the area.

First of all, because of the distributed tracing, the EPCIS Webadapter and EPCFind backend contain location information about ones belongings. This could potentially enable Alice to track Bob by querying the system for the location of his laptop. We take a rather simple approach and prevent this from happening by selectively giving access to object traces as hinted in [3]: A user can only query the system for traces of items he owns. Note that this method is not fail-proof and restricts quite a lot the possible usage of the precious data EPCFind collects. Kriplean et al. extensively discuss the general issue of protecting Auto-ID information servers in [3]. The other privacy concern is driven by the fact that users carry tagged objects, which can be read in a silent manner and from a distance. This introduces two major problems: Firstly, the EPC mobile phones could be used to *x-ray* bags or suitcases and detect items one does not want to publicly show. Secondly, the tags one carries on a regular basis could be used to profile the user (e.g., by stores) and possibly identify him using inference techniques and information leaks. These two flaws are not inherent to the EPCFind system, but rather to wireless identification and communication systems (and beyond) and thus are explored extensively in literature. Partial solutions for RFID range from encrypting the tags to providing means for *killing* a tag, a solution supported by the EPC Gen2 tags [106].

4.4 Sharing Layer

This section demonstrates how the Sharing Layer of the Web of Things Architecture can be leveraged to implement a simple and secure sharing mechanism for EPC events. It builds on top of the EPCIS Webadapter and leverages its Web API.

4.4.1 Pain-Point: Lack of Access Control

Core to the vision of the EPC Network is the notion of companies sharing their EPC traces openly or at least with partners [189]. However, this requires a change of strategy for several actors of the supply chain since they currently use the information opacity as a means to negotiate better deals.

For these important actors, a complete data openness is not really an option and was identified as one of the important barriers towards the adoption of the EPCIS as a data sharing standard [172, 72, 26]. Thus, a realistic global deployment of the EPC Network requires a more comprehensive and flexible access control and sharing framework that allows a thinner-grained selection of the partners the data are shared with. This topic is being researched on actively [117, 26, 211] and a standard at least partially addressing this issue is currently under development by EPC Global[1]. Rather than purely focusing

[1] As of June 23, 2011 according to the EPCglobal Web page [230].

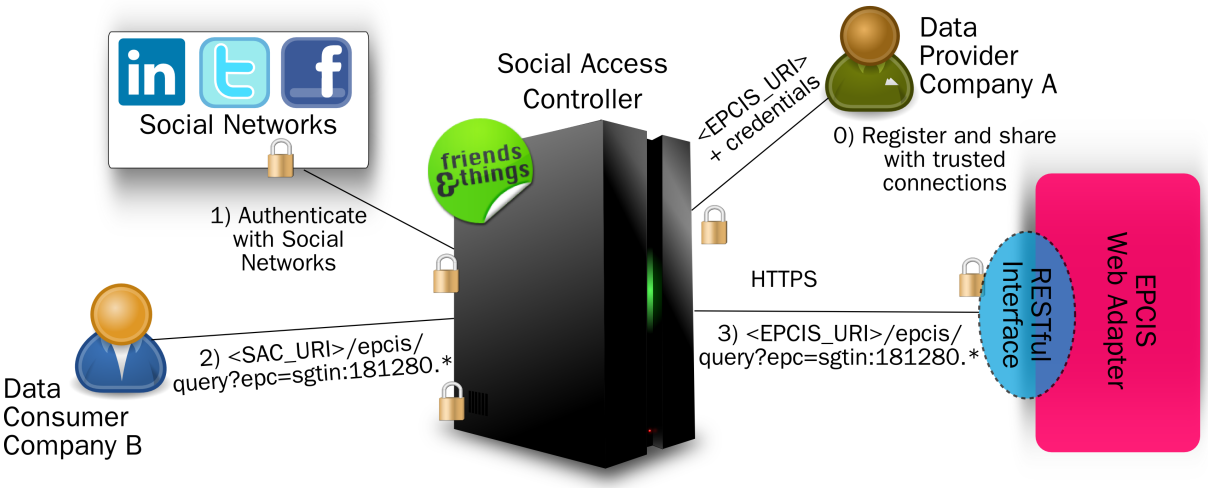


Figure 4.11: The Social Access Controller is managing the access control to the EPC events. In this case access to all products from the manufacturer 181280 is granted to the data consumer, through his social network credentials.

on access control, this new standard, currently called *Discovery Services* is also expected to solve another important problem to enable global sharing: the need for a discovery service that allows looking for product traces across several EPCIS instances.

While it does not solve all the open issues (see Section 4.8), the Sharing Layer of the Web of Things Architecture can easily be leveraged to provide a simple and secure mechanism for sharing data and managing access control on top of the EPCIS Webadapter.

4.4.2 System Architecture

The key idea is to secure the resources of the EPCIS Webadapter using a standard HTTP authentication method (e.g., HTTP Basic Authentication with SSL/TLS or HTTP Digest Authentication). Then, the Social Access Controller is used as an authentication and authorization proxy for accessing EPC traces. An overview of the system architecture is shown in Figure 4.11.

Registering and Sharing EPC Events The EPCIS Webadapter is registered with the Social Access Controller by the *Data Provider* at Company A. This is done either through the Friends and Things Web front-end or using the SAC API, by providing its root URI and access credentials (step 0 on Figure 4.11). Thanks to the fact that the EPCIS Webadapter respects the REST constraints, it can be easily crawled by the STM Translation Service used by SAC. As a result, the resources offered by the EPCIS Webadapter (e.g., products, readers, locations, etc.) are identified and can be shared.

The Data Provider then shares all (past and future) EPC events in which the manufacturer is 181280 which corresponds to the following URI: `<WEB-ADAPTER-URI>/rest/1/eventquery/result?epc=urn:epc:id:sgtin:181280.*`. He shares it with a particular

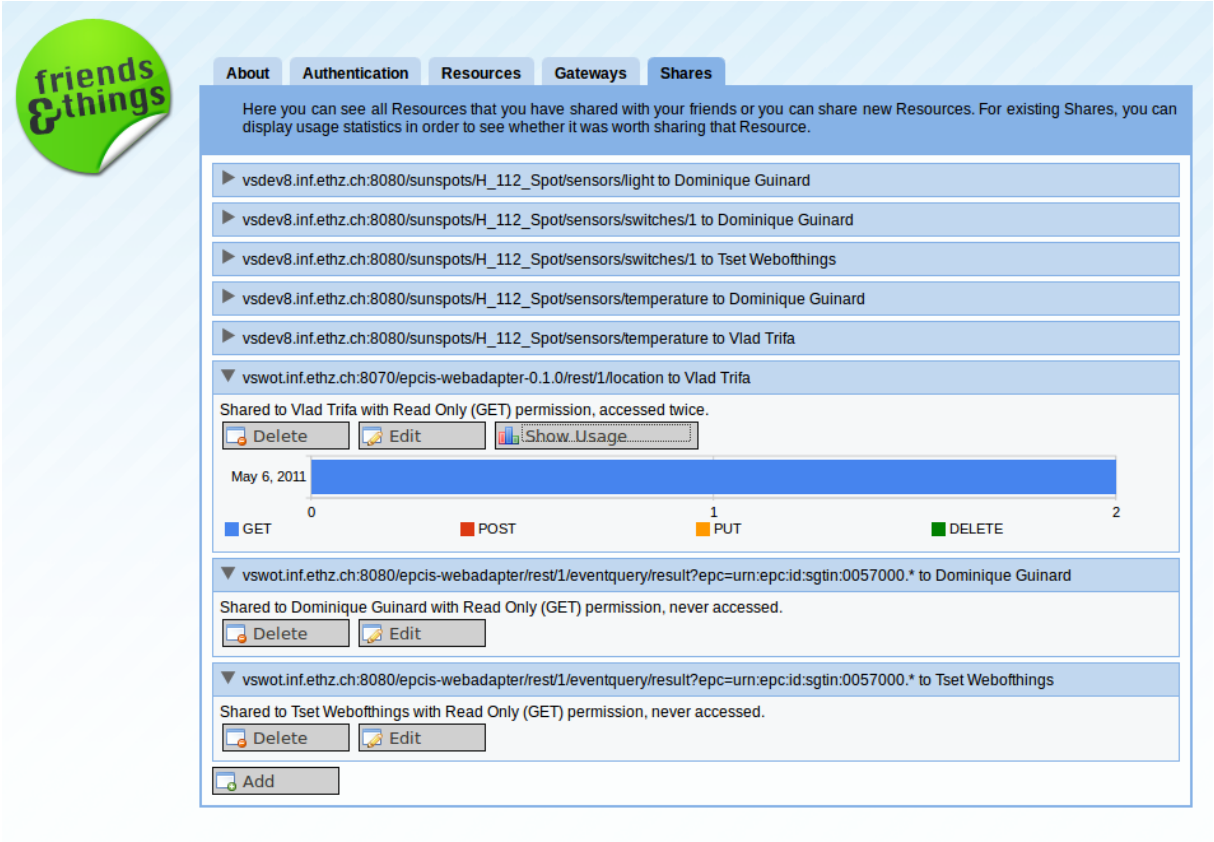


Figure 4.12: Friends and Things user interface for monitoring the resources' usage. Usage statistics for several resources are available on this page. For instance, traces of all the products from manufacturer 0057000 can be accessed by user *Tset Webofthings*. The displayed graph shows that user *Vlad Trifa* accessed the resource `location` twice on May 6, 2011.

Data Consumer at Company B. The Data Consumer is selected from a list of trusted connections amongst the social networks SAC has access to for the Data Provider. Once the resource is shared, SAC posts a message directly to the social network of the Data Consumer to inform him about the newly shared resource and its SAC-URI.

Accessing Shared EPC Events As shown in step 2 of Figure 4.11, once the Data Consumer at Company B received notification of the shared EPC resource he can access it simply by using the provided SAC-URI: `<SAC-URI>/gateways/<WEB-ADAPTER-URI>/resources/rest/1/eventquery/result?epc=urn:epc:id:sgtin:181280.*`.

In this URI, SAC considers the EPCIS Webadapter as a `gateway` which offers a number of `resources`. The URI is resolved by the corresponding SAC instance which checks whether the Data Consumer is allowed to access this EPC resource (or a resource located at an upper level in the hierarchy). If it is the case connects to the EPCIS Webadapter using HTTP Digest Authentication and redirects the results to the Data Consumer.

Since SAC acts as a proxy, it can keep a trace of each requests from Data Consumers. This data is made available to the Data Provider through the SAC API or visually through the Friends and Things Web front-end. This can be used to monitor the consumption

of shared data. As an example, Figure 4.12 shows that the resource `location` was used twice by Data Consumer Vlad Trifa on May 6, 2011. These logs can then be used to limit the number of accesses or to charge a fee for data access and facilitate the creation of an RFID data market place.

4.5 Composition Layer: Auto-ID Physical Mashups

As mentioned before, bringing RFID data closer to the Web creates opportunities for new applications. In this section we describe how the Composition Layer of the Web of Things Architecture can be leveraged to build these applications. We illustrate how we can create physical mashups for RFID applications on top of the the Device Accessibility Layer and the Sharing Layer.

4.5.1 Pain-Point: Tedious Business Case Modeling and Cross Systems Integration

RFID use-cases generally do not involve RFID readers and tags only, they are most of the time combined with sensors and actuators. These combinations of RFID, sensors and actuators often occur at a low level, sometimes even at the wiring level. This mainly has two drawbacks. First it requires to combine the complicated and often not homogeneous low-level APIs of devices which requires expert knowledge. Then, once installed, these compositions of devices are static and cannot be flexibly reconfigured to integrate new sensors or actuators.

In this section we illustrate how the architecture and approaches introduced in the Composition Layer can be leveraged to create physical mashups for RFID use-cases. We distinguish three levels of mashability as introduced in the Composition Layer: *Manual Mashup Development*, *Widget Based Mashup Development* and *End-User Development with Mashup Editors*.

We present three concrete prototypes illustrating these levels of mashability. The Mobile Tag Pusher prototype is an illustration of Manual Mashup Development. The EPC Dashboard Mashup demonstrates how Widget Based Mashup Development can be enabled. Finally, the RFID Physical Mashup Editor illustrates how End-User Development with Mashup Editors can be used by end-users to create simple applications.

4.5.2 Mobile Tag Pusher

When setting up RFID readers or maintaining existing deployments it is valuable to have a direct feedback of the tags observed by a particular reader in order to monitor the manufacturing process or to debug the readers. In the current implementations of the EPC software stack this would require to use and configure a monitoring tool such as the

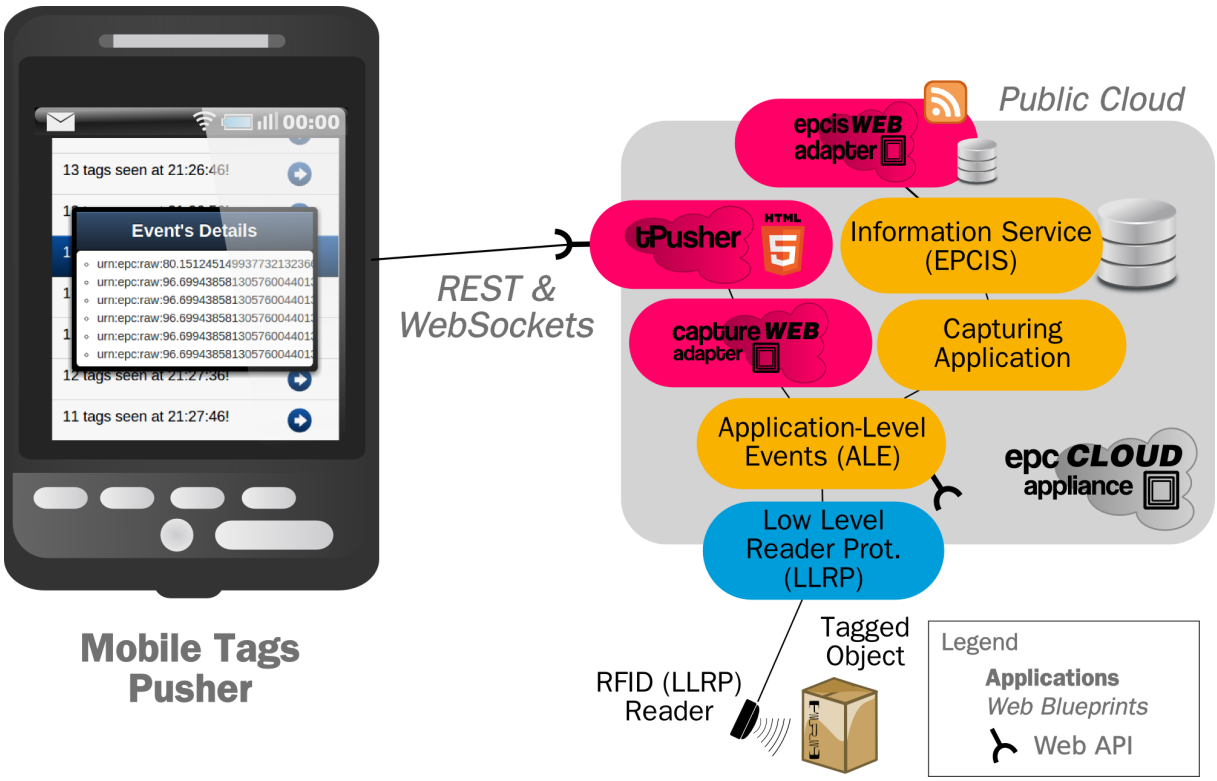


Figure 4.13: The Mobile Tag Pusher is built as a mobile Web application using the WebSocket API of the tPusher component. RFID events are routed by the Capture App Webadapter to the tPusher application where they are sent to all subscribed mobile Web browsers.

Fosstrak LLRP Commander on a desktop computer. Thanks to the *RESTful interface* of the Capture App Webadapter as well as the Real-Time Web capability of tPusher, the tags observed by any reader can now be directly pushed to any browser or HTTP library.

Because these events are of interest in-situ, we developed a Mobile Web mashup that can display them in a user-friendly manner. The Mobile Tag Pusher is a Web application that uses the tPusher service to subscribe to events coming from RFID readers, as shown in Figure 4.13. As a consequence, EPC events are pushed to the application after being filtered by the LLRP and routed by the Capture App Webadapter to the tPusher WebSocket service.

This enables users of the application to get near real-time feedback about the events an RFID reader is currently recording.

Implementation The Web application is a pure HTML5 and JavaScript Web application built using the Sencha Touch library [255]. Furthermore, it is based on an abstraction of push mechanisms using the Atmosphere JQuery Plugin described before. This basically means that it can be used, without installation, on most recent mobile browsers such as Safari (iOS) or Chrome (Android).

All code required for the mobile application to subscribe to events pushed by readers through the Capture App Webadapter and display them fits within 5 lines of JavaScript

Figure 4.14: The data read by the RFID LLRP Gate reader is sent via real-time Web (WebSockets) to a mobile phone application running in the mobile browser. The Web application is accessed simply by scanning a QR-code.

as shown below:

```
1 //called whenever an event is pushed:
2 function callback(response) {alert(response.responseBody +
     response.transport);}
3 //subs. to the events of reader "exit1"
4 $.atmosphere.subscribe(
5   "http://EPC_CLOUD_APPLIANCE/t-pusher/reader/exit1",
6   callback, $.atmosphere.request = { transport: 'websocket' }
7 );
```

As shown in Figure 4.14, we deployed this prototype in a lab environment. Each reader features a QR-Code containing its unique URI in the EPC Cloud. When scanning this tag with a mobile phone it redirects the user to the HTML5 Web page shown in Figure 4.13. As tags are read by the readers, the Web page automatically receives and displays new events.

4.5.3 The EPC Dashboard Mashup

The EPC Dashboard Mashup is a Widget Based Mashup that helps product, supply chain and store managers to have a live overview of their business at a glance. It can further help consumers to better understand where the goods are coming from and what other people think of them. The EPC Dashboard is based on the concept of widgets in

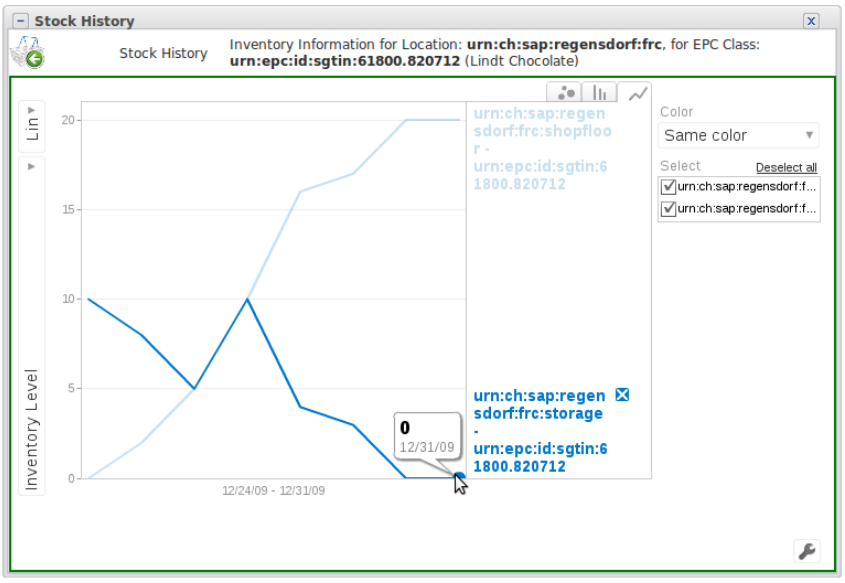


Figure 4.15: The Stock History widget allows for looking at the flows of goods through the supply chain. Here the manager can see that all the available Lindt chocolate has been transfered to the shop, leaving an empty stock.

which the event data are visualized in a relational, spacial or temporal manner. Widgets can be easily extended by developers using a simple framework.

The EPC Dashboard consumes data from the EPCIS Webadapter optionally shared through the Sharing Layer. Usually these data are hard to interpret and integrate. The dashboard makes it simple to browse and visualize the EPC data. Furthermore, it integrates the data with multiple sources on the Web such as Google Maps, Wikipedia, Twitter, etc. To better understand the use of such a tool, let us first introduce two use-cases before looking at the applications' architecture.

Use-Cases

Rachel, a customer, just bought Max Havelaar Bananas and Lindt Chocolate from a retail store *M* in Switzerland. She wants to know more about the Bananas. For that purpose, she opens the EPC Dashboard in her preferred browser. She activates the Product Description Widget and enters the EPC of the article. The EPC Dashboard now shows her a description of bananas and Max Havelaar extracted from Wikipedia. Likewise, the Product Video Widget provides her with video about planting of these bananas. She is further interested to know about where this particular banana has grown. Rachel activates the Map Widget and she can see on the map in Figure 4.16 where her banana originates from. In addition she also sees the route that the banana has taken from its origin to the M retail shop. She finally prepares a Banana Split with the chocolate she just bought and shares the recipe on Twitter through the Product Buzz Widget. In the M store in Zurich, Andy, the product manager of chocolate products wants to check the recent inventory levels of the Lindt Chocolate. He also browses to the EPC Dashboard Mashup and opens the Stock History Widget with the corresponding inventory RFID

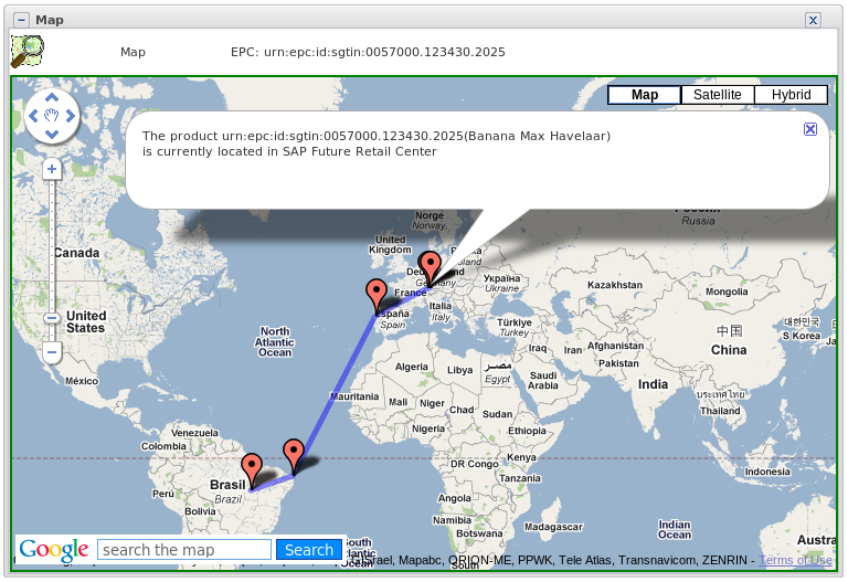


Figure 4.16: The Maps widget is following the route of the banana tagged with the EPC urn:epc:id:sgtin:0057000.123430.2025.

reader. According to Figure 4.15, he discovers that there had been an increase demand for Lindt Chocolate and that the chocolate has almost entirely been transfered from the stock to the shop. He directly orders larger shipping contingents. He also subscribes to the feed listing the arrival of Lindt products in the stock using the entry gate reader. This way, a feed reader on his mobile phone and in his favorite browser will inform him when the ordered products have arrived. He is further interested in knowing why the Lindt products are so popular recently. Andy activates the Product Buzz Widget and sees the current Twitter messages related to Lindt Chocolates as shown in Figure 4.17, including Rachel's recipe. He can use this information for marketing analysis.

Mashup Architecture

The EPC Dashboard integrates several information sources. This information is encapsulated in small windows called widgets. The widgets combine services on the Web with traces coming from the EPCIS Webadapter. The EPC Dashboard Mashup currently offers 12 widgets using different APIs and services. As an example, the Map Widget is built using the Google Maps Web API (see Figure 4.16), the Product Buzz Widget uses the Twitter RESTful API (Figure 4.17) and the Stock History Widget uses the Google Visualization API (Figure 4.15).

All widgets are connected to each other which means that actions on a given one can propagate the selection to the other widgets and changes their view accordingly. As such, widgets listen to selections and can make selections. This interaction is implemented using the observer pattern [62] where consumers (i.e., the widgets) register to asynchronous updates of the currently selected Locations, Readers, Time or EPCs. This architecture allows the creation and integration of other Web widgets with very little effort. The EPC

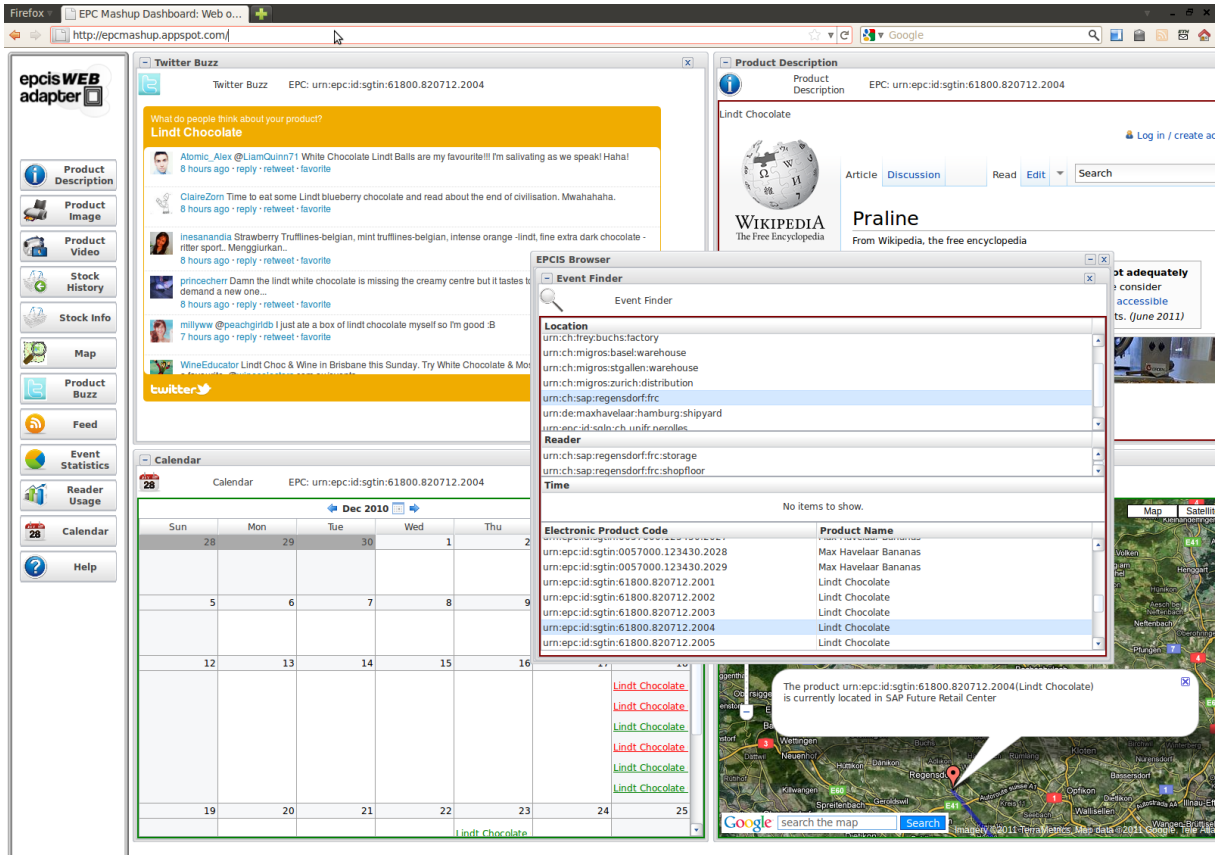


Figure 4.17: Screenshot of the EPC Dashboard Mashup Web page. On the left the user can select the widgets he wants to activate. The widget in the middle is used to browse the EPCIS data. The first widget (upper-left) is the Product Buzz Widget which extracts live opinions and information about the product (here Lindt Chocolate) from Twitter. The Product description widget queries Wikipedia for information, the Calendar Widget provides an overview of the EPC events by date and finally, the Map Widget show the location of the product.

Dashboard itself is a JavaScript application built using the Google Web Toolkit [239], a framework to develop rich Web clients. This type of development is possible thanks to the RESTful Web Interface of the EPCIS Webadapter.

4.5.4 RFID Physical Mashup Editor

Applications of RFID technologies are at strongly influenced by business processes within a company. Hence, it is relevant to give the power to create simple applications not only to developers but also to end-users, more aware of the process in their business.

As introduced in the Composition Layer, end-user mashability is usually enabled through simple drag-and-drop visual tools called Mashup Editors. In the following sections, we describe a typical use-case of business process in the RFID domain and then present a mashup editor tailored to RFID use-cases.

Use-case: Electronic Article Surveillance with RFID

Here, we describe a common RFID application, RFID as an Electronic Article Surveillance (EAS) technology, and illustrate how the EPC Network framework can be used to realize this application. This example illustrates the challenges in RFID application development and deployments.

In many clothing stores, RFID technology is set to replace existing Electronic Article Surveillance technology because of its many advantages. The two most important issues include knowledge about the product being stolen and the reduction in the number of false alarms. Today, retail stores have little information about which particular product is actually being stolen. As a result, the stores cannot replenish the shelves appropriately resulting in a possible lost sale to a consumer who is willing to pay for the item. There is also no way to prevent frequent false alarms where products with active EAS tags from another retailer trigger the alarm.

Using RFID technology as an EAS system relies on RFID tags on the individual clothing items as well as RFID readers at the back-room door to the store, where clothing enters the store, at the checkout and at the exit. When products are placed on the shop floor, the RFID tags are read as they pass the reader at the back-room store entry. The applications registers the tags and marks the IDs as 'on sale'. If a consumer decides to purchase an item, the RFID tag is read again at the checkout and flagged as 'sold'. If the user leaves the store with the product, the RFID readers at the exit report the RFID tag to the application, but no alarm is triggered because the product is marked as sold. If the consumer decides to leave the store without paying, the RFID tag is identified by the readers at the exit and an alarm is triggered because the product is still not paid for.

To realize the above example, the RFID readers need to be mounted in the store and connected to a local area (wireless) network. After discovery of the readers on the network, each RFID reader needs to be configured to read RFID tags and report RFID tag read reports via the binary EPCglobal LLRP reader protocol. To prevent RFID readers running continuously, the back-room reader is often triggered via a motion sensor. There is also an alarm connected to the network that can be triggered by the RFID readers at the exit. Following the configuration of the readers, an application server needs to be set up on a server in the clothing store that runs the RFID middleware. In the case of the EPC Network, such an application server would run an instance of an Application-Level-Events (ALE) compliant middleware that filters and aggregates the RFID data. Using the ALE WS-* API, the developer would need to group the RFID readers at the various locations (entry, exit, and checkout) and also define time filter and aggregators that eliminate redundant RFID reads. In a typical RFID deployment, the appearance of an RFID tag in the read range of an RFID reader can result in numerous tag reads of the same tag. To process the filtered and aggregated RFID data, custom business logic needs to be implemented. The business logic of the EAS application needs to deserialize the incoming ALE SOAP messages containing the tag reads, send off web service EPCIS (EPC Information Services) query interface to check the state of the particular tag ("on

Building Block	Inputs	Outputs
`RFIDReader`	URI, Reader ID (e.g., exit-reader)	EPCs
`EPCIS`	URI, EPC, Business step (e.g., checkout)	true/false (EPC seen at Business step)
`VideoCamera`	URI	URI of (stored) snapshot
tPusher	URI, String to push, topic name	true/false (success/failure)

Table 4.2: Additional building-blocks for implementing the EAS use-case.

sale", "sold"), and create a new EPCIS event that triggers a state change and possibly sound an alarm. To store and access these EPCIS events, the developer needs to set up a database on the application server and deploy an EPCIS repository that supports the EPCIS capture and query protocols. The developer might also decide to develop a custom applications that queries the EPCIS repositories across multiple stores to provide analytics capabilities. The retailer might for example want to identify the products most stolen and locations of stores with the most stores.

A direct consequence of the complexity of installing and implementing the use-case we described here is that many smaller businesses decide to adopt very basic solutions where non-standard tags simply trigger an alarm every time they pass the gate.

Use-Case Implementation

Thanks to the deployment of the EPC software stack in the cloud and the implementation of the Device Accessibility Layer and Sharing Layer, we can now implement physical mashup editors for enabling users to flexibly model variations of use-cases such as the EAS presented before. For this use-case, we design new building-blocks as shown in Table 4.2.

These modules were implemented as building-blocks of our modified version of the Clickscript (see Section 2.4.3) mashup editor. Reducing interfaces of the EPC Network to Web interfaces enables each building block to be implemented with a small amount of JavaScript code. Note that using a comprehensive Findability Layer for the EPC Network can enable the automatic generation of such building blocks as described in Chapter 2.2.

Using these building-blocks and other basic blocks, we can implement the EAS use-case we introduced before. As shown in Figure 4.18, the building-blocks of the RFID mashup editor communicate with several components of the EPC Cloud. First, the `RFIDReader` block subscribes to the tPusher service using a particular reader ID (e.g., *exit-gate*). As a consequence, it gets pushed (through LLRP, ALE and the Capture App Webadapter) all the EPC events for this reader. The `EPCIS` block is then used to check whether the pushed EPCs represent goods that were already sold. To check this, the block uses the EPCIS Webadapter.

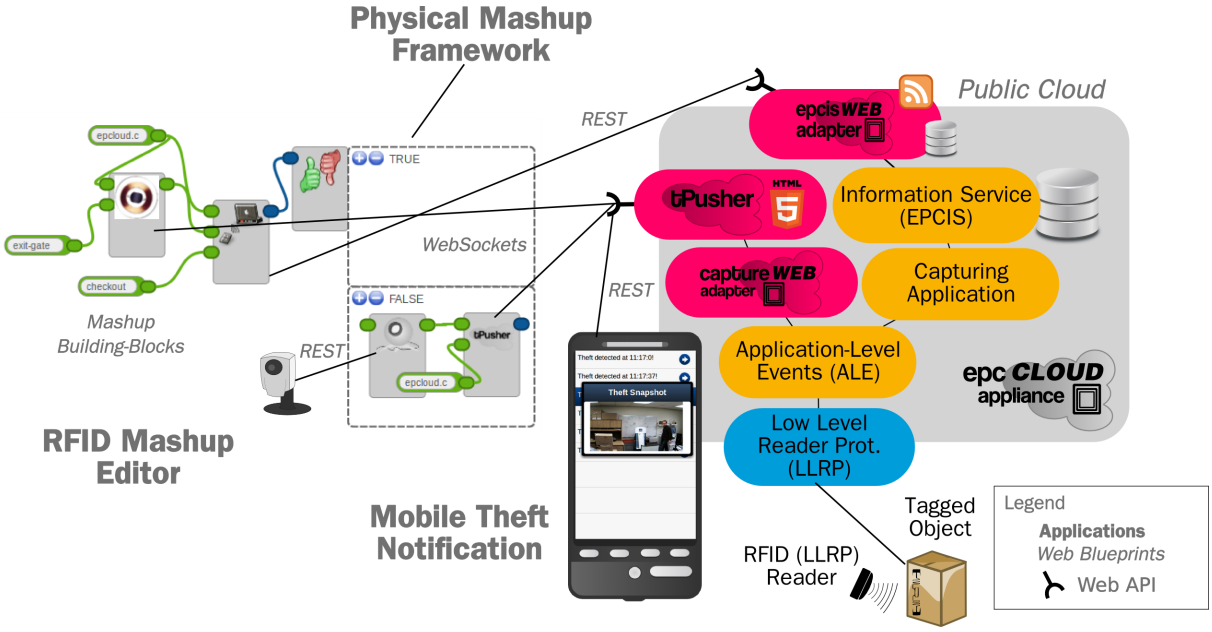


Figure 4.18: The RFID Mashup editor is used to model the EAS use-case. The created mashup can then be run from the Physical Mashups Framework. Thefts are reported to a mobile phone.

If it is the case, nothing happens. If it isn't the case (i.e., the goods were stolen), the `VideoCamera` block is triggered. This components represents a Web-enabled video camera that can be used to take snapshots through a RESTful API. The URI of the snapshot is then sent to all subscribers of a particular topic (i.e., URI) through tPusher. As an example we developed a small mobile Web application, similar to the Mobile Tag Pusher application, which subscribes to the topic and loads the corresponding image alongside with the EPC number of the stolen good (see mobile phone in Figure 4.18). Such an application can be used to push information about the theft to all staff members in a store. Once a mashup has been successfully created and tested locally using a mashup editor, it can be deployed to a mashup engine such as the Physical Mashups Framework (see Section 2.4.5) where is it going to be deployed remotely executed.

The full use-case was tested in a lab deployment at MIT featuring a gate LLRP reader and an off-the-shelf Webcam as shown in Figure 4.14. The average observed RTT (from the reader, to the Amazon Cloud instance, through the mashup engine and finally to the mobile Web application) was around 1 second. However, it is worth noting that this RTT strongly depends on factors such as the available connection bandwidth, the type of instances used on Amazon EC2, the current load of the cloud appliance. Since these factors cannot all be controlled this is a real challenge for this type of applications that we further discuss in Section 4.8.

4.6 Evaluating the EPCIS Webadapter

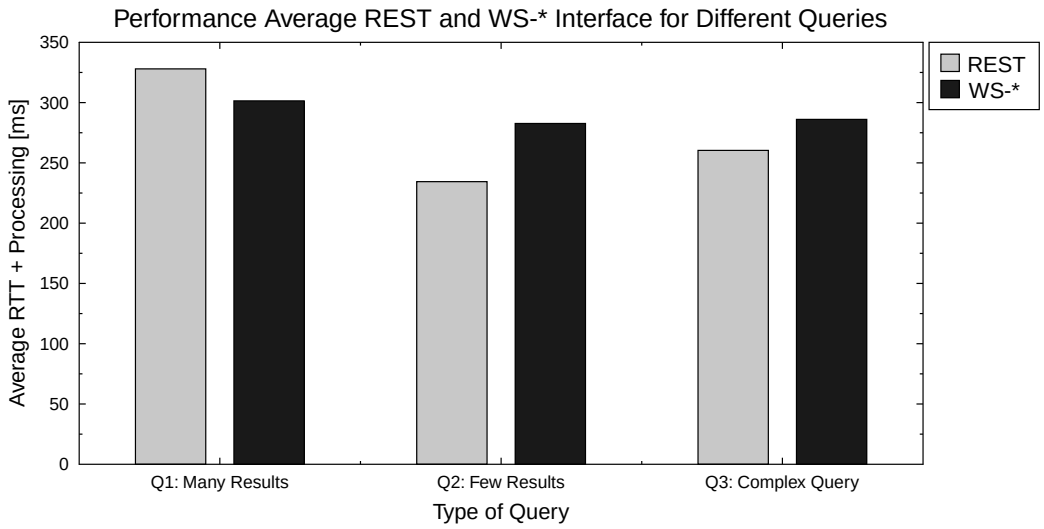


Figure 4.19: Average RTT and processing time when using the WS-* interface and the REST interface for three types of requests each run 100 times. Standard deviations are as follow: 49, 77, 39, 11, 38, 12 ms.

As mentioned before, the EPCIS Webadapter is an add-on to the standard EPCIS where each REST request is eventually translated to a (local) WS-* request. This results in an overhead that we evaluate here.

The experimental setup is composed of a Linux Ubuntu Intel dual-core PC 2.4 GHz with 2 GB of ram. We deploy Fosstrak and the EPCIS Webadapter on the same instance of Apache Tomcat with a heap-size of 512 MB. We evaluate three types of queries all returning the standard EPCIS XML representation.

The first query (Q1, *Many Results* in Figure 4.19) is a small request returning a relatively large set of results of about 30 KB (22 events each composed of about 10 EPCs). In the second test (Q2, *Few Results*), is a query returning 2.2 KB of data with only two results. The last test (Q3, *Complex Query*) is a query containing a lot of parameters and returning 10 events. We test each of these queries asking for the standard XML representation. All queries are repeated in 10 runs of 100 requests from a client located on a machine one hop away from the server with a Gigabit Ethernet connectivity. The client application is programmed in Java and uses a reference JAX-WS client implementation for the WS-* calls and the standard Apache HTTP Client and DOM (Document Object Model) library for the REST calls.

As shown in Figure 4.19, for Q1 the EPCIS Webadapter has an average overhead of 30 ms due to the computational power required to translate the requests from REST to WS-* and vice-versa. For Q2 and Q3 the REST requests are executed slightly faster (about 20 ms) than the WS-*. This is explained by three factors. First, since there are fewer results, the local WS-* request from the EPCIS Webadapter is executed faster. Then, REST packets are slightly smaller as there is no SOAP envelope [214]. Finally, unmarshalling

WS-* packets (using JAXB) on the client-side takes significantly longer than for REST packets with DOM. For Q3, similar results are observed. Overall, we can observe that the EPCIS Webadapter creates a limited overhead of about 10% which is (over) compensated in most cases by the relatively longer processing times of WS-* replies. This becomes a particularly important point when considering devices with limited capabilities such as mobile phones or sensor nodes as well as for client-side (e.g., JavaScript) Web applications.

It is worth mentioning that the WS-* protocol can be optimized in several ways to better perform, for example by compressing the SOAP packets and optimizing JAXB. However as the content of HTTP packets can also be compressed this is unlikely to drastically change the results. Furthermore, because they encapsulate requests with HTTP `POST`, WS-* services cannot be cached on the Web using standard mechanisms. For the EPCIS Webadapter however, all queries are formulated as HTTP `GET` requests and are fully contained in the request URI. This allows to directly leverage from standard Web caching mechanisms which would importantly reduce response time [214].

4.7 Related Work

Researchers in the fields of Ubiquitous and Pervasive Computing have long been using RFID as a means to enhance real-world objects in order for these to become smart things. In the DataTiles [159] project, Rekimoto et al. proposed a tangible user interface composed of acrylic tiles. These physical tiles represented virtual data and were identified thanks to an embedded RFID tag that was read by an array of reader. In the MouseField [130] project presented by Masui et al., a small RFID reader featuring motion sensors was used to identify RFID-tagged real-world objects such as compact discs.

These pioneer projects raised the awareness on the possibility to use RFID as an interface for interacting with virtual data. However, they did not yet propose a systematic solution to link them to an information network. In the Cooltown project [110] Kindberg et al. proposed to use the Internet and the Web as the information network of choice for smart things. Exploring this idea of merging RFID enhanced objects and the Web, Welbourne et al. [207] create an RFID-based microcosm for the Internet of Things deployed throughout the University of Washington. They further developed a suite of Web-based tools to help users manage their personal RFID data and triggers. Römer et al. looked at extending every-day objects such as playing cards or tool-boxes [165] and proposed the use of WS-* services to facilitate the real-world integration. In [61] we explored the use of pub/sub mechanisms to facilitate the integration of RFID-tagged objects with the backend systems of hospitals. Broll et al. looked at linking RFID tagged-objects to services on the Web. They proposed to use NFC tags on physical objects (e.g., posters) as a bootstrap for mobile interaction with WS-* services [23]. To be able to technically achieve this, in [24] the same authors proposed a system based on an Interaction Proxy that sits between the mobile phones and the WS-* services and renders adapted mobile content. Similarly, Vermeulen et al. [198] looked at creating mashups based on RFID tagged objects. For

instance, tagged pictures can be combined with a tagged physical map to create a Google Maps mashup where the pictures appear at the right position on the map. To achieve this they proposed a framework based on a Phidget RFID reader [261] communicating through a PC with WS-* services representing the virtual mashup building blocks.

These projects emphasize the potential of enabling the connectivity (or identification) of real-world objects on the Web through RFID. Our approach here differs in the sense that rather than looking at a macro, prototypical level, we look at an existing global network of RFID-tagged objects and propose Web APIs and technologies so that this network can be leveraged to build large-scale Web and mobile applications consuming RFID data. The great potential of the EPC network [169, 170] for researchers in the ubiquitous computing field has led to a number of initiatives trying to make it more accessible and open for building prototypes and disruptive applications. Floerkemeier et al. initiated the Fosstrak project [56], which is to date the most comprehensive open-source implementation of the EPC standards. The idea of the Fosstrak project was to be a reference implementation of the standards. Thus, it offers interfaces for applications as described in the standard, using WS-* interfaces. As an example, the Fosstrak EPCIS is an open-source implementation of a fully-featured standard EPCIS [57] which features a WS-* interface as an application integration end-point. A direct consequence is that it prevents the Fosstrak EPCIS to be used in a straightforward manner from Web languages such as JavaScript. Furthermore, resource constrained devices have difficulties accessing this type of interfaces [214] (see Section 2.5). This make it difficult for sensor nodes, embedded computers or even current mobile phones to access EPC traces and data.

To overcome these limitations, researchers started to create translation proxies between the EPCIS and their applications. In the *REST Binding* project [267] a translation proxy is implemented. The proxy offers URIs for accessing the EPCIS data but these data are provided using the XML format specified in the standard. While this is an important improvement, the proposed protocol does not respect the REST constraints but implements what is sometimes referred to as a REST-RPC style [160]. As the connectedness and uniform interface properties do not held, an EPCIS using this interface is not truly integrated to the Web [153, 160]. For instance, it does not offer alternative representations (e.g., JSON) and resources cannot be browsed for.

In [82] we presented an implementation of such a translation proxy. The *Mobile IoT Toolkit* offers a Java servlet based solution that allows to request some EPCIS data using URIs which are then translated by a proxy into WS-* calls. This solution is a step towards our goal as it enables resource-constrained clients such as mobile phones to access some data without the need for using WS-* libraries. Nevertheless, the proxy is directly built in the core of the Fosstrak EPCIS and thus does not offer a generic solution for all EPCIS compliant systems. Furthermore, the protocol used in this implementation as well as the data format is proprietary which requires developers to first learn it.

The EPCIS Webadapter builds upon this research. It offers an EPCIS vendor-independent module that implements a comprehensive Resource Oriented Architecture for the EPCIS and hence offers a RESTful API. It is further complemented by the tPusher service

and the Capture App Webadapter module that offers additional interfacing points for creating applications with the EPCIS. Finally, we illustrate how the Sharing Layer and the Composition Layer can be implemented to enable data sharing and physical mashups to leverage the potential of the EPC Network.

4.8 Discussion and Summary

In this chapter, we apply the Web of Things Architecture to the EPC Network with the goal of simplifying application development on top of this network. While working on the proposed approach we identified a number of real-world challenges and discuss three of them here.

Dealing with Firewalls and NATs First, while some standard LLRP readers offer a reader-initiated scheme, most operate on a server-initiated scheme. This means that the EPC Cloud server has to contact the RFID readers in order to start the reading process. While this works fine in places where a direct access to the Internet and the Web is available, it is problematic in industrial environments where RFID readers sit behind firewalls or NATs (Network Address Translation) and do not feature public IP addresses. This issue is not inherent to RFID readers but is a general issue when deploying WoT systems in the real-world and in particular in corporate environments.

A common practical solution to these problems is the use of the Reverse HTTP protocol [121] where a service on the Internet acts as a public proxy for devices behind firewalls and/or NATs on a private network [85]. In essence, the device has to initiate the connection with a `POST` to an Reverse HTTP proxy which will initiate a protocol switch to PTTH (Reverse HTTP). The proxy will then assign a URI within its domain to the device and forward the incoming requests directly to the device through a (device-initiated) kept alive channel. As an example, the open-source Yaler [288] project is providing a service implementing the Reverse HTTP protocol.

More specifically, in the case of the EPC Network, an option in the LLRP protocol called *reader-initiated* connection solves the problem if the readers manufacturers are implementing it consequently. Indeed, in this model, like in the Reverse HTTP model, the reader initiates the connection with the server (e.g., the EPC Cloud) upon startup and the connection is kept open until the reader shuts down again.

Leveraging the Cloud Furthermore, to optimize data access, most cloud infrastructures offer highly optimized storage services that can be easily distributed and load balanced within the infrastructure. In the Java world, the implementations of these services is compliant with the Java Persistence API (JPA) [251] or more recently with the Java Data Object (JDO) API [246] which abstract from the actual storage service being used and also allows to easily switch the service. Unfortunately, the current Fosstrak EPCIS is

not JDO or JPA compliant but uses JDBC and is rather tightly coupled with a MySQL database. Porting the EPCIS to JDO would allow to better leverage the scalability that cloud solutions and recent data stores (e.g., NoSQL databases) have to offer.

Moreover, for real-world applications, network delays might be a serious issue as events and actions are sent and triggered in the cloud. While it is unlikely that an EPC Cloud solution will support sub-second use-cases in the very near future, our average measurements have shown typical delays of about a second on average for the described EAS Mashup, from the reader, to the cloud and then to the mobile phone. While this is acceptable for most envisioned applications it strongly depends on the network configurations and infrastructure of real-world deployments. Hence, exhaustive evaluations of cloud solutions and their variations as well as models ensuring Quality of Service (QoS) requirements in smart things-to-cloud applications is an important part of the furture work.

Sharing RFID Data As illustrated in this chapter, the Sharing Layer of the Web of Things Architecture can be used to share EPC resources and feeds of events in a simple manner, based on social graphs instead of traditional access control lists which are hard to create, maintain and manage. However, the presented approach implies that companies use social networks for maintaining business to business relationships with other companies. It further implies that companies trust the social networks. While this is unlikely to happen with social networks like Facebook or Twitter, specialized networks such as LinkedIn will certainly play an important role as future platforms for Business to Business communication [180]. Hence, using the SAC architecture to manage access to EPC events through networks such as LinkedIn might be a viable future solution. It would, however, require for these networks to not only allow individuals to be connected together but also companies, a model that some social networks are moving forward to adopt [283].

Summary In this chapter we have shown how the Web of Things Architecture can be beneficial to the EPC Network and explained how virtualization, cloud computing, REST and the real-time Web, Social Networks as well as the concept of Physical Mashups can contribute to a wider adoption of the EPC Network standards and tools. Virtualization allows to package all the development tools into a single virtual machine that can be run virtually anywhere. Cloud computing simplifies deployment and maintenance of the EPC software stack. Thus, it pushes the standardized EPC Network closer to small and mid-size businesses that could benefit from it. By implementing the Device Accessibility Layer, we can offer more lightweight interfaces and allow innovative mobile, Web and WSN applications to directly use the EPC Network. We illustrated this with several mobile and Web prototypes. Furthermore, on top of this, the Sharing Layer can be used to share data amongst business partners. Finally, with the Composition Layer we offer a mashup editor and engine that allows more flexible real-world use-cases, where existing sensors and actuators can be directly integrated with RFID hardware from the Web and by end-users.

In order for the community to benefit from the novel applications this architecture enables,

the EPCIS Webadapter was added early 2011 as a module of the Fosstrak open-source project [235]. It already used in several projects, for instance to enable Android mobile phones to access EPCIS data.

Chapter 5

Conclusions and Outlook

In this last chapter we summarize the contribution of this thesis and discuss open issues and future challenges for the Web of Things.

5.1 Contributions

In this thesis we presented an architecture focusing on enabling a participatory WoT in which opportunistic applications can be easily created not only by embedded systems specialists but also by Web developers, tech-savvies and end-users. First, we described the Web of Things Architecture and its four layers:

- In the Device Accessibility Layer we addressed the application interfaces that smart things should expose to realize a seamless Web integration. We proposed taking a Resource Oriented approach (with extensions such as push support) and described a methodology to implement RESTful Web APIs on smart things either directly or through small modular software applications called Smart Gateways.
- In the Findability Layer we proposed a simple model to describe smart things using metadata implemented by re-using widespread standards such as microformats. We further described a discovery and lookup infrastructure that can, for example, be deployed alongside with Smart Gateways. This infrastructure discovers smart things at a Web layer and allows users to run distributed search queries to find adequate services to integrate in their composite applications. We further proposed extensions to the lookup infrastructure making searching for real-world services more efficient.
- In the Sharing Layer we proposed an innovative architecture for sharing smart things and their services by leveraging social networks as well as Web authentication and authorization protocols. The architecture also acts as a federation of social networks accessible through a single API that can be used by end-users, applications or smart things to access resources in a uniform and secure manner.

- In the Composition Layer we contributed to making Physical Mashups possible by showing how a simple mashup editor could be adapted to support smart things. We further proposed an architecture enabling the implementation of domain specific Physical Mashups.

The architecture was implemented in several independent but interoperable services and frameworks. Furthermore, we evaluated the implementations in two domains:

- We first studied Wireless Sensor Network platforms and applied the Web of Things Architecture to two of them: an energy sensing platform and a general purpose sensing platform.
- In a second phase, we applied the architecture to RFID and in particular to the EPC Network.

For each domain, we evaluated the implementation first empirically by means of several prototypes and applications, then in quantitative terms with performance studies. Finally in qualitative terms in a user-study of the development experience as well as reports of experience from external developers who used our open-source software. Overall the results demonstrate that the Web of Things Architecture can significantly simplify the development of applications for these platforms.

5.2 Discussion and Future Challenges

This thesis takes an exploratory approach to the Web integration of smart things. Rather than focusing on one particular problem we looked at the bigger picture of this integration and tried to understand and experience its implications. As a consequence, the thesis provides a holistic view of this emerging domain but also emphasizes on several challenges instrumental to the realization of the Web of Things.

Pushing Web and Internet Standards Forward First, although this thesis illustrates the suitability of Web standards and protocols for communicating real-world objects it also reveals their shortcomings. HTTP was designed as an architecture where clients initiate interactions and this model works fine for control-oriented WoT applications. However, monitoring-oriented applications are often event-based and thus smart things should also be able to push data to clients (rather than being continuously polled). Using syndication protocols such as Atom and AtomPub improves the model for monitoring applications, since devices can publish asynchronously data using AtomPub on an intermediate server. Nevertheless clients still have to pull data from Atom servers. Adapting the client-server architecture of the Web to more real-time use-cases is now a core research topic. A domain in which Internet and Web of Things researchers take an increasingly more important place. As a consequence, standards such as HTML5 are moving towards asynchronous bi-directional communication, e.g., with the Server Sent Events draft [271] or HTML5 WebSockets [282] upon which we proposed a solution for the real-time WoT.

These initiatives emphasize on how relevant it is to further work on lightweight Web-based messaging systems.

Furthermore, while the presented Smart Gateway and LLDU approach for the integration of highly constrained devices has a number of advantages (e.g., scalability, caching, discovery and lookup services, device management, etc.), it also introduces application level (software) gateways which complexifies WoT deployments. Hence, research on optimizing Internet and Web protocols for resource constrained devices is highly relevant. Projects such as 6LoWPAN, which adapt the IPv6 protocols to low power small footprint radio networks [148, 99] are important research efforts towards this direction. Closer to the application layer, initiatives such as CoAP (Constrained Application Protocol) [176] propose a new application protocol borrowing the core concepts of the Web (e.g., REST and HTTP) to better meet the needs of very constrained devices. While taking the approach of building alternative (Web) architectures to achieve fine-tuned optimizations might not be required anymore in a couple of years, these initiatives push forward the efforts towards end-to-end IP and Web networks of smart things.

Deploying the Web of Things In this thesis particular care was given to test the proposed architectures by building concrete prototypes and evaluating them with actual devices. However, most of these prototypes were deployed and evaluated in a lab environment (except for the Energie Visible prototype presented in Chapter 3). More generally, there is a significant lack of large-scale real-world deployments for the Web of Things, perhaps because the WoT research community as defined in this thesis has only recently emerged. However, the vision behind the Web of Things is to implement a global network of smart things. Hence, future work should also focus on larger deployments of the developed concepts and technologies that will certainly raise challenging issues but also perhaps make an even stronger point for using Web standards. Efforts should also be made to bring these technologies closer to real-world use-cases and to the business. Towards this aim we open-sourced several of the software components that were presented in this thesis and they are increasingly being used by third-parties to implement their particular use-cases. In this direction, strategic alliances such as the IPSO (IP for Smart Objects) [244] work on the industrial dissemination of the Internet and Web of Things and emphasize the relevance of these topics outside of academic research.

Freeing the Social WoT The recent emergence of a Social Web of Things offers some unprecedented opportunities to use social connections and their underlying social graphs to share digital artifacts and make them more socially aware. The Sharing Layer of this thesis only scratches the surface of applications enabled by bridging the gap between social networks and networks of objects. Object to object communication, objects to people communication and actuation of the physical world based on processing social event streams are just a few examples that are increasingly being explored. However, WoT applications leveraging social networks also face important challenges. First, applications built using social network APIs are also strongly coupled with the social network

platforms, and so are their users. Additionally, social networks APIs offer different functionalities, therefore it is difficult to group them under a single common denominator. Several initiatives attempt to solve these problems such as the OpenSocial standards presented before. Initially started by Google, many social networks such as Orkut, Linkedin, Netlog, Yahoo!, Hi5, Myspace and many others have also joined and implemented the standards in their APIs. However, while the initiative has a lot of potential, the current implementations are not entirely homogeneous yet and still under construction for most of them. Moreover, some of the major social networks do not comply with OpenSocial as of today, most likely due to the strategic implications of such standards [291].

Once smart things increasingly blend with social networks, these challenges will have important consequences, and hence it is highly relevant for researchers to work on preventing users from social networks lock-in. Our implementation of the Social Access Controller supporting several social networks and offering a unified access to their basic data is a step towards this direction but research should further explore the notion of meta social networks for the WoT. Furthermore, ensuring portability of users or things-generated data will be a significant and necessary step to move towards a truly open and interoperable Social Web of Things.

Increasing the Intelligence The Findability Layer presented in this thesis is a first step towards more intelligence on top of the WoT. However, it raises a number of issues and paves the way for future work. First, the presented approach enables users to search for smart things as well as the automatic generation of mashup building-blocks. It does not, however, enable a complete and dynamic thing to thing service discovery where smart things can use and understand each others services in an entirely automated way. The automatic mashability of the physical world is thus yet to be implemented and has been on the agenda of researchers for several years already.

Furthermore, this thesis and most of the current research in the WoT has been focusing on accessibility: Making smart things accessible and enabling cross-integration with other devices and services on the Web. One of the natural follow-ups of this research is intelligence and reasoning: Given the fact that smart things sense the physical world, how do we develop, in an open and loosely-coupled way, frameworks, languages and algorithms that extract meaning from – and react upon – these valuable streams of sensed data?

Final Thoughts This thesis illustrates how introducing support for Web standards at the device-level (in a direct or Smart Gateways mediated way) is beneficial for developing a new generation of networked devices that are much simpler to deploy, program, and reuse. We illustrate how applying the design principles that supported the success of the Web and in particular openness, connectedness and simplicity can significantly ease the development process on top of smart things. Thanks to the wide-spread deployments of Web browsers (e.g., in desktop computers, mobile phones, machines, modern home appliances, etc.), and to the ubiquitous HTTP support in programming and scripting

languages, we tap into very large communities (e.g., Web developers) as potential application developers for the WoT. Furthermore, with Physical Mashups we demonstrated how tech-savvies and end-users are given the power to develop small but tailored applications on top of smart things. As a consequence, we believe that the Web of Things Architecture has the potential to foster open public innovation, leading to an increasing number of interesting applications involving smart things.

In this thesis, we demonstrate the fact that the Web of Things is interesting not only because it forces smart things to all understand the same basic and interoperable standards but also because it significantly eases the direct integration of smart things with an impressive number of services on the Web: Composition tools, visualization APIs, distributed data-stores, app-stores, cloud infrastructures, social networks, (micro) blogging services, search engines, etc. Through the presented architecture smart things become seamlessly part of the programmable, real-time, semantic and social Web.

Bibliography

[1] K. Aberer, M. Hauswirth, and A. Salehi. Infrastructure for Data Processing in Large-Scale Interconnected Sensor Networks. In *Proc. of the International Conference on Mobile Data Management*, pages 198–205, 2007.

[2] W. Abrahamse, L. Steg, C. Vlek, and T. Rothengatter. A review of intervention studies aimed at household energy conservation. *Journal of Environmental Psychology*, 25(3):273–291, 2005.

[3] Robert Adelmann, Marc Langheinrich, and Christian Floerkemeier. A Toolkit for Bar-Code-Recognition and -Resolving on Camera Phones Jump Starting the Internet of Things. In *Proc. of the workshop on Mobile and Embedded Interactive Systems (MEIS'06) at Informatik 2006. GI Lecture Notes in Informatics Series (LNI)*, Dresden, Germany, 2006.

[4] Azucena Guillen Aguilar. *Exploring Physical Mashups on Mobile Devices.* Master thesis, ETH Zurich, Switzerland, 2010.

[5] Ryan Aipperspach, Ben Hooker, and Allison Woodruff. The heterogeneous home. *Interactions*, 16:35–38, January 2009.

[6] I. F Akyildiz, Weilian Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102– 114, August 2002.

[7] Rosa Alarcón and Erik Wilde. RESTler: crawling RESTful services. In *Proc. of the 19th international conference on World Wide Web (WWW '10)*, pages 1051–1052, New York, NY, USA, 2010. ACM.

[8] Subbu Allamaraju. *RESTful Web Services Cookbook.* Yahoo Press, March 2010.

[9] John Allsopp. *Microformats: Empowering Your Markup for Web 2.0.* friendsofED, March 2007.

[10] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications.* Springer, December 2010.

[11] W.T. Balke and M. Wagner. Through different eyes: assessing multiple conceptual views for querying web services. In *Proc. of the 13th international World Wide Web conference (WWW '04)*, pages 196–205, New York, NY, USA, 2004. ACM.

[12] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36:81–94, November 1993.

[13] M. Baqer and A. Kamal. S-Sensors: Integrating physical world inputs with social networks using wireless sensor networks. In *Proc. of the 5th International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP 2009)*, pages 213–218, Melbourne, Australia, 2009.

[14] John Barton, Tim Kindberg, Hui Dai, Nissanka B Priyantha, and Fahd Al-bin ali. Sensor-enhanced mobile web clients: an XForms approach. In *Proc. of the 12th international conference on World Wide Web (WWW '03)*, pages 80–89, Budapest, Hungary, 2003. ACM.

[15] Michael Blackstock and Adrian Friday. Uniting Online Social Networks with Places and Things. In Dominique Guinard, Vlad Trifa, and Erik Wilde, editors, *Proc. of the 2nd International Workshop on the Web of Things (WoT 2011)*, San Fransisco, USA, 2011. ACM.

[16] Miodrag Bolic and David Simplot-Ryl. *RFID Systems: Research Trends and Challenges*. Wiley, September 2010.

[17] Rachel Botsman and Roo Rogers. *What's Mine Is Yours: The Rise of Collaborative Consumption*. HarperBusiness, September 2010.

[18] Mike Botts and Alex Robin. OpenGIS Sensor Model Language (SensorML) Implementation Specification, 2007.

[19] Danah M. Boyd and Nicole B. Ellison. Social Network Sites: Definition, History, and Scholarship. *Journal of ComputerMediated Communication*, 13(1):210–230, October 2008.

[20] J. Brandt, P.J. Guo, J. Lewenstein, S.R. Klemmer, and M. Dontcheva. Writing Code to Prototype, Ideate, and Discover. *IEEE Software*, 26(5):18–24, 2009.

[21] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *Proc. of the seventh international conference on World Wide Web, WWW '98*, pages 107–117, Brisbane, Australia, 1998. Elsevier Science Publishers B. V.

[22] Andreas Brodt and Daniela Nicklas. The TELAR mobile mashup platform for Nokia internet tablets. In *Proc. of the 11th international conference on Extending DataBase Technology: Advances in database technology (EDBT '08)*, pages 700–704, Nantes, France, 2008. ACM.

[23] Gregor Broll, John Hamard, Massimo Paolucci, Markus Haarländer, Matthias Wagner, Sven Siorpaes, Enrico Rukzio, Albrecht Schmidt, and Kevin Wiesner. Mobile interaction with web services through associated real world objects. In *Proc. of the 9th international conference on Human computer interaction with mobile devices and services (MobileHCI '07)*, pages 319–321, New York, NY, USA, 2007. ACM.

[24] Gregor Broll, Sven Siorpaes, Enrico Rukzio, Massimo Paolucci, John Hamard, Matthias Wagner, and Albrecht Schmidt. Supporting Mobile Service Usage through Physical Mobile Interaction. In *Fifth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom '07)*, pages 262–271. Ieee, 2007.

[25] J. Bungo. Embedded Systems Programming in the Cloud: A Novel Approach for Academia. *IEEE Potentials*, 30(1):17–23, 2011.

[26] T. Burbridge and M. Harrison. Security considerations in the design and peering of RFID Discovery Services. In *Proc. of the IEEE International Conference on RFID*, pages 249–256. IEEE, April 2009.

[27] Davide Carboni and Pietro Zanarini. Wireless wires: let the user build the ubiquitous computer. In *Proc. of the 6th international conference on Mobile and Ubiquitous Multimedia (MUM '07)*, pages 169–175, Oulu, Finland, 2007. ACM.

[28] Gong Chen, Nathan Yau, Mark Hansen, and Deborah Estrin. Sharing Sensor Network Data. Technical report, UC Los Angeles, 2007.

[29] Marshini Chetty, David Tran, and Rebecca E Grinter. Getting to green: understanding resource consumption in the home. In *Proc. of the 10th International Conference on Ubiquitous Computing (UbiComp '08)*, pages 242–251, Seoul, Korea, 2008. ACM.

[30] Paul Couderc. Spreading the web. In Marco Conti, Silvia Giordano, Enrico Gregori, and Stephan Olariu, editors, *Personal Wireless Communications*, volume 2775 of *LNCS*, pages 375–384. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[31] Marco Crasso, Alejandro Zunino, and Marcelo Campo. Easy web service discovery: A query-by-example approach. *Science of Computer Programming*, 71(2):144–164, April 2008.

[32] Fred D. Davis. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly*, 13(3):319–340, 1989.

[33] Stephen Dawson-Haggerty, Xiaofan Jiang, and Gilman Tolle. sMAP: a simple measurement and actuation profile for physical information. In *Proc. of the 8th ACM Conference on Embedded Networked Sensor Systems (Sensys '10)*, pages 197–210, New York, NY, USA, 2010. ACM.

[34] J.D. Day and H. Zimmermann. The OSI reference model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983.

[35] S. de Deugd, R. Carroll, K.E. Kelly, B. Millett, and J. Ricker. SODA: Service Oriented Device Architecture. *Pervasive Computing, IEEE*, 5(3):94–96, 2006.

[36] L. de Souza, P. Spiess, D. Guinard, M. Köhler, S. Karnouskos, and D. Savio. Socrades: A web service based shop floor integration infrastructure. In Christian Floerkemeier, Marc Langheinrich, Elgar Fleisch, Friedemann Mattern, and Sanjay E. Sarma, editors, *Proc. of the Internet of Things Conference (IoT '08)*, LNCS, pages 50–67, Zurich, Switzerland, March 2008. Springer Berlin Heidelberg.

[37] Robert Dickerson, Jiakang Lu, Jian Lu, and Kamin Whitehouse. Stream Feeds - An Abstraction for the World Wide Sensor Web. In *Proc. of the Internet of Things Conference (IoT '08)*, LNCS, pages 360–375, Zurich, Switzerland, 2008. Springer Berlin Heidelberg.

[38] Bettina Dober. *Exploring Query Augmentation for the SOCRADES Application Service Catalogue*. Master thesis, University of Fribourg, Switzerland, 2009.

[39] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin. pREST: a REST-based protocol for pervasive systems. In *Proc. of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, pages 340–348. IEEE, 2004.

[40] Adam Dunkels. Full TCP/IP for 8-bit architectures. In *Proc. of the 1st international conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 85–98, New York, NY, USA, 2003. ACM.

[41] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. Consistency and scalability in event notification for embedded Web applications. In *Proc. of the 11th IEEE International Symposium on Web Systems Evolution (WSE '09)*, pages 89–98. IEEE, 2009.

[42] Simon Duquennoy, Gilles Grimaud, and J.J. Vandewalle. The Web of Things: interconnecting devices with high usability and performance. In *Proc. of the International Conference on Embedded Software and Systems*, pages 323–330, HangZhou, Zhejiang, China, May 2009. IEEE.

[43] Charles Engelke and Craig Fitzgerald. Replacing legacy web services with RESTful services. In *Proc. of the First International Workshop on RESTful Design (WS-REST '10)*, pages 27–30, Raleigh, North Carolina, 2010. ACM.

[44] EPCglobal. EPC Information Services (EPCIS) Version 1.0.1 Specification. Technical report, GS1, 2007.

[45] EPCglobal. The Application Level Events (ALE) Specification ver 1.1.1. Technical report, GS1, 2009.

[46] EPCglobal. EPC Tag Data Standard. Technical report, GS1, 2010.

[47] EPCglobal. Low Level Reader Protocol (LLRP) ver. 1.1. Technical report, GS1, 2010.

[48] Eran Hammer-Lahav. The OAuth 1.0 Protocol. Technical report, April 2010.

[49] Thomas Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall, April 2004.

[50] R. Fielding. *Architectural styles and the design of network-based software architectures*. Phd thesis, 2000.

[51] R Fielding, J Gettys, J C Mogul, H Frystyk, L Masinter, P Leach, and Tim Berners-Lee. Hypertext Transfer Protocol-HTTP/1.1. Technical report, World Wide Web Consortium, 1999.

[52] Klaux Finkenzeller and D. Mueller. *RFID Handbook: Fundamentals and Applications in Contactless Smart Cards, Radio Frequency Identification and Near-Field Communication*. Wiley, second edition, 2010.

[53] C. Fischer. Feedback on household electricity consumption: a tool for saving energy? *Journal of Energy Efficiency*, 1(1):79–104, 2008.

[54] Christian Floerkemeier. EPC-Technologie vom Auto-ID Center zu EPCglobal. In Elgar Fleisch and Friedemann Mattern, editors, *Das Internet der Dinge*, pages 87–100. Springer-Verlag, 2005.

[55] Christian Floerkemeier. Integrating RFID Readers in the Enterprise IT Overview of Intra-organizational RFID System Services and Architectures. In *Integrating RFID Readers in Enterprise IT*, pages 269–295. John Wiley & Sons, Ltd, 2010.

[56] Christian Floerkemeier and Matthias Lampe. Facilitating RFID development with the accada prototyping platform. In *Fifth IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom '07)*, pages 495–500, New York, NY, USA, 2007. IEEE Computer Society.

[57] Christian Floerkemeier, Christof Roduner, and Matthias Lampe. RFID application development with the Accada middleware platform. *IEEE Systems Journal*, 1(2):82–94, December 2007.

[58] C. Frank, P. Bollinger, C. Roduner, and W. Kellerer. Objects Calling Home: Locating Objects Using Mobile Phones. In *Proc. of the International Conference on Pervasive Computing (Pervasive '07)*, Toronto, Canada, 2007.

[59] Christian Frank, Philipp Bolliger, Friedemann Mattern, and Wolfgang Kellerer. The Sensor Internet at Work: Locating Everyday Items Using Mobile Phones. *Pervasive and Mobile Computing*, 4(3):421–447, June 2008.

[60] Jon Froehlich. Promoting Energy Efficient Behaviors in the Home through Feedback : The Role of Human-Computer Interaction. *Computing Systems*, 9:10, 2009.

[61] Patrik Fuhrer and Dominique Guinard. Building a Smart Hospital using RFID Technologies. In *Proc. of the European Conference on eHealth (ECEH)*, Fribourg, Switzerland, October 2006.

[62] Erich Gamma, Richard Helm, Ralph Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*, volume 206. Addison-wesley Reading, MA, November 1995.

[63] David Gefen and Mark Keil. The impact of developer responsiveness on perceptions of usefulness and ease of use: an extension of the technology acceptance model. *SIGMIS Database*, 29:35–49, April 1998.

[64] Hans Gellersen, Carl Fischer, and Dominique Guinard. Supporting device discovery and spontaneous interaction with spatial references. *Journal of Personal and Ubiquitous Computing (PUC)*, 2009.

[65] N. Gershenfeld and D. Cohen. Internet 0: Interdevice Internetworking - End-to-End Modulation for Embedded Networks. *Circuits and Devices Magazine, IEEE*, 22(5):48–55, 2006.

[66] Antonio Goncalves. *Beginning Java EE 6 with GlassFish 3*. Apress, 2 edition, August 2010.

[67] W. I Grosky, A. Kansal, S. Nath, Jie Liu, and Feng Zhao. SenseWeb: An Infrastructure for Shared Sensing. *IEEE Multimedia*, 14(4):8–13, December 2007.

[68] Dominique Guinard. Mashing up your web-enabled home. In *Proc. of the 10th International conference on Current trends in Web Engineering (ICWE '10)*, pages 442–446, Vienna, Austria, July 2010. Springer-Verlag.

[69] Dominique Guinard, Oliver Baecker, and Florian Michahelles. Supporting a mobile lost and found community. In *Proc. of the 10th international conference on Human Computer Interaction with mobile devices and services (Mobile HCI '08)*, pages 407–410. ACM, September 2008.

[70] Dominique Guinard, Oliver Baecker, Patrik Spiess, Stamatis Karnouskos, Moritz Koehler, Luciana Moreira Sa De Souza, Domnic Savio, and Mihai Vlad Trifa. On-demand provisionning of services running on embedded devices, Patent, 2010.

[71] Dominique Guinard, Mathias Fischer, and Vlad Trifa. Sharing using social networks in a composable web of things. In *Proc. of the First IEEE International Workshop on the Web of Things (WoT 2010)*, pages 702–707, Mannheim, Germany, March 2010. IEEE.

[72] Dominique Guinard, Christian Floerkemeier, and Sanjay Sarma. Cloud Computing, REST and Mashups to Simplify RFID Application Development and Deployment. In *Proc. of the 2nd International Workshop on the Web of Things (WoT 2011)*, San Fransisco, USA, June 2011. ACM.

[73] Dominique Guinard, Iulia Ion, and Simon Mayer. In Search of an Internet of Things Service Architecture: REST or WS-*? A Developers' Perspective. In *Proc. of Mobiquitous 2011 (8th International ICST Conference on Mobile and Ubiquitous Systems).*, Copenhagen, Denmark, 2011.

[74] Dominique Guinard, Mathias Mueller, and Jacques Pasquier. Giving RFID a REST: Building a Web-Enabled EPCIS. In *Proc. of the International conference on the Internet of Things (IoT '10)*, LNCS, Tokyo, Japan, November 2010. Springer Berlin / Heidelberg.

[75] Dominique Guinard, Mathias Mueller, and Vlad Trifa. *RESTifying Real-World Systems: a Practical Case Study in RFID*, chapter 16. Springer, August 2011.

[76] Dominique Guinard and Vlad Trifa. Towards the web of things: Web mashups for embedded devices. In *Proc. of the Second Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM '09)*, WWW '09, Madrid, Spain, April 2009. ACM.

[77] Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess, and Domnic Savio. 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):223–235, February 2010.

[78] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices. In Dieter Uckelmann, Mark Harrison, and Florian Michahelles, editors, *Architecting the Internet of Things*, pages 97–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[79] Dominique Guinard, Vlad Trifa, Thomas Pham, and Olivier Liechti. Towards physical mashups in the web of things. In *Proc. of the 6th International Conference on Networked Sensing Systems (INSS '09)*, pages 1–4, Pittsburgh, USA, June 2009. IEEE.

[80] Dominique Guinard, Vlad Trifa, Patrik Spiess, Bettina Dober, and Stamatis Karnouskos. Discovery and On-Demand Provisionning of Real-World Web Services. In *Proc. of the IEEE International Conference on Web Services (ICWS '09)*, Los Angeles, California, USA, July 2009.

[81] Dominique Guinard, Vlad Trifa, and Erik Wilde. A Resource Oriented Architecture for the Web of Things. In *Proc. of the 2nd International Conference on the Internet of Things (IoT 2010)*, LNCS, Tokyo, Japan, November 2010. Springer Berlin / Heidelberg.

[82] Dominique Guinard, Felix von Reischach, and Florian Michahelles. MobileIoT Toolkit: Connecting the EPC Network to MobilePhones. In *Proceedings of Mobile Interaction with the Real World (MIRW '08)*, Amsterdam, Netherlands, September 2008. The University of Oldenburg.

[83] Dominique Guinard, Markus Weiss, and Vlad Trifa. Are you energy-efficient? sense it on the web. In *Adjunct Proc. of the International Conference on Pervasive Computing (Pervasive '09)*, Nara, Japan, May 2009. Springer.

[84] V. Gupta, P. Udupi, and A. Poursohi. Early lessons from building Sensor.Network: an open data exchange for the web of things. In *Proc. of the First International Workshop on the Web of Things (WoT 2010)*, pages 738–744. IEEE, 2010.

[85] Vipul Gupta, Ron Goldman, and Poornaprajna Udupi. A network architecture for the Web of Things. In *Proc. of the Second International Workshop on Web of Things (WoT 2011)*, WoT '11, pages 3:1–3:6, San Francisco, California, 2011. ACM.

[86] Marc J Hadley. Web application description language (WADL). Technical report, Sun Microsystems, Inc., 2006.

[87] Richard Hall, Karl Pauls, Stuart McCulloch, and David Savage. *OSGi in Action: Creating Modular Applications in Java*. Manning Publications, April 2011.

[88] Mark Halvorson and Andy Smith. OpenSocial 2.0 Specification. Technical report, Open Social Working Group, 2010.

[89] Haodong Wang, C. C Tan, and Qun Li. Snoogle: A Search Engine for Pervasive Environments. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1188–1202, August 2010.

[90] B. Hartmann and S. Doorley. Hacking, mashing, gluing: understanding opportunistic design. *IEEE Pervasive Computing*, 7(3):46–54, 2008.

[91] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, and E. Jansen. The Gator Tech Smart House: a programmable pervasive space. *Computer*, 38(3):50–60, 2005.

[92] Johannes Helander. Deeply embedded XML communication: towards an interoperable and seamless world. In *Proc. of the 5th ACM international conference on embedded software (EMSOFT '05)*, pages 62–67, Jersey City, NJ, USA, 2005. ACM.

[93] M. Hepp, K. Siorpaes, and D. Bachlechner. Harvesting Wiki Consensus: Using Wikipedia Entries as Vocabulary for Knowledge Management. *IEEE Internet Computing,*, 11(5):54–65, 2007.

[94] Soojung Hong. *Mobile Discovery in a Web of Things*. Master thesis, ETH Zurich, Switzerland, 2010.

[95] A. Hornsby and E. Bail. μXMPP: Lightweight implementation for low power operating system Contiki. In *International Conference on Ultra Modern Telecommunications (ICUMT '09)*, pages 1–5. IEEE, October 2009.

[96] A. Hornsby and R. Walsh. From instant messaging to cloud computing, an XMPP review. pages 1–6. IEEE, June 2010.

[97] Volker Hoyer, Katarina Stanoesvka-Slabeva, Till Janner, and Christoph Schroth. Enterprise Mashups: Design Principles towards the Long Tail of User Needs. In *IEEE International Conference on Services Computing (SCC '08)*, volume 2, pages 601–602, 2008.

[98] J.W. Hui and D.E. Culler. Extending IP to low-power, wireless personal area networks. *IEEE Internet Computing*, 12(4):37–45, 2008.

[99] J.W. Hui and D.E. Culler. IP is dead, long live IP for wireless sensor networks. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 15–28, Raleigh, NC, USA, 2008. ACM.

[100] Jonathan J Hull, Xu Liu, Berna Erol, Jamey Graham, and Jorge Moraleda. Mobile image recognition: architectures and tradeoffs. In *Proc. of the Eleventh Workshop on Mobile Computing Systems & Applications (HotMobile '10)*, pages 84–88, New York, NY, USA, 2010. ACM.

[101] Iulia Ion, Marc Langheinrich, Ponnurangam Kumaraguru, and Srdjan Čapkun. Influence of user perception, security needs, and social factors on device pairing method choices. In *Proc. of SOUPS 2010*, SOUPS '10, pages 6:1–6:13, Redmond, Washington, USA, 2010. ACM.

[102] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. Technical report, IETF, 1999.

[103] F. Jammes and H. Smit. Service-oriented paradigms in industrial automation. *IEEE Transactions on Industrial Informatics*, 1(1):62–70, 2005.

[104] François Jammes, Antoine Mensch, and Harm Smit. Service-oriented device communications using the devices profile for web services. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, MPAC '05, pages 1–8, New York, NY, USA, 2005. ACM.

[105] Xiaofan Jiang, Stephen Dawson-Haggerty, Prabal Dutta, and David Culler. Design and implementation of a high-fidelity AC metering network. In *Proc. of the 2009 International Conference on Information Processing in Sensor Networks (IPSN '09)*, pages 253–264. IEEE Computer Society, 2009.

[106] A. Juels. RFID security and privacy: a research survey. *IEEE Journal on Selected Areas in Communications*, 24(2):381–394, February 2006.

[107] Andreas Kamilaris, Nicolas Iannarilli, Vlad Trifa, and Andreas Pitsillides. Bridging the Mobile Web and the Web of Things in Urban Environments. In *Proc. of the first Workshop on the Urban Internet of Things (UrbanIoT '10)*, Tokyo, Japan, 2010.

[108] S Karnouskos, D Savio, P Spiess, D Guinard, V Trifa, and O Baecker. Real-world Service Interaction with Enterprise Systems in Dynamic Manufacturing Environments. In Lyes Benyoucef and Bernard Grabot, editors, *Artificial Intelligence Techniques for Networked Manufacturing Enterprises Management*, Springer Series in Advanced Manufacturing, pages 423–457. Springer London, 2010.

[109] A. Katasonov and M. Palviainen. Towards ontology-driven development of applications for smart environments. In *Proc. the International Conference on Pervasive*

Computing and Communications Workshops (PERCOM Workshops 2010), pages 696–701. IEEE, April 2010.

[110] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, and Others. People, places, things: Web presence for the real world. *Mobile Networks and Applications*, 7(5):365–376, 2002.

[111] Jacek Kopecký, Karthik Gomadam, and Tomas Vitvar. hRESTS: An HTML Microformat for Describing RESTful Web Services. pages 619–625. IEEE Computer Society, 2008.

[112] Tiiu Koskela, Kaisa Väänänen-Vainio-Mattila, and Lauri Lehti. Home Is Where Your Phone Is: Usability Evaluation of Mobile Phone UI for a Smart Home. In Stephen Brewster and Mark Dunlop, editors, *Mobile Human-Computer Interaction MobileHCI 2004*, volume 3160 of *LNCS*, pages 74–85. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[113] Mathias Kovatsch, Markus Weiss, and Dominique Guinard. Embedding internet technology for home automation. In *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pages 1–8, Bilbao, Spain, September 2010. IEEE.

[114] D. Kristol and L. Montulli. HTTP State Management Mechanism. Technical report, 1997.

[115] Kirk L Kroeker. The evolution of virtualization. *Communications of the ACM*, 52:18–20, March 2009.

[116] Roland Kübert, Gregory Katsaros, and Tinghe Wang. A RESTful implementation of the WS-agreement specification. In *Proc. of the Second International Workshop on RESTful Design (WS-REST '11)*, WS-REST '11, pages 67–72, New York, NY, USA, 2011. ACM.

[117] Chris Kürschner, Cosmin Condea, Oliver Kasten, and Frédéric Thiesse. Discovery Service Design in the EPCglobal Network. In Christian Floerkemeier, Marc Langheinrich, Elgar Fleisch, Friedemann Mattern, and Sanjay E. Sarma, editors, *Proc. of the Internet of Things Conference (IoT '08)*, volume 4952 of *LNCS*, pages 19–34, Zurich, Switzerland, 2008. Springer Berlin Heidelberg.

[118] Marc Langheinrich. *Personal Privacy in Ubiquitous Computing – Tools and System Support*. PhD thesis, ETH Zurich, Zurich, Switzerland, May 2005.

[119] Jon Lathem, Karthik Gomadam, and Amit P Sheth. SA-REST and (S)mashups: Adding Semantics to RESTful Services. In *Proc. of the International Conference on Semantic Computing (ICSC '07)*, pages 469–476. IEEE Computer Society, 2007.

[120] J.S. Lee, Y.W. Su, and C.C. Shen. A comparative study of wireless protocols: Bluetooth, UWB, ZigBee, and Wi-Fi. In *Proc. of the 33rd Annual Conference of the IEEEIndustrial Electronics Society (IECON 2007)*, pages 46–51. IEEE, 2007.

[121] Mark Lentczner and Donovan Preston. Reverse HTTP. Technical report, IETF, March 2009.

[122] Joshua Lifton, Mark Feldmeier, Yasuhiro Ono, Cameron Lewis, and Joseph A Paradiso. A platform for ubiquitous sensor deployment in occupational and domestic environments. In *Proc. of the 6th international conference on Information processing in sensor networks (IPSN '07)*, pages 119–127, Cambridge, Massachusetts, USA, 2007. ACM.

[123] Tao Lin, Hai Zhao, Jiyong Wang, Guangjie Han, and Jindong Wang. An Embedded Web Server for Equipments. In *Proc. of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'04)*, Hong Kong, Hong Kong, 2004.

[124] Alexander Linden, Jackie Fenn, David W McCoy, David W Cearley, Nikos Drakos, Jim Tully, Monica Basso, Phillip Redman, Erik Dorr, Allen Weiner, Kenshi Tazaki, Ant Allan, Rafe John Graham Ball, Jim Sinur, Michael A Silver, Martin Gilliland, Carl Claunch, Ray Valdes, Betsy Burton, Jeff Woods, Nick Jones, Jeffrey Mann, Whit Andrews, John Pescatore, Christophe Uzureau, Mary Knox, Rita E Knox, Van L Baker, Mike McGuire, Leslie Fiering, Christopher Ambrose, Steve Cramoysan, Bern Elliot, Bob Hafner, Yefim V Natis, Benoit J Lheureux, Howard J Dresner, Michael J Blechar, Brian Gammage, and Mark A Margevicius. Hype Cycle for Emerging Technologies 2005, August 2005.

[125] Peter Lubbers, Brian Albers, and Frank Salim. *Pro HTML 5 Programming.* Apress, March 2010.

[126] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim. TinyREST: A protocol for integrating sensor networks into the internet. In *Proc. of REALWSN*, Stockholm, Sweden, 2005. Citeseer.

[127] Liqian Luo, Aman Kansal, Suman Nath, and Feng Zhao. Sharing and exploring sensor streams over geocentric interfaces. In *Proc. of the 16th ACM SIGSPATIAL international conference on advances in geographic information systems (GIS '08)*, pages 1–10, Irvine, California, 2008. ACM.

[128] Urs Maeder. *Smart Lost & Found Services for Insurers.* Master thesis, University of St-Gallen, Switzerland, 2007.

[129] M. Marin-Perianu, N. Meratnia, P. Havinga, L.M.S. de Souza, J. Muller, P. Spieß, S. Haller, T. Riedel, C. Decker, and G. Stromberg. Decentralized enterprise systems: a multiplatform wireless sensor network approach. *IEEE Wireless Communications*, 14(6):57–66, 2007.

[130] Toshiyuki Masui, Koji Tsukada, and Itiro Siio. MouseField: A Simple and Versatile Input Device for Ubiquitous Computing. In Nigel Davies, Elizabeth D. Mynatt, and

Itiro Siio, editors, *Proc. of the Conference on Ubiquitous Computing (Ubicomp '04)*, volume 3205 of *LNCS*, pages 319–328, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[131] Mathias Fischer. *Social Access Controller: Sharing, Protecting and Publishing RESTful Resources through Social Networks*. Semester thesis, ETH Zurich, Zurich, Switzerland, July 2009.

[132] Kieran Mathieson, Eileen Peacock, and Wynne W Chin. Extending the technology acceptance model: the influence of perceived user resources. *ACM SIGMIS Database*, 32:86–112, July 2001.

[133] Friedemann Mattern. Die technische Basis für das Internet der Dinge. In Elgar Fleisch and Friedemann Mattern, editors, *Das Internet der Dinge*, pages 39–66. Springer-Verlag, 2005.

[134] Friedemann Mattern and Christian Floerkemeier. From the Internet of Computers to the Internet of Things. In Kai Sachs, Ilia Petrov, and Pablo Guerrero, editors, *Active Data Management to Event-Based Systems and More*, volume 6462 of *LNCS*, pages 242–259. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010.

[135] Friedemann Mattern, Thorsten Staake, and Markus Weiss. ICT for green: how computers can help us to conserve energy. In *Proc. of the 1st International Conference on Energy-Efficient Computing and Networking (e-Energy '10)*, e-Energy '10, pages 1–10, Passau, Germany, 2010. ACM.

[136] E. Michael Maximilien. Mobile Mashups: Thoughts, Directions, and Challenges. In *Proc. of the IEEE International Conference on Semantic Computing*, pages 597–600, 2008.

[137] Simon Mayer. *Deployment Support for an Infrastructure for Web-enabled Devices*. Master thesis, ETH Zurich, Switzerland, 2010.

[138] Simon Mayer and Dominique Guinard. An Extensible Discovery Service for Smart Things. In *Proc. of the 2nd International Workshop on the Web of Things (WoT 2011).*, San Fransisco, USA, June 2011. ACM.

[139] Michelle L Mazurek, J. P Arsenault, Joanna Bresee, Nitin Gupta, Iulia Ion, Christina Johns, Daniel Lee, Yuan Liang, Jenny Olsen, Brandon Salmon, Richard Shay, Kami Vaniea, Lujo Bauer, Lorrie Faith Cranor, Gregory R Ganger, and Michael K Reiter. Access Control for Home Data Sharing: Attitudes, Needs and Practices. In *Proc. of the 28th international conference on Human factors in computing systems (CHI 2010)*, CHI '10, pages 645–654, New York, NY, USA, 2010. ACM.

[140] T. Mikkonen and A. Salminen. Towards Pervasive Mashups in Embedded Devices. In *Proc. of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'10)*, pages 35–42, 2010.

[141] Andrew D Miller and W. Keith Edwards. Give and take: a study of consumer photo-sharing culture and practice. In *Proc. of the SIGCHI conference on Human factors in computing systems*, CHI '07, pages 347–356, New York, NY, USA, 2007. ACM.

[142] Richard Monson-Haefel. *J2EE Web Services: XML SOAP WSDL UDDI WS-I JAX-RPC JAXR SAAJ JAXP*. Addison-Wesley Professional, October 2003.

[143] Richard Monson-Haefel and Bill Burke. *Enterprise JavaBeans 3.0*. O'Reilly Media, 5 edition, May 2006.

[144] Peter Morville. *Ambient Findability: What We Find Changes Who We Become*. O'Reilly Media, October 2005.

[145] Peter Morville and Louis Rosenfeld. *Information Architecture for the World Wide Web: Designing Large-Scale Web Sites*. O'Reilly Media, third edit edition, December 2006.

[146] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, 43(3):19:1–19:51, April 2011.

[147] Mathias Müller. *Design and Implementation of a Web-enabled Electronic Product*. Master thesis, University of Fribourg, 2009.

[148] Geoff Mulligan. The 6LoWPAN architecture. In *Proc. of the 4th workshop on Embedded networked sensors (EmNets '07)*, EmNets '07, pages 78–82, Cork, Ireland, 2007. ACM.

[149] Lukas Naef. *ClickScript a visual programming language in the browser*. Master thesis, ETH Zurich, Switzerland, 2009.

[150] L. O'Gorman and T. Pavlidis. Auto ID technology: From barcodes to biometrics. *IEEE Robotics & Automation Magazine*, 6(1):4–6, 1999.

[151] Benedikt Ostermaier, Kay Römer, Friedemann Mattern, Michael Fahrmair, and Wolfgang Kellerer. A Real-Time Search Engine for the Web of Things. In *Proceedings of Internet of Things 2010 International Conference (IoT 2010)*, Tokyo, Japan, November 2010.

[152] Danny Parker, David Hoak, Jamie Cummings, and Florida Solar. Pilot Evaluation of Energy Savings and Persistence from Residential Energy Demand Feedback Devices in a Hot Climate. In *Proc of the ACEEE Summer Study on Energy Efficiency in Buildings (ACEEE 2010)*, pages 245–259, 2010.

[153] Cesare Pautasso and Erik Wilde. Why is the web loosely coupled?: a multi-faceted metric for service design. In *Proc. of the 18th international conference on World Wide Web (WWW '09)*, pages 911–920, Madrid, Spain, April 2009. ACM.

[154] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. big web services: making the right architectural decision. In *Proc. of the 17th*

international conference on World Wide Web (WWW '08), pages 805–814, New York, NY, USA, 2008. ACM.

[155] Thomas Pham. *Resource Oriented Architecture in Wireless Sensor Network*. Bachelor thesis, University of Applied Science of Western Switzerland, 2008.

[156] Antonio Pintus, Davide Carboni, and Andrea Piras. The Anatomy of a Large Scale Social Web for Internet Enabled Objects. In Dominique Guinard, Vlad Trifa, and Erik Wilde, editors, *Proc. of the 2nd International Workshop on the Web of Things (WoT 2011)*, San Fransisco, USA, 2011. ACM.

[157] N.B. Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao. Tiny web services: design and implementation of interoperable and evolvable sensor networks. In *Proc. of the 6th ACM conference on Embedded Network Sensor Systems (SenSys '08)*, pages 253–266, Raleigh, NC, USA, 2008. ACM.

[158] T Quack, H Bay, and L Van Gool. Object Recognition for the Internet of Things. In Christian Floerkemeier, Marc Langheinrich, Elgar Fleisch, Friedemann Mattern, and Sanjay E Sarma, editors, *First International Conference on the Internet of Things (IoT 2008)*, volume 4952 of *Lecture Notes in Computer Science*, pages 230–246. Springer-Verlag New York Inc, 2008.

[159] Jun Rekimoto, Brygg Ullmer, and Haruo Oba. DataTiles: a modular platform for mixed physical and graphical interactions. In *Proc. of the SIGCHI conference on Human factors in computing systems (CHI '01)*, CHI '01, pages 269–276, Seattle, Washington, United States, 2001. ACM.

[160] Leonard Richardson and Sam Ruby. *RESTful web services*. O'Reilly Media, May 2007.

[161] T. Riedel, N. Fantana, A. Genaid, D. Yordanov, H.R. Schmidtke, and M. Beigl. Using web service gateways and code generation for sustainable IoT system development. In *Internet of Things (IOT), 2010*, pages 1–8. IEEE, 2010.

[162] Joel J. P. C. Rodrigues and Paulo A. C. S. Neves. A survey on IP-based wireless sensor network solutions. *International Journal of Communication Systems*, 2010.

[163] Marc Roelands, Laurence Claeys, Marc Godon, Marjan Geerts, Mohamed Ali Feki, and Lieven Trappeniers. Enabling the Masses to Become Creative in Smart Spaces. In Dieter Uckelmann, Mark Harrison, and Florian Michahelles, editors, *Architecting the Internet of Things*, pages 37–64. Springer Berlin Heidelberg, 2011.

[164] Michael Rohs. *Linking Physical and Virtual Worlds with Visual Markers and Handheld Devices*. Phd thesis, ETH Zurich, Zurich, Switzerland, August 2005.

[165] Kay Römer, Thomas Schoch, Friedemann Mattern, and Thomas Dübendorfer. Smart Identification Frameworks for Ubiquitous Computing Applications. *Wireless Networks*, 10(6):689–700, December 2004.

[166] K. Romer, B. Ostermaier, F. Mattern, M. Fahrmair, and W. Kellerer. Real-Time Search for Real-World Entities: A Survey. *Proceedings of the IEEE*, 98(11):1887–1902, 2010.

[167] Enrico Rukzio, Gregor Broll, Karin Leichtenstern, and Albrecht Schmidt. *Mobile Interaction with the Real World: An Evaluation and Comparison of Physical Mobile Interaction Techniques*, pages 1–18. Springer Berlin / Heidelberg, 2007.

[168] Daniel Salber, A.K. Dey, and G.D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *Proc. of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, pages 434–441, Pittsburgh, Pennsylvania, United States, 1999. ACM.

[169] S. Sarma, D. Brock, and D. Engels. Radio frequency identification and the electronic product code. *IEEE Micro*, 21(6):50–54, 2001.

[170] Sanjay Sarma, David L Brock, and Kevin Ashton. The Networked Physical World. Technical report, MIT Auto-ID Labs, 2001.

[171] Albrecht Schmidt, Michael Beigl, and H.W. Gellersen. There is more to context than location. *Computers & Graphics*, 23(6):893–901, November 1999.

[172] Patrick Schmitt. *Adoption und Diffusion neuer Technologien am Beispiel der Radiofrequenz-Identifikation (RFID).* Phd thesis, ETH Zurich, 2008.

[173] Lars Schor, Philipp Sommer, and Roger Wattenhofer. Towards a zero-configuration wireless sensor network architecture for smart buildings. In *Proc. of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys '09)*, pages 31–36. ACM, November 2009.

[174] P. Schramm, E. Naroska, P. Resch, J. Platte, H. Linde, G. Stromberg, and T. Sturm. A service gateway for networked sensor systems. *IEEE Pervasive Computing*, 3(1):66– 74, March 2004.

[175] S. Senecal. The influence of online product recommendations on consumers' online choices. *Journal of Retailing*, 80(2):159, 2004.

[176] Z. Shelby. Embedded web services. *IEEE Wireless Communications*, 17(6):52–57, December 2010.

[177] A. P Sheth, K. Gomadam, and J. Lathem. SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups. *IEEE Internet Computing*, 11(6):91–94, December 2007.

[178] A.S. Shirazi, C. Winkler, and A. Schmidt. SENSE-SATION: An extensible platform for integration of phones into the Web. In *Proc. of the Internet of Things 2010 International Conference (IoT 2010)*, pages 1–8, 2010.

[179] Shwetak Patel, Thomas Robertson, Julie Kientz, Matthew Reynolds, and Gregory Abowd. At the Flick of a Switch: Detecting and Classifying Unique Electrical

Events on the Residential Power Line. In *Proc. of the International Conference on Ubiquitous Computing (UbiComp 2007)*, pages 271–288, 2007.

[180] Meredith M Skeels and Jonathan Grudin. When social networks cross boundaries: a case study of workplace use of facebook and linkedin. In *Proc. of the ACM International Conference on Supporting Group Work (Group '09)*, pages 95–104, Sanibel Island, Florida, USA, 2009. ACM.

[181] H. Song, Doreen Cheng, A. Messer, and S. Kalasapur. Web Service Discovery Using General-Purpose Search Engines. In *Proc. of the IEEE International Conference on Web Services (ICWS 2007)*, pages 265–271, 2007.

[182] Steve Souders. *High Performance Web Sites: Essential Knowledge for Front-End Engineers*. O'Reilly Media, Inc., September 2007.

[183] Patrik Spiess, Stamatis Karnouskos, Dominique Guinard, Domnic Savio, Oliver Baecker, Luciana Moreira Sa de Souza, and Vlad Trifa. SOA-based Integration of the Internet of Things in Enterprise Services. In *Proc of the IEEE International Conference on Web Services (ICWS 2009)*, Los Angeles, California, USA, July 2009.

[184] Katarina Stanoevska-Slabeva and Thomas Wozniak. Cloud Basics An Introduction to Cloud Computing. In Katarina Stanoevska-Slabeva, Thomas Wozniak, and Santi Ristol, editors, *Grid and Cloud Computing*, chapter 4, pages 47–61. Springer, Berlin, Heidelberg, 2010.

[185] V. Stirbu. Towards a RESTful Plug and Play Experience in the Web of Things. In *Proc. of the IEEE International Conference on Semantic Computing (ICSC '08)*, pages 512–517, 2008.

[186] J. Swartz. The growing magic of automatic identification. *IEEE Robotics & Automation Magazine*, 6(1):20–23, 56, 1999.

[187] C Tan, Bo Sheng, and Haodong Wang. Microsearch: When search engines meet small devices. In *Pervasive Computing*, Sydney, Australia, 2008. Springer-Verlag.

[188] Schmid Thomas, Young Cho, and Mani B. Srivastava. Exploiting Social Networks for Sensor Data Sharing with SenseShare. Technical report, CENS 5th Annual Research Review, 2007.

[189] Ken Traub, Felice Armenio, Henri Barthel, Paul Dietrich, John Duker, Christian Floerkemeier, John Garrett, Mark Harrison, Bernie Hogan, Jin Mitsugi, Josef Preishuber-Pfluegl, Oleg Ryaboy, Sanjay Sarma, KK Suen, and John Williams. The EPCglobal Architecture Framework. Technical report, EPCglobal, 2010.

[190] Vlad Trifa, Dominique Guinard, Philipp Bolliger, and Samuel Wieland. Design of a Web-based distributed location-aware infrastructure for mobile devices. In *Proc. of the First International Workshop on the Web of Things (WoT 2010)*, pages 714–719, Mannheim, Germany, March 2010. IEEE.

[191] Vlad Trifa, Dominique Guinard, Vlatko Davidovski, Andreas Kamilaris, and Ivan Delchev. Web-based Messaging Mechanisms for Open and Scalable Distributed Sensing Applications. In *Proceedings the International Conference on Web Engineering (ICWE 2010)*, Vienna, Austria, July 2010.

[192] Vlad Trifa, Dominique Guinard, and Simon Mayer. *Leveraging the Web to Build a Distributed Location-aware Infrastructure for the Real World*, chapter 17. Springer, August 2011.

[193] Vlad Trifa, Samuel Wieland, Dominique Guinard, and T.M. Bohnert. 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.

[194] Brygg Ullmer, Hiroshi Ishii, and Dylan Glas. mediaBlocks: physical containers, transports, and controls for online media. In *Proc. of the 25th annual conference on Computer graphics and interactive techniques (SIGGRAPH '98)*, pages 379–386, New York, NY, USA, 1998. ACM.

[195] M M Van Raaij Theo and W Fred. A behavioral model of residential energy use. *Journal of Economic Psychology*, 3(1):39–63, 1983.

[196] J.P. Vasseur and Adam Dunkels. *Interconnecting smart objects with IP: the next internet*. Morgan Kaufmann Publishers Inc., June 2010.

[197] Juan Vazquez and Diego Lopez-de Ipina. Social Devices: Autonomous Artifacts That Communicate on the Internet. In *Proc. of the 1st international conference on the Internet of Things (IoT 2008)*, pages 308–324, 2008.

[198] Jo Vermeulen, Kris Luyten, Karin Coninx, and Ruben Thys. Tangible Mashups: Exploiting Links between the Physical and Virtual World. In *Proc. of the 1st International Workshop on System Support for the Internet of Things (WoSSIoT'07)*, 2007.

[199] Jo Vermeulen, Ruben Thys, Kris Luyten, and Karin Coninx. Making Bits and Atoms Talk Today. In *Proc. of the 1st International Workshop on Design and Integration Principles for Smart Objects (DI PSO 2007)*, 2007.

[200] Felix von Reischach, Dominique Guinard, Florian Michahelles, and Elgar Fleisch. A mobile product recommendation system interacting with tagged products. In *Proc. of PerCom 2009 (IEEE International Conference on Pervasive Computing and Communications)*, Galveston, Texas, USA, March 2009.

[201] Felix Von Reischach, Florian Michahelles, Dominique Guinard, Robert Adelmann, Elgar Fleisch, and A. Schmidt. An evaluation of product identification techniques for mobile phones. In *Proc. of the Conference in Human-Computer Interaction (INTERACT 2009)*, pages 804–816, Uppsala, August 2009. Springer.

[202] Roy Want, Kenneth P Fishkin, Anuj Gujar, and Beverly L Harrison. Bridging physical and virtual worlds with electronic tags. In *Proc. of the SIGCHI conference on Human factors in computing systems (CHI '99)*, pages 370–377, Pittsburgh, Pennsylvania, United States, 1999. ACM.

[203] Mark Weiser. The computer for the 21st century. *Scientific American*, 3:94–104, July 1991.

[204] Markus Weiss and Dominique Guinard. Increasing energy awareness through web-enabled power outlets. In *Proceedings of the 9th International Conference on Mobile and Ubiquitous Multimedia*, page 20, Limassol, Cyprus, December 2010. ACM.

[205] Markus Weiss, Dominique Guinard, T. Staake, and Wolf Roediger. eMeter: An interactive energy monitor. In *Adjunct Proc. of the International Conference on Ubiquitous Computing (UbiComp 2009)*, Orlando, Florida, USA, September 2009.

[206] Markus Weiss, Friedemann Mattern, Tobias Graml, Thorsten Staake, and Elgar Fleisch. Handy feedback: connecting smart meters with mobile phones. In *Proc. of the 8th International Conference on Mobile and Ubiquitous Multimedia (MUM '09)*, MUM '09, pages 15:1–15:4, Cambridge, United Kingdom, 2009. ACM.

[207] E. Welbourne, L. Battle, G. Cole, K. Gould, K. Rector, S. Raymer, M. Balazinska, and G. Borriello. Building the Internet of Things Using RFID: The RFID Ecosystem Experience. *IEEE Internet Computing*, 13(3):48–55, June 2009.

[208] T. Wiechert, F. Thiesse, F. Michahelles, P. Schmitt, and E. Fleisch. Connecting mobile phones to the Internet of things: A discussion of compatibility issues between EPC and NFC. In *Americas Conference on Information Systems, AMCIS*, Keystone, Colorado, USA, 2007.

[209] Erik Wilde. Putting Things to REST. Technical report, School of Information, UC Berkeley, November 2007.

[210] Erik Wilde. Feeds as Query Result Serializations. Technical Report 2009-030, April 2009.

[211] Jakkhupan Worapot, Yuefeng Li, and Arch-Int Somjit. Design and implementation of the EPC Discovery Services with confidentiality for multiple data owners. In *Proc of the IEEE International Conference on RFID-Technology and Applications (RFID-TA 2010)*, pages 19–25. IEEE, June 2010.

[212] Ke Xu, Xiaoqi Zhang, Meina Song, and Junde Song. Mobile Mashup: Architecture, Challenges and Suggestions. In *Proc. of the International Conference on Management and Service Science (MASS '09)*, pages 1–4, Beijing, China, September 2009.

[213] Kok-Kiong Yap, Vikram Srinivasan, and Mehul Motani. MAX: human-centric search of the physical world. In *Proc. of the 3rd international conference on embedded networked sensor systems (SenSys '05)*, pages 166–179, San Diego, California, USA, 2005. ACM.

[214] Dogan Yazar and Adam Dunkels. Efficient application integration in IP-based sensor networks. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, page 4348, Berkeley, CA, USA, November 2009.

[215] Raymond Yee. *Pro Web 2.0 Mashups: Remixing Data and Web Services*. Apress, March 2008.

[216] Jin Yu, Boualem Benatallah, Fabio Casati, and Florian Daniel. Understanding Mashup Development. *IEEE Internet Computing*, 12(5):44–52, 2008.

[217] Shuai Zhang, Shufen Zhang, Xuebin Chen, and Xiuzhen Huo. Cloud Computing Research and Development Trend. In *Second International Conference on Future Networks (ICFN '10)*, pages 93–97. IEEE, January 2010.

[218] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems (TODS)*, 23:453–490, December 1998.

Referenced Web Ressources

[219] Amazon web services. http://aws.amazon.com/.

[220] American energy review (EIA) for 2010. http://www.eia.gov/totalenergy/data/annual/.

[221] Apache abdera atompub server. http://abdera.apache.org/.

[222] Apache tomcat. http://tomcat.apache.org/.

[223] Atmosphere: Web push abstraction framework. http://atmosphere.java.net/.

[224] autowot - a toolkit for the rapid integration of smart devices into the web of things - google project hosting. http://code.google.com/p/autowot/.

[225] The contiki operating system - instant contiki. http://www.sics.se/contiki/instant-contiki.html.

[226] Craigslist. http://www.craigslist.com.

[227] Dpws specification 1.1. http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.pdf.

[228] EnergieVisible: our consumption is visible! opensource software for sensing and visualizing electricity consumption. http://webofthings.com/energievisible/.

[229] EPCglobal architecture framework standards. http://www.gs1.org/gsmp/kc/epcglobal/architecture.

[230] EPCglobal discovery services standard (in development). http://www.gs1.org/gsmp/kc/epcglobal/discovery.

[231] European environment agency: Maps and graphs. http://www.eea.europa.eu/data-and-maps/figures#c15=all&c5=&c9=&c0=15&b_start=0.

[232] Extended environments markup language: EEML. http://www.eeml.org/.

[233] Feel, act, make sense - sen.se. http://open.sen.se/.

[234] Fosstrak - open source software for track and trace. http://www.fosstrak.org/.

[235] Fosstrak EPCIS - EPCIS webadapter home page. http://www.fosstrak.org/epcis/docs/webadapter-guide.html.

[236] GlassFish: open source application server. http://glassfish.java.net/.

[237] Google maps. http://maps.google.com.

[238] Google visualization API reference - google chart tools - google code. http://code.google.com/apis/chart/interactive/docs/reference.html.

[239] Google web toolkit. http://code.google.com/intl/en/webtoolkit/.

[240] Grizzly NIO web server. http://grizzly.java.net/.

[241] HousingMaps. http://www.housingmaps.com.

[242] IETF working group for constrained RESTful environments. http://tools.ietf.org/wg/core/.

[243] Internet of things - ThingSpeak. https://www.thingspeak.com/.

[244] IPSO alliance: Enabling the internet of things. http://ipso-alliance.org/.

[245] iui - web UI framework for mobile devices - iOS, android, palm, and others. http://code.google.com/p/iui/.

[246] The java community Process(SM) program - JSRs: java specification requests - JSR# 244 (jdo). http://www.jcp.org/en/jsr/detail?id=243.

[247] Java ME (mobile and embedded edition). http://www.oracle.com/technetwork/java/javame/index.html.

[248] JAX-RS standard for RESTful web services (JSR 311). http://jsr311.java.net/.

[249] Jersey JAX-RS reference implementation for building RESTful web service. http://jersey.java.net/.

[250] jQuery: the write less, do more, JavaScript library. http://jquery.com/.

[251] JSR-000220 enterprise JavaBeans 3.0 - final release. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html.

[252] kSOAP 2 WS-* for resource constrained devices. http://ksoap2.sourceforge.net/.

[253] Metering portal: Information for metering professionals. http://metering.com.

[254] Microformats. http://microformats.org/.

[255] Mobile JavaScript framework for developing HTML5 web applications. http://www.sencha.com/products/touch/.

[256] Mongoose - easy to use embedded web server. http://code.google.com/p/mongoose/.

[257] NorhTec - products. http://www.norhtec.com/products/mcsr/index.html.

[258] OpenPICUS is an open source embedded platform for smart sensors and the internet of things. http://www.openpicus.com/cms/.

[259] OpenWrt: linux distribution for embedded devices. https://openwrt.org/.

[260] OSGi alliance. http://www.osgi.org/Main/HomePage.

[261] Phidgets prototyping RFID reader. http://www.phidgets.com/products.php?product_id=1023.

[262] Plogg: wireless energy sensor networks. http://www.plogginternational.com/.

[263] ProgrammableWeb - mashups, APIs, and the web as platform. http://www.programmableweb.com.

[264] pubsubhubbub - a simple, open, web-hook-based pubsub protocol & open source reference implementation. http://code.google.com/p/pubsubhubbub/.

[265] RDFa in XHTML: syntax and processing. http://www.w3.org/TR/rdfa-syntax/.

[266] Real-Time open data web service for the internet of things - pachube. http://pachube.com/.

[267] REST binding project at AUTO-ID labs at MIT. http://autoidlabs.mit.edu/CS/content/OpenSource.aspx.

[268] Restlet - RESTful web framework for java. http://www.restlet.org/.

[269] Roving networks wifi modules. http://www.rovingnetworks.com/.

[270] ruote - open source ruby workflow engine. http://ruote.rubyforge.org/.

[271] Server-Sent events. http://dev.w3.org/html5/eventsource/.

[272] Size of wikipedia. http://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia.

[273] SunSPOT - embedded development platform. http://www.sunspotworld.com/.

[274] ThingWorx the 1st application platform for the connected world. http://www.thingworx.com/.

[275] Travel - guides - the new york times, supporting microformats. http://travel.nytimes.com/.

[276] Ubuntu cloud. http://www.ubuntu.com/business/cloud/overview.

[277] Unbeatable JavaScript tools - the dojo toolkit. http://dojotoolkit.org/.

[278] URIs, URLs, and URNs: clarifications and recommendations 1.0. http://www.w3.org/TR/uri-clarification/#contemporary.

[279] VirtualBox. http://www.virtualbox.org/.

[280] VMware virtualization software for desktops, servers & virtual machines for public and private cloud solutions. http://www.vmware.com/.

[281] Webnergy - a project to web-enable smart meters. http://code.google.com/p/webnergy/.

[282] The WebSocket API. http://dev.w3.org/html5/websockets/.

[283] Weebiz: Business community. https://weebiz.com/.

[284] Wikipedia – comet programming. http://en.wikipedia.org/wiki/Comet_(programming).

[285] Wikipedia – reflection in computer programming. http://en.wikipedia.org/wiki/Reflection_(computer_programming).

[286] WS4D web services for devices. http://www.ws4d.org/.

[287] Yahoo pipes mashup editor: Rewire the web. http://pipes.yahoo.com/pipes/.

[288] Yaler - access small devices from the web. http://yaler.org/.

[289] Yahoo search blog: We now support microformats. http://www.ysearchblog.com/2006/06/21/we-now-support-microformats/, June 2006.

[290] Marking up products for rich snippets - webmaster tools help. http://www.google.com/support/webmasters/bin/answer.py?answer=146750, March 2011.

[291] Tim Berners-Lee. Keynote at WWW 2009: Twenty years: Looking forward, looking back. http://www2009.eprints.org/212/1/index.html, April 2009.

[292] Tantek Celik. Geo microformats. http://microformats.org/wiki/geo, November 2009.

[293] Tantek Celik, Ali Diab, Ian McAllister, John Panzer, Adam Rifkin, and Michael Sippey. hReview 0.3 microformats. http://microformats.org/wiki/hReview, February 2006.

[294] Tantek Celik and Brian Suda. hCard 1.0 microformats wiki. http://microformats.org/wiki/hCard, May 2011.

[295] Rory Cellan-Jones. BBC - dot.life: Things that tweet. http://www.bbc.co.uk/blogs/technology/2009/06/things_that_tweet.html, June 2009.

[296] F. Dawson and T. Howes. vCard MIME directory profile, IETF memo. http://www.ietf.org/rfc/rfc2426.txt, September 1998.

[297] Paul Lee and Jay Myers. hProduct microformat draft specification. http://microformats.org/wiki/hproduct, June 2010.

[298] Lukas Naef. ClickScript: easy to use visual programming language. http://clickscript.ch/site/home.php.

[299] OpenSocial and Gadgets Specification Group. OpenSocial RESTful protocol specification v0.9. http://www.opensocial.org/Technical-Resources/opensocial-spec-v09/REST-API.html, April 2009.

[300] G. J. Rothfuss. Official google maps API blog: Microformats in google maps. http://googlemapsapi.blogspot.com/2007/06/microformats-in-google-maps.html, July 2007.

[301] Alex Williams. SOAP is not dead - it's undead, a zombie in the enterprise. http://www.readwriteweb.com/enterprise/2011/05/soap-is-not-dead---its-undead.php.

Curriculum Vitae: Dominique Guinard

Personal Data

Date of Birth	February 27, 1981 in Fribourg
Citizenship	Swiss

Education

2007–2011	*ETH Zurich*, Ph.D. Student in Pervasive Computing (Department of Computer Science), Zurich Switzerland
2010–2011	*MIT*, Visiting Researcher at the MIT Auto-ID Labs, Cambridge, USA
2006–2007	*University of Lancaster*, Visiting Graduate Student in Ubiquitous Computing (Master thesis), Lancaster, UK
2005–2007	*Universities of Fribourg and Bern*, M.Sc. in Computer Science, Fribourg & Bern, Switzerland
2002–2005	*University of Fribourg*, B.Sc. in Computer Science and Business Management, Fribourg, Switzerland
1998–2001	*Collège de Gambach*, Baccalauréat in Business Administration (High school), Fribourg, Switzerland

Employment

2007 – 2011	Research Associate, *SAP Research*, Zurich, Switzerland
2008 – 2011	Research Assistant, *Department of Computer Science, ETH Zurich*, Zurich, Switzerland
2007 – 2008	Research Associate, *ETH Auto-ID Labs*, Zurich, Switzerland
2005 – 2007	External Scientific Collaborator, *Software Engineering Group, University of Fribourg*, Fribourg, Switzerland
2005 – 2006	Teaching Assistant, *Pervasive and Artificial Intelligence Group, University of Fribourg*, Fribourg, Switzerland
2002 – 2006	Professional Trainer in Web Development, *Ecole-Club Migros*, Fribourg, Switzerland
1999 – 2006	Co-founder, Project Manager, *Spoker and GMIPSoft*, Fribourg, Switzerland
2005 – 2005	Intern at the Software Practice, *Sun Microsystems*, Gland, Switzerland
1998 – 1999	Web Developer, *Dartfish*, Fribourg, Switzerland