Diss. ETH No. 22787

The Web as an Interface to the Physical World

Real-time Search and Development Support

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

Andreas Benedikt Ostermaier
Diplom-Informatiker, Technische Universität München
born on March 9, 1979
citizen of Germany

accepted on the recommendation of

Prof. Dr. Friedemann Mattern, examiner
Prof. Dr.-Ing. Wolfgang Kellerer, co-examiner
Prof. Dr. Kay Römer, co-examiner

2015

Abstract

Connecting our physical world to the Internet is the ambitious vision of the *Internet of Things*. Embedding sensors and actuators into arbitrary "things" and connecting them to the Internet enables the location-independent monitoring, controlling, and programming of the physical world on an unprecedented scale, often in real time. Beyond connectivity, the ability of devices to interoperate with other devices and services will be an important requirement in a future Internet of Things – the sheer number and heterogeneity of Internet-connected things renders manual integration infeasible.

Based on the existing architecture of the World Wide Web, the so-called *Web of Things* addresses the challenge of interoperability for Internet-connected things. By leveraging existing Web standards and infrastructure, one can benefit from the many aspects that have already been covered in the Web architecture as well as from the plethora of existing Web services and Web-enabled applications. Exposing physical entities, in particular their sensors and actuators, on the Web in order to provide an interface for retrieving and changing the state of such entities and their environment is a well-known concept. With the increasing amount of Internet-connected devices and the advent of novel standards and services, this concept has recently gained momentum but has also shown the limitations of existing approaches.

This thesis addresses this development by providing the following contributions to a future Web of Things:

- The development and evaluation of a prototypical real-time search engine for the physical world that is based on the Web architecture. In contrast to traditional Web search engines, a search engine for the physical world has to support searching for structured and rapidly changing, distributed state in real time. The proposed solution is based on an open architecture and requires neither a global view of the world's state nor a limitation of the search space while still providing accurate results in real time.

- A prototypical framework for the Web of Things that simplifies the connection of sensors and actuators to the Web as well as their composition to novel services. It provides key primitives identified during the development of several experimental applications. The proposed solution does not advocate a central hub but strives to enhance today's inherently decentralized Web architecture.
- A concept of connecting everyday objects directly to the Web that leverages the ubiquity of Wi-Fi access points and the interoperability of the HTTP protocol, utilizing programmable low-power Wi-Fi modules. Using a loosely coupled approach, the seamless association of sensors, actuators, and everyday objects with each other and with the Web is enabled. Experimental results show that low-power Wi-Fi modules can achieve long battery lifetime despite using the resource-intensive protocols and data formats of the Web for communication.

In summary, this thesis demonstrates that leveraging the Web's architecture in order to address interoperability in a future Internet of Things is a promising concept by covering several aspects of this development.

Kurzfassung

Die ambitionierte Vision des *Internets der Dinge* ist die Verbindung der physischen mit der virtuellen Welt. Durch die Einbettung von Sensoren und Aktuatoren in beliebige „Dinge" und deren Verbindung mit dem Internet wird es möglich, die physische Welt ortsunabhängig, in einem beispiellosen Ausmass und oftmals in Echtzeit zu beobachten, zu steuern und zu programmieren. Neben der reinen Konnektivität wird die Interoperabilität von Geräten mit anderen Geräten und Diensten eine wichtige Anforderung in einem zukünftigen Internet der Dinge werden, da auf Grund der grossen Anzahl und Heterogenität solcher Geräte eine manuelle Integration nicht praktikabel ist.

Das sogenannte *Web der Dinge* nutzt die existierende Architektur des World Wide Webs, um Interoperabilität für Dinge im Internet zu ermöglichen. Durch die Verwendung der existierenden Standards und Infrastruktur des Webs kann sowohl von den zahlreichen der dort bereits gelösten Problemen profitiert werden, als auch die grosse Anzahl der dort vorhanden Dienste und Anwendungen genutzt werden. Die Repräsentation physischer Objekte im Web, inklusive deren Sensoren und Aktuatoren, wird seit Längerem praktiziert. Dies ermöglicht das Auslesen und Verändern der Zustände solcher Objekte und ihrer unmittelbaren Umgebungen über eine Web-Schnittstelle. Mit der steigenden Anzahl von mit dem Internet verbundenen Geräten und dem Aufkommen neuer Standards und Dienste hat dieses Konzept in letzter Zeit an Bedeutung gewonnen, aber auch die Grenzen existierender Ansätze gezeigt.

Diese Arbeit nimmt sich dieser Entwicklung an, indem sie die folgenden Beiträge zu einem zukünftigen Web der Dinge leistet:

- Die Entwicklung und Evaluation einer prototypischen Echtzeitsuchmaschine für die physische Welt, welche auf der Web-Architektur beruht. Anders als traditionelle Web-Suchmaschinen muss eine Suchmaschine für die physische Welt die Suche nach strukturierten und sich schnell ändernden, verteilten Zuständen in Echtzeit unterstützen. Die vorgeschlagene Lösung basiert auf einer of-

fenen Architektur und benötigt weder eine globale Sicht auf den Zustand der Welt noch eine Begrenzung des Suchraums, um akkurate Ergebnisse in Echtzeit zu liefern.

- Ein prototypisches Framework für das Web der Dinge, welches die Bereitstellung von Sensoren und Aktuatoren im Web und deren Zusammensetzung zu neuen Diensten vereinfacht. Es stellt dazu Grundbausteine zur Verfügung, welche bei der Entwicklung mehrerer experimenteller Anwendungen identifiziert wurden. Dieser Ansatz erfordert keinen zentralen Knotenpunkt, sondern ermöglicht stattdessen die Erweiterung der heutigen inhärent dezentralisierten Web-Architektur.

- Ein Konzept, um Alltagsgegenstände direkt mit dem Web zu verbinden. Dieses basiert auf programmierbaren und energiesparenden Wi-Fi-Modulen und macht sich die Allgegenwart von Wi-Fi-Zugangspunkten und die Interoperabilität des HTTP-Protokolls zunutze. Durch einen lose gekoppelten Ansatz wird die einfache Verknüpfung von Sensoren, Aktuatoren und Alltagsgegenständen untereinander und mit dem Web ermöglicht. Experimentelle Ergebnisse zeigen, dass solche batteriebetriebenen Wi-Fi-Module auch bei der Verwendung der ressourcenintensiven Protokolle und Datenformate des Webs für die Kommunikation lange Laufzeiten erreichen können.

Zusammengefasst zeigt diese Arbeit, dass die Verwendung der Web-Architektur ein vielversprechendes Konzept ist, um Interoperabilität in einem zukünftigen Internet der Dinge zu erreichen, indem sie wesentliche Aspekte dieses Ansatzes aufgreift und weiterentwickelt.

Acknowledgements

First of all, I would like to express my deep gratitude to my doctoral adviser Friedemann Mattern for the opportunity to pursue my PhD thesis at his group, for the excellent working conditions, for his longstanding and strong support, and for his almost unlimited patience. In the same way, I want to sincerely thank my co-examiner Kay Römer, who provided invaluable feedback on so many parts of my work, and always found time for discussions, despite his many obligations. I would also like to thank my co-examiner Wolfang Kellerer for his longstanding support.

During my time at the group, I had the pleasure to work with great and bright colleagues. Many of them also supported this work, by contributing or by providing support. Of all, I owe Christian Flörkemeier a debt of gratitude. Without his coaching, this thesis would probably not exist. I would especially like to thank Christof Roduner, Matthias Kovatsch, Wilhelm Kleiminger, Silvia Santini, Philipp Bolliger, Alexander Bernauer, and Hossein Shafagh for providing support at important times of this work, and my longstanding roommate Robert Adelmann for providing such a pleasant working atmosphere. Furthermore, I want to thank Marc Langheinrich, Matthias Ringwald, Iulia Ion, Gábor Sörös, Christian Beckel, Markus Weiss, Ruedi Arnold, Jonas Wolf, Dominique Guinard, Vlad Trifa, Simon Mayer and Elke Schaper for our intense and thoughtful discussions.

I was also fortunate enough to work with many talented students, who directly or indirectly supported the development of this thesis. I would like to take the opportunity to thank all of them for their efforts and for the insightful discussions. Maryam Elahi and Ronny Meier provided important groundwork for the real-time search engine. Fabian Schlup provided the initial implementation of the framework for the Web of Things. Andreas Börnert, Adrian Helfenstein, Moritz Hartmeier, Robert Weiser, David Degen, Beatrice Meier, Danilo Buloncelli, Christoph Bäni, Vlatko Davidovski, Yuna Roh and Andreas Keller all conducted their work in the broader context of this thesis.

I would also like to thank Denise Spicher, who was very supportive in the final stages of this thesis.

Last but not least, I would like to express my deepest gratitude to my family, who always supported me during this long-lasting effort. To Melanie, who went out of her way to support my work, and to my parents, who always provided support when needed.

Preface

During the work on this thesis, the concept of an "Internet of Things" gained significant popularity, driven by the noticeable advancements in technology.

Cars were brought to market that are wirelessly upgraded by software to acquire new driving features, smartphones became the dominant mobile computing platform and replaced an impressive set of single-purpose devices, and everyday objects such as light bulbs and body scales were introduced that can easily integrate with remote services to provide added value. All of these "things" were designed to be continuously connected to the Internet, which in turn continues to grow on its outskirts.

Internet access at my home is now faster than what my local network infrastructure can handle. Cell phone networks offer mobile Internet at high bandwidth, which can be used for outdoor video calls using ordinary smartphones – a couple of years ago, this was only possible through the use of an extensive infrastructure. Wi-Fi access points are now available at almost any building and are not only used to provide Internet access but also to assist in the localization of mobile devices.

Looking at the big picture, this is a truly impressive development that continues at significant pace. It seems that the time has come for a large-scale adoption of the Internet of Things. Significant value can be added when connecting formerly isolated systems and composing novel services. However, it is much to be hoped that, analog to the World Wide Web, a standardized, open, and decentralized Web of Things emerges that enables interoperability for devices and full participation for users.

Contents

Acronyms

6LoWPAN IPv6 over Low power Wireless Personal Area Networks

AES Advanced Encryption Standard

API application programming interface

APM aggregated prediction model

BPWD battery-powered wireless device

BSSID basic service set identification

CoAP Constrained Application Protocol

CPS cyber-physical system

CPU central processing unit

CSS Cascading Style Sheets

DHCP Dynamic Host Configuration Protocol

DNS Domain Name System

DSL digital subscriber line

DTLS Datagram Transport Layer Security

EPC Electronic Product Code

EXI Efficient XML Interchange

GATT Generic Attribute Profile

GENA General Event Notification Architecture

GPS Global Positioning System

GUI graphical user interface

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPU HTTP over UDP

IC integrated circuit

IEEE Institute of Electrical and Electronics Engineers

IoT Internet of Things

IP Internet Protocol

ISM industrial, scientific and medical

JSON JavaScript Object Notation

LAN local area network

LED light-emitting diode

LLRP Low Level Reader Protocol

M2M Machine to Machine

MAC media access control

MEMS microelectromechanical systems

MPPM multi-period prediction model

MTU maximum transmission unit

NAT network address translation

NFC near field communication

NSF National Science Foundation

PIR passive infrared

PSK pre-shared key

RAM random-access memory

RDF Resource Description Framework

REST Representational State Transfer

RFID radio-frequency identification

ROM read-only memory

RPC remote procedure call

RSS Rich Site Summary

RSSI received signal strength indication

SIM subscriber identity module

SMS Short Message Service

SOAP Simple Object Access Protocol

SPARQL SPARQL Protocol and RDF Query Language

SPPM single-period prediction model

SQL Structured Query Language

SSID service set identifier

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

UHF ultra high frequency

ULP ultra-low power

UPnP Universal Plug and Play

URI Uniform Resource Identifier

URL Uniform Resource Locator

URN Uniform Resource Name

WEP Wired Equivalent Privacy

WLAN wireless local area network

WoT Web of Things

WPA Wi-Fi Protected Access

WPAN wireless personal area network

WPS Wi-Fi Protected Setup

WSN wireless sensor network

WWW World Wide Web

XML Extensible Markup Language

XMPP Extensible Messaging and Presence Protocol

1 Introduction

Since their appearance in the middle of the last century, digital computers have gained an impressive distribution, leading to the so-called "digital revolution". In their early days, computers were big and expensive machines that could only be afforded by large organizations. Their operation required several specialists, and their capabilities were shared within an organization. At that time, computers were called "mainframes" and could easily fill a room. With the ability to realize complex integrated circuits in silicon, microprocessors that integrated all the functions of a processor on a single chip became available at the beginning of the 1970s. This resulted in a significant reduction of the size of a computer, along with its acquisition and operating costs, and therefore paved the way for computers to move to desktops. "Personal computers" (PCs) no longer had to be shared with other users and became affordable even for hobbyists. The dissemination of computers also created the desire to connect them, share resources, and exchange data. This eventually resulted in the creation of the Internet, which enabled communication between computers on a global scale. Progress continued at an astonishing pace, enabling increasingly powerful and ever-smaller computers. Wireless communication technologies emerged that allowed computers to become truly mobile, contributing to the immense growth of the Internet. Portable computers became feasible, and novel device classes, such as smartphones, appeared. Today, our life is closely intertwined with tiny embedded computers, such as in credit cards, household appliances, medical devices, vehicles, and industrial machines. In contrast to the servers deployed in data centers, these computers operate "close" to the physical world and can automatically interact with their immediate physical surroundings through sensors and actuators.

While such tiny computers currently often operate in isolation, there is an increasing trend to connect even the smallest "things" to the Internet, thus creating an Internet of Things. Given that most processors sold today are not used for general purpose computing, but are embed-

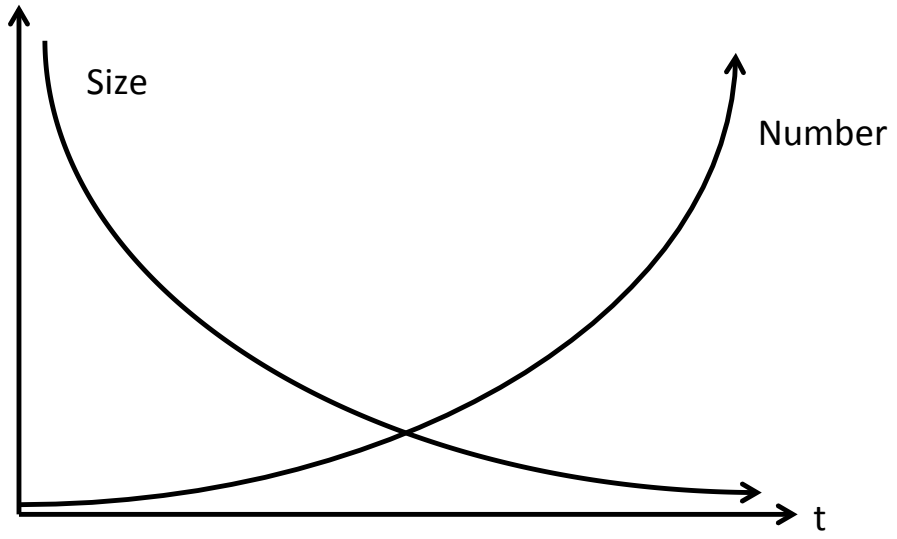


Figure 1.1: Miniaturization and proliferation of computers (based on [1])

ded in dedicated applications, this is expected to trigger yet another fundamental paradigm shift in computing. In 1999, Gershenfeld commented on this then-anticipated development: "in retrospect it looks like the rapid growth of the World Wide Web may have been just the trigger charge that is now setting off the real explosion, as things start to use the Net." [2]. Technologically, this development is driven by advancements in IT hardware as well as the proliferation of Internet access. Since the start of this century, significant progress has been made in both areas, which paved the way for a large-scale adoption of an Internet of Things.

Advancements in IT Hardware Arguably the most contributing factor to this development is Moore's Law, which suggests that the number of transistors in integrated circuits (ICs) double every 18 months [3]. This explains the increasing storage capabilities of IC-based memory, such as RAM and flash memory, and contributes to the significant performance gains achieved by microprocessors. Increasing the number of transistors on ICs eventually leads to a reduction in their structure size. This in turn improves energy efficiency, reduces production costs, and may help to reduce the physical footprint. For sensors, microelectromechanical systems (MEMS) enable a dramatic improvement over conventional sensors with respect to size, robustness, energy efficiency, and costs. This enables novel use cases, such as the inclusion of MEMS-based gyroscope sensors in modern smartphones. Some examples of miniaturized sensors are depicted in Fig. 1.2. Improved

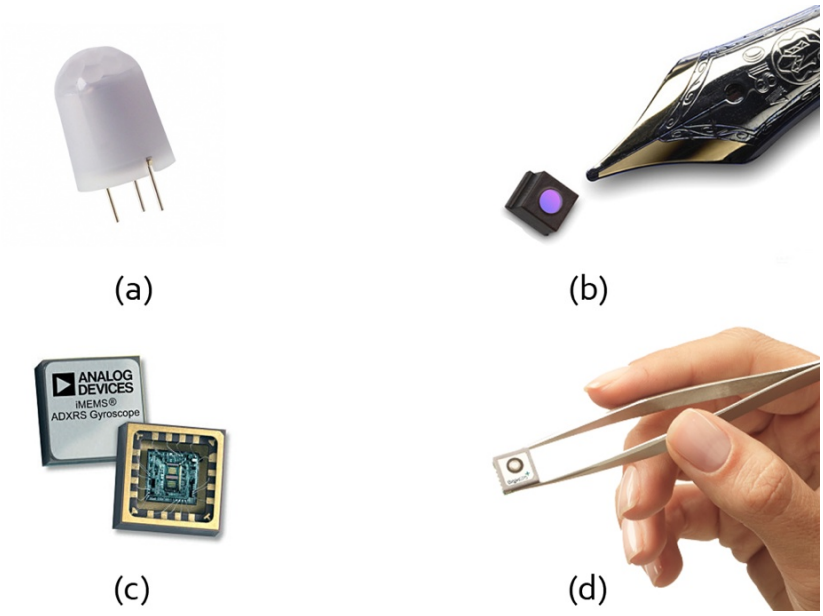


Figure 1.2: Some examples of miniaturized sensors: (a) a low-power passive infrared (PIR) sensor that can be used to detect the presence of people, (b) a camera sensor, (c) a gyroscope sensor based on a microelectromechanical system (MEMS), and (d) a GPS receiver module with integrated antenna[1]

modulation techniques and increased receiver sensitivity significantly improved wireless communication with respect to data transfer rates and energy efficiency.

Proliferation of Internet Access A crucial driving force is today's almost ubiquitous availability of broadband Internet access in industrial nations. While high-bandwidth Internet access in private homes originally required a coax cable or glass fiber, xDSL enabled broadband Internet for end users over unshielded copper cables, using the existing telephony wiring. However, glass fibers are currently being deployed to buildings that enable even higher transfer rates. Wireless Internet access over Wi-Fi has become the standard for mobile devices, and Wi-Fi Internet access points can now be found in almost any building. The underlying standards of IEEE 802.11 are continuously improved to enable even higher data transfer rates. Mobile Internet access based on cell phone networks also found widespread adoption in the recent decade. Improvements in modulation now allow for theoretical data

[1]Image sources: (a) Digi-Key (http://media.digikey.com/photos/PanasonicElectWorksPhotos/AMN41122,AMN11112,AMN21112.jpg), (b) based on image by LetsGoDigital (http://www.letsgodigital.org/images/artikelen/233/cameracube.jpg), (c) Sensors Online (http://www.sensorsmag.com/files/sensor/nodes/2010/6533/Figure2.jpg), (d) OriginGPS (http://www.origingps.com/wp-content/uploads/2014/01/hand.png)

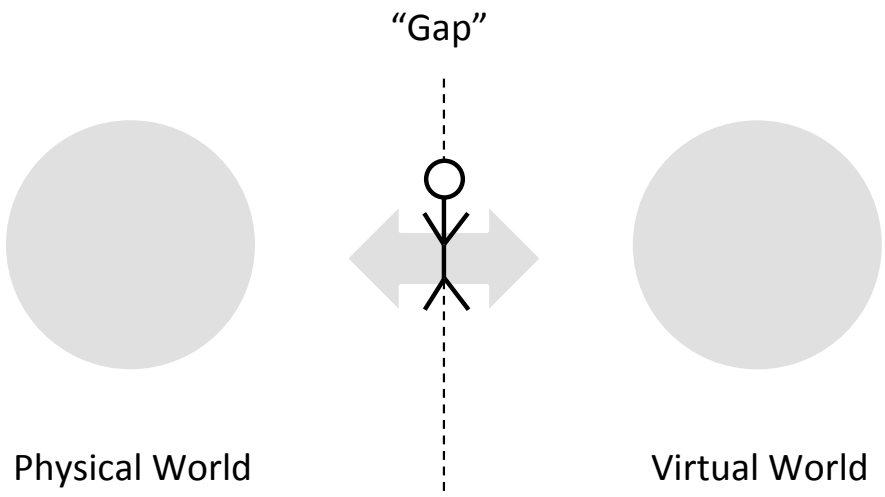


Figure 1.3: Today, the “gap” between the physical and the virtual world is often bridged by manual intervention.

transfer rates, in 4G mobile networks, that are comparable to xDSL-based landlines. The increasing availability and bandwidth of Internet access was followed by a significant reduction in its “costs per bit”, culminating in free Wi-Fi-based Internet access at public places.

1.1 The Convergence of the Physical and the Virtual World

Driven by the recent developments in IT hardware and the proliferation of Internet access, the long-running trend of “connecting” our physical world to the virtual world has recently gained momentum. On the one side, there is an increasing number of computers already embedded into our everyday world, which however currently often operate in isolation. On the other side, there is the virtual world, formed by an ever increasing number of networked computers in large data centers. Nevertheless, humans are often required to bridge the “gap” between these two worlds, such as by entering data regarding some state of the physical world (Fig. 1.3).

For example, standard electric meters require a manual readout on-site, often by an appointed person, which is usually performed at intervals of several months or even yearly. This data is currently used only for billing purposes and provides little feedback to the customer. Due to delayed and aggregated invoicing, the cause of higher or lower power consumption may be hard to determine when the bill finally arrives. Automating this process using so-called “smart meters” that can be read

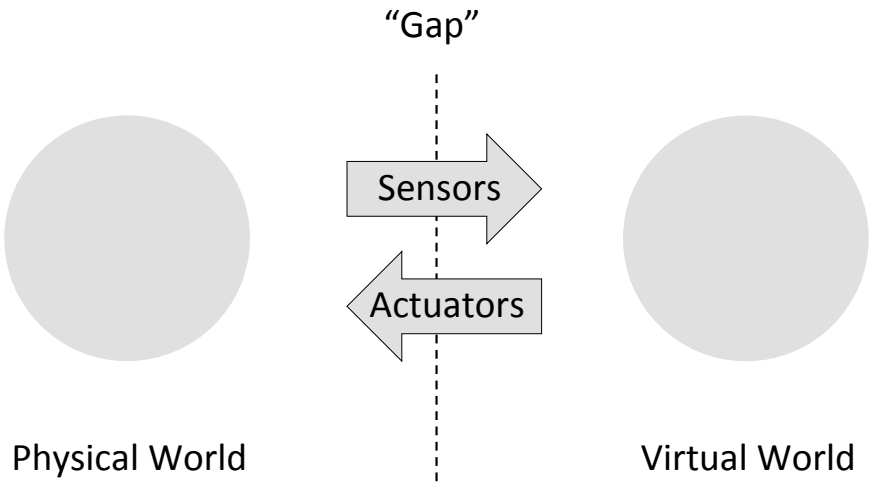


Figure 1.4: Bridging the “gap” between the physical and the virtual world using sensors and actuators.

out remotely and automatically at high temporal resolution provides transparency for the customer, reduces costs for the energy provider, and also enables novel use cases, such as appliance classification [4].

It is apparent that, in such scenarios, humans are the limiting factor. They often cannot compete with the speed, costs, and accuracy that can be provided by an automatic mediation performed by sensors and actuators (Fig. 1.4). Sensors measure certain parameters of their environment and provide readings that can be automatically interpreted by a computer. Actuators are used to effect a change in the physical world and can also be controlled by a computer. The global networking of sensors, actuators and computing resources enables location-independent monitoring, controlling, and programming of the physical world at an unprecedented scale, often in real time.

1.2 Related Fields

Several research fields and initiatives address the possibilities provided by low-cost, tiny and networked computers that are integrated into our everyday world in order to bridge the gap between the physical and the virtual world. In the following sections, we provide a summary of such fields and give examples.

1.2.1 Ubiquitous Computing

The field of Ubiquitous Computing (UbiComp) is the first and probably most well-known area of research that addressed the convergence of the physical and the virtual world. It was established by Marc Weiser at Xerox Parc at the end of the 1980s. Ubiquitous Computing envisions the seamless integration of tiny networked computers into our everyday world, to the point where they become invisible [5]. The focus is on assisting people with their daily activities by providing smart environments that are aware of the happenings within. In such scenarios, the use of computers is often implicit, as they are embedded in everyday objects. The term "Pervasive Computing" originated a few years later from an industrial context and denotes the same approach, albeit more focused on technology.

Early work in the field of ubiquitous and pervasive computing used infrared communication for connecting devices, because it is a cheap and low-power technology [6, 5, 7, 8]. The connection to the Web was often an afterthought and used to generate a dynamic Web page that displayed the state sensed by the devices. An early example is the Active Badge system, which consisted of small wearable computers that could communicate with a dedicated infrastructure using infrared links. In a paper from 1994, the authors mention that the system was used to display a person's current location, which was gathered using its Active Badge, on its personal communication profile on the Web [7]. In [9], the authors mention the use of a Web interface to provide an overview of the state of all of their augmented coffee cups, called MediaCups. In the same paper, the authors outline how small computing nodes that feature an RF interface can trigger the execution of an existing Web service: These devices can be utilized to detect the occupancy of a meeting room and then used to automatically set the occupancy state of the room in the booking system, via a Web-based meeting-room scheduling service.

UbiComp is a diverse field that its related to several research disciplines, such as human computer interaction, sensor networks, systems, and context detection and classification (e.g., places, activities, emotions). Examples include infrastructure-mediated sensing to detect human activity [10, 11] and the sensing of human emotions [12, 13].

1.2.2 Wireless Sensor Networks

Wireless sensor networks (WSNs) monitor certain phenomena of the physical world in a geospatial area of interest. Applications include the monitoring of wildlife, building structures, and climate change. Each sensor node is equipped with sensing, computation, and communication capabilities. WSNs may be located far away from researchers in an area that is difficult to access, such as a glacier. Since the existence of local infrastructure cannot be assumed in these regions, the sensor nodes communicate over radio links and create a local ad-hoc network, a wireless sensor network. In order to be able to cover a geospatial area that extends beyond the transmission range of a node, sensor readings are forwarded on behalf of other nodes (multi-hop routing and forwarding). WSNs usually operate off the grid and use dedicated low-power radio interfaces and application-specific protocols for communication. Data is usually routed to a dedicated sink node, using communication protocols such as the Collection Tree Protocol (CTP) [14]. The sink node features additional resources and connects the WSN to the Internet.

An early example for the application of WSNs is the monitoring of the Leach's storm petrel, a seabird, on a small island in the Gulf of Maine in 2002 [15]. For this, small sensor nodes that included seven different sensors were placed next to the birds' nesting environments. The network was connected through a base station, via satellite, to the Internet. The project also provided a Web page, which displayed current sensor readings from selected sensors. The sensor node used in this project was the Mica mote [16]; other well-known platforms include the TelosB/Tmote Sky [17, 18] and the BTnode [19]. Since then, sensor networks have been deployed for various applications, such as volcano monitoring [20], structural health monitoring [21], or sniper detection [22].

The proliferation of smartphones in recent years also spawned interest from the WSN research community to use them as a sensing platform. Several projects leveraged the built-in sensors of smartphones for sensing, often in an opportunistic and urban context [23, 24, 25, 26].

1.2.3 Embedded Systems

Embedded systems denote computing applications that serve a special purpose, usually in the control or monitoring domain of the physical

world. There, the system is embedded in a domain-specific application and can access both sensors and actuators in order to perform its task. Such applications often require real-time capabilities; i.e., the system needs to guarantee that it can perform a given task within a given timeframe. The usage of embedded systems reaches back to the 1960s, where on-board computers were used for guidance and navigation, first in the Minuteman intercontinental ballistic missiles [27] and subsequently in NASA's Gemini and Apollo space programs [28]. Since then, embedded systems proliferated and became truly pervasive. Examples of their usage include household appliances, building automation, traffic lights, cars and airplanes, medical devices, and machines for industrial production.

Many embedded systems are based on old but well-known processor architectures, which are sufficient for the given task [29]. Embedded systems typically communicate mostly locally, if at all. However, the networking trend has also affected embedded systems in recent years.

1.2.4 Cyber-physical Systems

Cyber-physical systems (CPS) address the tight integration of physical and computational resources to enable complex monitoring and control applications for the physical world. For this, distributed and networked embedded systems that provide both computation capabilities and interfaces to the physical world are utilized, possibly on a large scale [30, 31]. CPS focus on the engineering aspects of such large distributed systems, such as real-time constraints, closed-loop control, dependability, and correctness. The term "cyber-physical system" is attributed to Helen Gill, who coined it in 2006 at the National Science Foundation in the context of a research initiative [32]. However, much of the foundations of CPS were laid down by Stankovic et al. in 2005, despite lacking the exact term[2] [33]. Examples of CPS include the monitoring and control of a nation-wide power grid, the operation of a self-driving car, and the optimization of production processes of future factories. CPS are already deployed, for example, the handling of modern cars is defined in software to a significant extent. Research topics in the context of CPS include real-time constraints [34], robustness [35, 36], modeling aspects [37], and security [38].

[2] The authors use the term "physical computing systems".

Figure 1.5: Examples of M2M-related hardware: (a) standard SIM card and miniaturized M2M SIM card (non-removable but can be remotely provisioned), (b) a selection of M2M modules that simplifies the connection of machines to a cellular network[3].

1.2.5 Machine to Machine (M2M) Communication

The term "M2M" originated from mobile phone operators and stands for "machine to machine" communication. In contrast to the providers' original business model, M2M denotes applications in which humans are out of the loop. A typical application scenario consists of large numbers of machines in the field that feature a cell phone interface (such as GSM, GPRS, UMTS, LTE) through which the machines can communicate with a remote service. This can be used for remote monitoring, mobile payment, and also automation scenarios, for example. Initially based on SMS or dial-up connections [39], this cellular network-based communication is now shifting to IP. In that way, there is also a convergence of the terms M2M and Internet of Things [40, 41]. Just like with mobile phones, a SIM card is required to authenticate the machine to the operator's network. Today, the network coverage of mobile operators typically remains unrivaled in the field.

An upcoming and potentially large use case for M2M is the European eCall initiative, which makes an automatic emergency call system mandatory for all cars sold in the European Union as of April 2018 [42]. In case of an accident, the system will automatically transmit crash-related data to the closest emergency center, thus requiring a cell phone interface in every eCall-enabled car.

[3]Image sources: Gemalto (http://m2m.gemalto.com/tl_files/m2m_exp/content/header_pic/cinterion_m2m_products_and_sevices_round.jpg)

1.2.6 Industry 4.0

The term "Industry 4.0" refers to advanced manufacturing processes realized by a smart factory: production machines, workpieces, factory workers, and cloud services work hand in hand to improve the production of goods [43]. The notion refers to the upcoming fourth industrial revolution (with previous revolutions being driven by mechanics, electricity, and embedded computers) that is expected to be triggered by the application of cyber-physical systems/the Internet of Things (see next section) to manufacturing processes. A crucial aspect is the automatic identification of workpieces, which enables adaptive production lines. The notion is of German origin ("Industrie 4.0") and was coined around 2011. It currently has mostly local prominence, while the underlying concepts are of global interest [44].

Industry 4.0 strives to provide a competitive advantage over existing manufacturing processes through increased transparency, higher flexibility, and improved efficiency and manufacturing quality. This enables production methods such as mass customization or the cost-effective production of small batch sizes down to one-off items [43]. Industry 4.0 is part of a strategic initiative of the German government [45].

1.2.7 Internet of Things

Extending the Internet to the physical world, thereby pushing its borders up to the smallest everyday objects, is the ambitious vision of the Internet of Things (IoT) [46].

One of the first notions of the term "Internet of Things" stems from the context of augmenting consumer goods with RFID labels [47]. In this approach, physical entities are assigned an identifier that can be automatically read at well-known locations by a dedicated infrastructure that is connected to the Internet. In this scenario, all of the added value is provided by the infrastructure – the thing itself only needs to be identified, which can by achieved by attaching an RFID tag. A popular use case for this approach is "track and trace", where the whereabouts of physical entities can be determined automatically. Visual codes, such as barcodes and 2D codes, can also be used to tag and identify objects. Research includes middleware [48, 49] and the use of mobile phones as mediators between things and services [50, 51]. The approach of tagging physical objects to form an Internet of Things was pushed by the Auto-ID Center and is now actively pursued by its

successor organization, the Auto-ID Labs[4].

Today, the term "Internet of Things" is used in a broader context. It also denotes physical entities that are augmented with sensors and actuators, as well as computation and communication capabilities and that are connected to the Internet. This approach enables "smart" things, because they can be aware of their environment and also alter it to a certain degree. The connection to the Internet can be either direct or through some gateway, enabling the usage of the devices' capabilities without requiring physical proximity. Furthermore, the thing itself may leverage data and services available on the Internet in order to provide a certain functionality. This is a different approach in which much of the added value can be provided by the thing itself. An early and popular example is the vending machine that served cold beverages at the department of computer science at Carnegie Mellon University, which was first connected to the Internet in the 1970s [52]. Using a simple text-based interface, users could not only check whether beverages were available, but also if they were cold. The motivation for this interface originated from the desire to avoid unnecessary trips to the vending machine, which was located far from the computer scientists' offices. However, because the approach was based on a standard Internet service[5], the machine's status was easily accessible from any Internet-connected computer, which therefore gained some popularity.

Significant work to support the IP protocol stack on resource-constrained systems was performed by Dunkels [54, 55, 56]. This enables direct IP connectivity without the need for a gateway, making even tiny computers equal participants of the Internet. The address space of IPv4 [57], the current version of IP, features addresses with a length of 32 bits, and their availability is currently running out [58]. IPv6 [59], the next version of the Internet protocol, is in the process of wide-scale adoption. The most important improvement is the large address space of IPv6 addresses, which consist of 128 bits, making it possible to assign an IP address to every "thing" on the planet while still having significant reserve[6]. Using IPv6 over low-power wireless links, which are often slow and lossy, is addressed by 6LoWPAN [60, 61]. Today, the IPSO Alliance[7] is actively promoting the use of IP for connecting smart objects [62].

[4] http://www.autoidlabs.org

[5] It used the then-popular Finger service [53].

[6] There are 340,282,366,920,938,463,463,374,607,431,768,211,456 unique IPv6 addresses.

[7] http://www.ipso-alliance.org/

Several application protocols currently compete for adoption in the Internet of Things. Web services based on SOAP [63] or on the "pure" REST paradigm of HTTP [64, 65] have a head start, since they may leverage the existing ecosystem of the Web [66, 67, 68]. CoAP [69, 70, 71] is a resource-efficient implementation of the REST paradigm, providing additional features such as group communication and publish/-subscribe. It is feasible for resource-constraint networks and devices [72, 73, 74]. Messaging protocols such as MQTT [75], AQMD [76] and XMPP [77] have also been considered for use in the Internet of Things [67, 78, 79].

1.2.8 Web of Things

The Web of Things (WoT) uses the existing concepts, standards, and infrastructure of the World Wide Web in order to provide an interface between the physical and the virtual world. This approach can be used in both directions: On the one hand, things can act as Web clients to leverage the vast amounts of data, services, and applications available on the Web in order to provide enhanced functionality. On the other hand, the Web, with its large user base, can be extended to the physical world by utilizing existing Web-enabled applications and services to interact with Web servers that provide access to physical entities. Such Web servers can either run standalone and act as a gateway or be embedded in the actual physical objects. Access to such objects is not limited to Web pages through the use of a Web browser, but also includes programmatic interaction between things and applications based on Web services. At its core, the Web of Things addresses interoperability, which can be defined as the *"Ability of a system or a product to work with other systems or products without special effort on the part of the customer."* according to the IEEE [80].

This approach reaches back to the early days of the Web, where the state of the physical world was presented on Web pages of a Web server that acted as a gateway [7]. Running Web servers on embedded and resource-constrained devices was pursued a few years later [54, 81]. The idea of leveraging the Web architecture for the interaction with the physical world was pursued in depth by the pioneering Cooltown project at HP Labs [82, 83, 84] at the beginning of this century. The usage of the term "Web of Things" emerged later, around 2007 [85]. In recent years, the concept of a Web of Things gained significant mo-

mentum, partly fueled by the increasing number of Web services made possible by the advent of cloud computing. Such services can be used in concert with physical entities in order to create novel application scenarios, so-called physical mash-ups [46, 85, 86]. Web front-ends can be used to automatically realize dynamic user interfaces that adapt to their physical surroundings and provide provide control and monitoring functionality [67, 87]. Adding concepts from the Semantic Web [88] to the Web of Things in order to support semantic descriptions and reasoning is another promising approach that leverages parts of the Web's ecosystem [89, 90].

1.3 Motivation and Approach

Since Tim Berners-Lee laid the foundations of the World Wide Web in 1990 [91], it has become a remarkable success. While originally a document-centric hypermedia information sharing system, its protocol HTTP was quickly used for remote communication between machines as soon as the Web became more widespread [82, 92]. Today, the Web is the prevailing platform on the Internet to access and distribute information, as well as to share resources. It has proven to scale to large numbers of users and, at the same time, support novel application scenarios.

One reason for the success of the Web can be attributed to its open, decentralized architecture. Compared to earlier approaches, which were under the control of a single authoritative domain and utilized proprietary protocols and software[8], the Web is inherently distributed, both technically and with respect to authority. This makes participation simple and also fosters novel use cases.

Another reason for the Web's success lays in its technical foundations – at its core, it addresses some of the key problems that developers of distributed applications face:

- *Identification of resources:* In order to access remote resources such as files, the resources need to be identified. The concept of uniform resource identifiers (URIs) provides a global address space for identifiers of arbitrary types [93]. Since URIs are human-readable, they can easily be shared through informal methods[9].

[8] For example, CompuServe.
[9] For example, written down on a piece of paper.

- *Access to resources:* The HyperText Transfer Protocol (HTTP) provides a unified mechanism to access remote resources. It offers several features that ease the interaction with resources, such as pre-defined methods for creating, updating, reading and deleting resources, support for caching, access control, content negotiation, etc. Despite featuring "HyperText" in its name, HTTP is actually agnostic to the data that it transports.

- *Representation and linkage of resources:* The Hypertext Markup Language (HTML) [94, 95] is the foundation for formatting data that is presented to humans. *Hyperlinks* are HTTP URIs embedded in HTML that enable users to access linked information by following them using a Web browser. There are also standardized data description formats for the exchange of data between applications, such as XML [96] and JSON [97].

The Web was also used early in practice in order to provide access to current readings of remote sensors, such as webcams and weather stations. With the increasing amount of Internet-connected devices and the advent of novel standards and services, this approach has recently gained momentum. Today, a growing number of physical entities are accessible on the Web, usually providing an HTML-based interface for humans as well as an application programming interface (API) to be used by applications. Examples include body scales, light bulbs, plant sensors, cars, naval buoys, and building automation systems. In retrospect, one of the researchers of Cooltown motivates the approach as follows [98]:

> "First, since the Web provides a rich and extensible set of resources in the virtual world, much can potentially be gained by extending the Web's architecture and the Web's existing resources to the physical world. [...] The second objective was to achieve the Web's high degree of interoperability for interactions with devices. [...]"

However, there is a current trend wherein devices communicate with central hubs, which act as a gateway to the Web. While the advantage of using such hubs is that all connected devices benefit from the features they offer (e.g., a GUI for creating application scenarios), the drawback is that these are centralized platforms that run under a single authoritative domain. This approach does not fully profit from the Web

architecture and also conflicts with the spirit of an open, decentralized platform. We argue that users should be empowered to participate in a Web of Things, just as they are in the World Wide Web. In this thesis, we support this undertaking by providing several contributions.

1.4 Contributions

This thesis provides three main contributions to a future Web of Things, each addressing a different aspect. Common to all contributions is that they leverage the existing Web architecture and avoid central hubs to foster an open and decentralized platform.

1.4.1 Searching the Physical World

Similar to today's Web, a key service for the Web of Things is expected to be a search engine that allows users to search for real-world entities with certain properties. While the traditional Web is dominated by static or slowly changing, unstructured content that is being manually created by humans, a key feature of the Web of Things is rapidly changing, structured content that is being automatically produced by sensors. Thus, a search engine for the Web of Things has to support searching for structured and rapidly changing content, which is a key challenge given that existing Web search engines are based on the assumption that most Web content changes slowly, such that it is sufficient to update an index at a frequency of days or weeks. This is clearly insufficient for the Web of Things, where the state of many real-world entities changes within minutes or even seconds.

The contribution is a prototypical real-time search engine for the Web of Things that addresses the key challenge of scalable search for rapidly changing content while leveraging existing Web infrastructure. Essentially, our search engine supports the search for real-world entities with a user-specified current state. For example, it could be used to search for rooms in a large building that are currently occupied, for bicycle rental stations that have currently bikes available, for currently quiet places at the waterfront, or for current traffic jams in a city. Contrary to other approaches, our solution is based on an open architecture that requires neither a global view of the world's state nor a limitation of the search space, while still providing accurate results in real time. We evaluate our approach using a real-world dataset that

features data gathered from a bicycle-sharing system over the course of several months.

1.4.2 A Framework for the Web of Things

In the Web of Things, *sensors* and *actuators* play a central role, because they constitute the physical interface between the virtual and the physical world: They enable us to capture and change aspects of the physical world, possibly in real time. Unfortunately, the current Web architecture does not address some of the key features that are often required when interacting with sensors and actuators; in particular:

- The *historization* of past sensor readings as well as actuator values. The former is often required as a basis for queries, while the latter may be required to provide a log of manipulations of the surroundings.
- The support for *notifications* based on specified events. This obsoletes the need for continuous polling of certain resources in order to detect events and enables real-time reactions that are based upon the occurrence of such events.
- The support for *simple queries*, which enables users to analyze and export data like past sensor readings.
- The ability to *compose* and run simple application scenarios in a decentralized way.

We address these issues by suggesting new primitives that can be used to extend today's Web architecture. We implemented these primitives in a prototypical framework and evaluate them using several application scenarios. The features of the framework could be distributed among connected devices but also among different cloud computing services. In contrast to existing solutions, our approach does not rely on a central hub but recommends the extension of today's inherently decentralized Web architecture.

1.4.3 Extending the Web Down to Constrained Wireless Devices

The usual approach of connecting resource-constrained wireless devices to the Web is to use an application-level gateway that mediates be-

tween the devices and the requests from the Web. This enables devices to save valuable energy by utilizing application-specific protocols and optimized hardware, such as dedicated low-power radio interfaces. However, this approach impedes the interoperability and mobility of devices because the gateway is usually application-specific and uplinks to the Internet based on the utilized low-power radio technology are not broadly available.

In contrast, we leverage the ubiquity of Wi-Fi access points and the interoperability of the HTTP protocol. This approach has become possible due to the improvements of available hardware. So-called ultra low-power IEEE 802.11 transceivers claim to achieve an operating time of years on batteries, for event-based interaction, while working with the existing Wi-Fi infrastructure.

We evaluate whether it is technologically feasible that the resource-intensive protocols and data formats used for today's Web (such as TCP, HTTP, and JSON) are used to communicate with battery-powered, wireless, resource-constrained devices. Using a loosely coupled approach, we can enable seamless interaction of sensors, actuators, and everyday objects with each other and with the Web. Our experimental results show that low-power Wi-Fi modules can achieve long battery lifetime despite using the verbose Web architecture. We argue that "connecting" the physical world directly to the Web, thereby avoiding additional infrastructure such as gateways, reduces complexity, enables mobility, simplifies interoperability, and therefore fosters a wide and rapid deployment of a Web of Things.

1.5 Thesis Outline

The structure of this thesis is as follows: Our approach of searching the physical world in real time and using an open, distributed approach that leverages the existing Web infrastructure is presented in Chapter 2. In Chapter 3, we introduce our prototypical framework for the Web of Things. The third contribution of this thesis, a concept and evaluation of connecting everyday objects to the Web using programmable low-power Wi-Fi modules, is portrayed in Chapter 4. Finally, this thesis concludes with Chapter 5, wherein we summarize our contributions, discuss their limitations, and provide an outlook on possible future work.

2 Searching the Physical World in Real Time

Our world is being pervaded with an increasing amount of sensors that can exhibit their readings on the Web, enabling us to monitor an ever-growing fraction of the world's phenomena, independent from physical proximity and possibly in *real time*. Traffic congestion data gathered from mobile phones, data from bicycle and car sharing providers, and meteorologic data from weather stations are just a few examples of sensor-acquired data that can be accessed on the Web. We assume that these developments will eventually lead to a comprehensive real-time view of the physical world, available through the Web. Making this emerging part of the Web searchable will therefore enable users to search the physical world, at a global scale.

Today, using a search engine is the dominant way to find information on the Web. In a similar way, we assume that the search for *physical entities* based on their *current state* (which is automatically gathered by sensors) will be a popular use case for a future Web of Things. For example, users might search for nearby restaurants that are currently well-attended and quiet. However, the current architecture of Web search engines does not scale to the expected number of sensors and the dynamics of sensor data. We assume that the number of connected sensors will be much higher than the number of people connected to the Internet, and sensors may update their readings on a much higher frequency than manually created content is updated.

In this chapter, we present a novel approach for searching in dynamic, distributed data in real time, and its application to the Web. In contrast to existing approaches, our prototypical search engine for the physical world is based on an open architecture which does neither require a global view of the world's state nor a limitation of the search space, while still providing accurate results in real time. We evaluate our approach using a real-world data set featuring data from a bicycle sharing system over the course of several months, both conceptionally in Matlab as well as using our prototypical implementation of a search

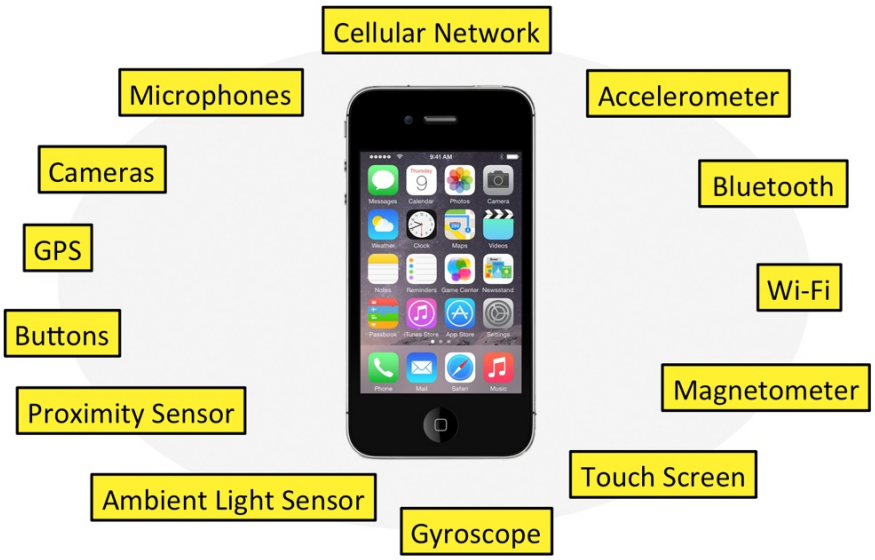


Figure 2.1: Physical interfaces of an Apple iPhone 4S that can be used to sense the environment (product image source: Apple).

engine for the Web of Things called Dyser.

Parts of this chapter have been published in [99, 100, 101, 102, 103].

2.1 Background

In the following sections, we provide some background on aspects of a real-time search engine for the physical world.

2.1.1 Searching the Web

Today, search engines like Google[1] and Bing[2] can be considered a vital part of the Web infrastructure, allowing the user to search for Web sites, images, news, blog posts and other information on the Web. This data is often *unstructured*, such as free text on Web pages. As the Web is inherently decentralized, users depend on search engines in order to quickly find relevant documents in the vast amount of publicly available data.

Current Web search engines use a simple search interface that is mostly based on keywords in order to find documents on the Web that contain those keywords. Since the list of results is usually far too long to be fully considered by a user, results are sorted according to their expected relevance and presented in descending order to the user. The

[1] http://www.google.com
[2] http://www.bing.com

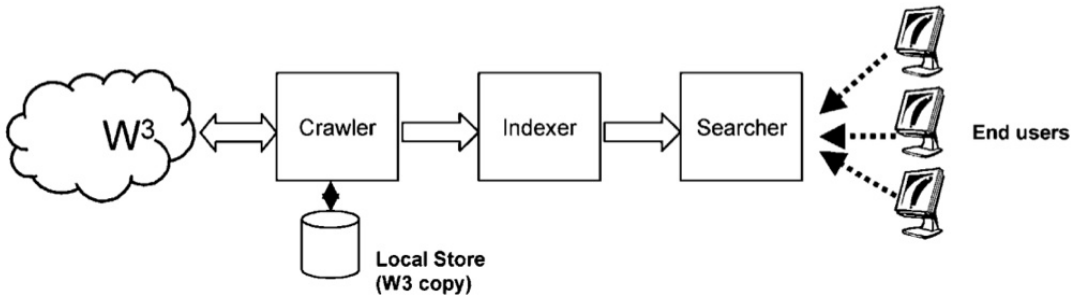


Figure 2.2: Search Engine Reference Model (source: [108]).

calculation of relevance is thus a crucial step of a Web search engine. For example, Google claims to consider over 200 so-called "signals" in order to determine the relevancy of a Web page [104]. Signals include terms on Web pages, the age of content, and the PageRank [105]. Web search engines usually deliver their results in real time (sometimes even while the query is typed), which is impressive given the large dimensions of the search space.

The global dependency of users on search engines in order to find information on the Web was demonstrated unexpectedly on August 16th, 2013, when a short downtime of google.com caused a drop of 40% in Web traffic [106].

2.1.2 Basic Architecture of Web Search Engines

The following summary of the basic architecture of a Web Search Engine is based on [107, 108, 109]. Note that in practice, the architecture is much more complex: Additional components are required, and they need to be distributed and replicated.

The World Wide Web (the "Web") originally consisted solely of Web pages distributed among Web servers. Web pages are defined in HTML[3] [94, 95] and interlinked using *hyperlinks*, which are unidirectional pointers specified on the source page that refer to a target page. The user can navigate through the Web (known as surfing the Web) by following these hyperlinks in a Web browser. Today, the Web is much more than a collection of documents. For example, it includes applications, APIs, gateways to other services, and connected devices.

[3]CSS and JavaScript were introduced a few years after HTML and complement the definition of a Web page.

A Crawler (also called *spider*, *robot*, or *bot*) is an application that periodically or continuously "visits" Web pages, and downloads a copy to a local database. As a starting point, a crawler is given a list of well-connected Web sites to visit. Each time a crawler downloads a Web page, it extracts its contained hyperlinks and adds them to the list of pages to visit. Using this approach, a crawler is able to eventually visit all Web pages that can be found when starting from the initial set. Note that building a crawler is not a trivial task, as issues like download rate limitation, scalability, and revisiting strategies have to be addressed.

The Indexer processes the local copy of the Web that has been generated by the crawler and creates data structures that can be used to efficiently resolve search queries. Without this step, the search engine would need to linearly search all documents in order to process a query, which does not scale with respect to both the number of search requests and the size of the search space. Data structures generated usually include an *inverted index*, which stores for each word extracted from the set of documents a list of references to all documents that include it. Each entry in these lists also holds additional information such as the position of the word in the document, its frequency or the overall impact factor. The latter is computed based on multiple *signals*, i.e. features that can be extracted from the document itself or from other sources. An example of a signal is the well-known "Page Rank" of a Web page [107]. The impact factor is used to sort the final list of results according to their expected relevance to the user.

The Searcher is in charge of processing the search requests. For this, it receives and parses search requests, resolves the requests using the index, and finally returns the ranked list of results to the user. Ranking is performed according to the expected relevance of results for the user. Due to the size of the search space, ranking is an essential aspect of a search engine, since the list of results is usually far too large to be completely inspected by the user.

Search Language Today's Web Search Engines queries are typically based on unstructured keywords, and do not rely on sophisticated query language such as SQL [110]. To search, the user specifies a list of words separated by spaces for a search request, and the search engine then

returns list of documents that contain all or some of these keywords, sorted by relevance. For example, the search request for `foo bar` will return all pages that both contain the word "for" and "bar".

In order to refine the search, one can also use search operators, which are *(key, value)* pairs denoted as `key:value` (there are also other operators, which we will discuss below). Using these search operators, the search for structured data is supported, to some degree. For example, Google supports `intitle:foo` and `inurl:bar`, which will return only Web pages that contain the term "foo" in its title and "bar" in its URL, respectively. One can also restrict the search to certain (sub-)domains using the "site" operator. For example, `site:example.org foo` will only return results from the domain example.org and its subdomains (e.g., www.example.org).

Search engines usually also support additional operators that follow a different syntax. For example, prepending a term with a minus sign will invert its semantic: `foo -bar` will return only pages that contain "foo" but not "bar". `foo OR bar` will return pages that contain either "foo" or "bar", but not both in the same document. Using quotes around (parts of) a search term requires the given words in the exact order, e.g. `"bar foo"` will only return documents that contain the word "bar" followed by "foo" but not vice versa.

2.1.3 Searching the Physical World

Given the vision of the Web as an interface to the physical world, we expect that just like for today's Web, *search* will be an important service. However, we assume that searching the physical world will significantly differ from searching for documents. For example, users might want to search for nearby *italian restaurants* that are currently *quiet* but also *well-attended* in order to spontaneously find a nice place to eat.

More general, we expect that users are interested in searching for *entities* of the physical world (e.g., places, objects, creatures) with a specific *current state* rather than in *sensors* with a specific *raw reading*. So instead of searching for loudness sensors with a current reading below 30 *dB*, we expect that users will rather be interested in searching for *places* which are *quiet*. The dynamic properties of entities are automatically gathered by associated sensors or deduced from their readings.

In the example of the search for a restaurant, there are two required sensors, one that measures the noise level and one that measures the density of people. Such sensor readings could either be provided by the restaurant owner or be acquired by leveraging the built-in sensors of mobile phones carried by visitors of this restaurant. In the latter case, that could be the microphone and the Bluetooth interface[4], for example. This example also illustrates that, similar as for Web search engines, the expected users of a search engine for the physical world are mostly average Internet users which use the service for their everyday activities, and not domain experts. Additionally, we do not assume that the outcome of a successful search necessarily implies a physical interaction between the user and the found entity. Therefore, a geographic limitation of search queries cannot be required by the search engine.

The major challenges of a search engine for the physical world can be summarized as follows:

High Dynamics of Sensor Data. The central challenge in building a search engine for the physical world is the dynamics of sensor data. Sensor readings are expected to change at much higher frequencies than Web pages, rendering the usual approach of indexing published data useless: This approach does not scale to large amounts of real-time data, as the the index would be outdated as soon as it is build, which would result in wrong results.

Large Amount and Diversity of Sensors. We expect that the number of sensors in a future Web of Things will greatly surpass the number of humans. Even today, there are more devices connected to the Internet than people [112]. Given the open architecture of the Internet, there will be a large heterogeneity of connected sensors.

Distributed Publishing of Sensor Data. Given the global challenge of monitoring the physical world, sensors will be inherently distributed: Technically, geographically and administratively. Technically since they will use different underlying systems, geographically since they will be distributed over many different places, and administratively since they will be under the control of different authoritative domains.

[4] A Bluetooth inquiry detects "visible" Bluetooth devices such as mobile phones, and could be used to estimate the number of people nearby [111].

No Accepted Standards. There are currently no accepted standards for publishing sensor data on the Web. Contrary to the first Web search engines which could rely on the existing Web, a search engine for the Web of Things faces a chicken-and-egg problem.

2.1.4 Real-Time Search Engine

In this chapter, the term *real time* is used in the context of perceivable delay, not in the context of real-time constraints on computing tasks. It addresses two aspects of delay:

- *Real-time computing* (also denoted as something is performed *in real time*): the result of the operation is returned with imperceptible or negligible delay.
- *Real-time data*: the presented data is up to date. For example, data based on sensor readings is considered to be real-time when it reflects the current state of the environment.

The term "real-time search engine" is used to refer to a search engine which satisfies both of the following requirements:

a) it delivers search results that are based on *real-time data*

b) it computes search results *in real time*

Note that traditional search usually engines only fulfill b). There are also examples of real-time search approaches which only fulfill a) [113, 114]. When we soften the definition by slightly increasing the tolerated delays we call this *near real time*.

2.1.5 Dynamics of the Search Space

The set of elements that can be searched is called the *search space*. There are two dimensions of changes in the search space:

- *Contentual changes* are modifications of the contents of elements, such as updates to indexed Web pages.
- *Structural changes* changes are modifications of the "visibility" of elements. Creating, deleting, copying and moving an element to a new location (changing its address) are all considered structural changes.

Both types of changes may be highly dynamic. However, for a search engine for the physical world, we expect contentual changes (that are based on sensor readings) to be more frequent than structural changes.

2.2 Requirements

We can now state our requirements for a real-time search engine for thy physical world that is based on a future Web of Things:

Search Results based on Real-Time Data Obviously, a real-time search engine for the physical world needs to return results which are based on real-time data, i.e., the results reflect the current state of the physical world (Sect. 2.1.4). Of course, results also need to match the *search term*. While this is trivial for search engines that rely on data which is mostly static, it may be challenging with real-time data as the current state might not be known directly or it may change during the processing of the search request.

Computation in (near) Real Time As defined in Sect. 2.1.4, a real-time search engine also needs to *compute* its results in real time or near real time. Note that there are approaches of "real-time" search engines that deliver results based on real-time data but may require a considerable amount of time to do so.

Support for Sensor Data As sensors constitute the interface between the physical and the virtual world, the search engine needs to support sensor data. As we assume that a wide variety of sensors will be available on the Web, the search engine needs to support sensors of various types. Furthermore, it has to support the high dynamics of sensor data.

Search for Entities rather than for Sensors We argue that when users will search the physical world, they will expect the search engine to return *entities* (i.e., people, places, things) whose dynamic properties currently match the search query rather than *sensors* which currently match the search query. The attributes of an entity can be updated automatically by sensors attached to or associated with the entity. For example, we expect that users will be interested in meeting rooms which are currently *empty* rather than in occupancy sensors which currently read "none".

Usability We want to offer the search service to a broad audience, rather than just domain experts. This implies that we cannot expect sophisticated knowledge from the users, concerning query languages or the structure of published sensor data, for example.

Scalability The search engine needs to able to scale to large numbers of sensors, high update rates and also high query rates. Both the number of supported sensors and the number of supported search requests need to scale to numbers which can be compared to the amount of Web pages today and the number of Web search requests, respectively.

Open, decentralized System The approach should not require users to commit themselves exclusively to our search engine. Providers of sensor data should be able to publish those data individually, without the the need to stream the sensor readings to a central sink (as it is done with Twitter, for example). Multiple, competing real-time search engines should be able to utilize published sensor data, much like it is for Web search engines and published Web pages today. Note that this decentralized publishing of data implies that our search engine has no *global view* of the world.

To summarize, a search engine for the physical world needs to support the search for *dynamic*, *distributed*, *structured* data in *real time*: Sensors automatically acquire data of the physical world at short intervals, so the search engine needs to support the search for dynamic data. For scalability reasons, one cannot assume that all these sensor readings are streamed to the search engine in real time, hence these data are distributed and the search engine has no global view of the current state of the world. In contrast to data published in text documents, data published by sensors will usually be structured, i.e., following some schema. Finally, just like for current Web search engines, users will expect their results in almost real time. However, enabling real-time search on dynamic data is challenging: The traditional approach of creating an index of the search space at regular intervals, which is then used to resolve search requests, does not scale with respect to the expected number of sensors and their update rate. Also, as argued before, the search engine has no global view, so it cannot answer a search request by only performing local operations.

2.3 Approach

In this section, we introduce our approach for searching in dynamic, distributed, structured data in real time. While the approach can be implemented using the Web architecture, its concepts are independent from a specific implementation. For this reason, we first present our approach on a conceptual basis. Its application to the Web, using a prototypical search engine, is presented later in Sect. 2.6.

The key challenge that needs to be addressed in constructing a search engine for the Web of Things is the anticipated huge size and extreme dynamics of the search space. Extrapolating the current trend of instrumenting objects, places, and even people with sensors several years into the future, we can expect that the Web of Things may contain orders of magnitude more sensors than currently existing Web pages. Moreover, the output of these sensors is highly dynamic. In contrast, the large majority of today's Web is static in the sense that Web pages are changed at time intervals orders of magnitude longer than the update rate of sensors. Thus, traditional indexing approaches are insufficient as an index would be outdated as soon as it has been constructed.

There are two fundamental approaches to construct a search engine for the Web of Things. With a *push* approach, sensor output is proactively pushed to a search engine, such that the search engine can resolve queries based on that data. With a *pull* approach, only upon a user entering a query the search engine sends the query to the sensors to pull the relevant data.

In the Web of Things we can expect substantially more sensors producing readings than users typing queries, and sensors would produce data at a much higher rate than users can type queries. Hence, the pull approach can be expected to generate a substantially smaller communication volume between sensors and search engine than the push approach. More importantly, a push approach might not scale to the expected dynamics and size of a global Web of Things. Finally, not having to push sensor readings to a central sink simplifies tasks for sensor publishers.

For this reasons, we are using a pull approach, which implies that our search engine has no global view of the current state of the world. To answer a query, the search engine needs to contact relevant sensors (i.e., sensors that could possibly read the searched state) at the time a query is posed, in order to determine whether they currently match

the query. Entities whose associated sensors do not read the searched state are excluded from the result set, which is returned to the user as soon as enough matches have been found. Note that indexing current sensor readings is not an option, as the index would be outdated as soon as it was built due to the anticipated frequency of changes in sensor readings.

Even though this query resolution process can be optimized by contacting multiple sensors in parallel, it is not scalable with respect to network traffic – given that the number of possible results is significantly larger than the number of actual results, many sensors would be contacted unnecessarily by the search engine when processing a single query. However, instead of contacting the sensors in an arbitrary sequence, we can contact them in an order that reflects the probability that they are currently matching the query. We call this approach *sensor ranking* and argue that it enables scalability, provided we can order the list of relevant sensors sufficiently well.

2.3.1 Sensor Ranking

The core concept of our approach is denoted as *sensor ranking*. It works as follows: Each sensor publishes its current reading, along with associated meta-data. Included with this data is a *prediction model* which can be used to compute the probability that the sensor will sense a given value at a given point in time. At regular intervals, the crawler of our search engine is looking for published sensor data. Whenever a sensor description is found, its data, including its prediction model, is stored in a local index. This process is analog to the crawling and indexing process of traditional Web search engines, except that the indexed data follows a given format.

Let us now assume that a user is searching for loudness sensors which currently sense the state *quiet*. In order to resolve this search request, the search engine will evaluate the prediction models of all indexed loudness sensors in order to determine the probability for each sensor that it is *currently* reading the state *quiet*. This list is then sorted according to the outcomes of the prediction models, starting with the sensor that is most likely to read the searched state. Beginning with the top-ranked sensor, the search engine contacts the sensors over the Internet to determine their current readings. Sensors that currently sense the searched state (i.e., *quiet*) are kept in the list, sensors that

do not are removed. This process continues until a sufficient number of results is found. The list of results is then returned to the user. Provided that the prediction models map reality sufficiently well, it should be possible to significantly reduce the number of sensors contacted per query.

The main assumption in this approach is that there are enough sensors which offer a sufficient level of predictability. While this may not be the case for arbitrary sensors, many phenomena in the physical world feature periodic characteristics, especially those related to people. For example, in a person's life, daily, weekly and yearly cycles can usually be identified. Research has shown promising results regarding the predictability of human behavior [115, 116]. For this reason, we focus on sensors related to human behaviors.

2.3.2 Basic Principle

Based on the concept of sensor ranking, the search for *entities* by their *current state* can be realized as follows: Each entity publishes a textual description and at least one association with at a sensor. Each sensor publishes its current reading, along with associated meta-data and a prediction model. The state of an entity is defined as the combined outcome of the readings of its associated sensors. Note that there is a *many-to-many* relationship between sensors and entities. Entities can use multiple sensors, and a single sensor may be used by multiple entities.

At *indexing time*, our search engine indexes data published by sensors and entities, including their relationships. This data, consisting of both unstructured data such as free text and structured data such as the prediction models is then stored in a local index. Note that all indexed data is assumed to change at much longer intervals than sensor readings, which are not indexed. For this reason, and given the fact that indexing a large data set is an expensive operation, re-indexing needs to be performed only at intervals that approximate the expected change rate of such slowly-changing data.

At *query time*, our search engine receives a search request by the user, which consists of at least a searched state and possibly some keywords. The searched state in turn consists of tuples of *(sensor type, required state)*. For example, "`room IFW occupancy:empty`" might denote entities that feature the keywords "room" and "IFW" and an occupancy

sensor that states that this entity is currently "empty". In order to process a search request for a given number of results, the search engine performs the following steps:

1. All entities that do not match the given keywords or sensor types are filtered out. The remaining entities are ranked according to their expected relevance to the user. The process starts with the first x entities.

2. For the entities currently considered, the probability that the entity currently matches the searched state is computed. For each entity, the prediction models of its associated sensors are evaluated, using the requested states. The probability of an entity is then computed as a function of these outcomes.

3. The considered entities are re-sequenced (in descending order of probability).

4. Beginning with the top-ranked entity, the search engine contacts the entity sensors to check whether their values actually match the searched state. Only entities that fulfill the search request are added to the result set.

5. If enough matching entities have been found, the list of results is re-sorted according to the relevance criterion in 1) and returned to the user. If not, the process continues from step 2) with the next x entities.

While this approach is based on prediction models, only entities matching the requested state(s) are returned to the user. The quality of prediction models will only influence the efficiency of the search engine, and not introduce wrong results. This approach also leverages the fact that users of search engines are typically *not* interested in all results, since these are typically far too many to check manually.

Note that we are using two different ranking approaches: *sensor ranking*, which affects the order in which sensors are contacted in order to test whether they are actually reading the searched state; and *relevance ranking*, which adjusts the order in which elements are displayed to the user. We need to combine both these approaches – sensor ranking for efficiency and relevance ranking to fulfill the expectations of the user. However, this combination is admittedly a trade-off between efficiency and relevance.

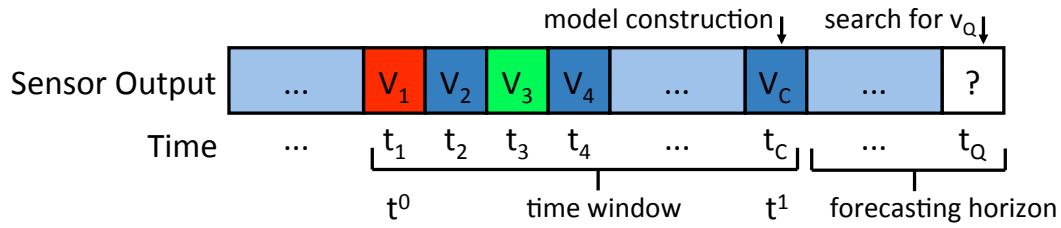


Figure 2.3: Time series of sensor readings, and time window and forecasting horizon for prediction models (based on [100]).

In the following subsections we formally define our system model and detail the basic operation of the search engine. The formal definitions are based on [100, 101, 102].

2.3.3 System Model

In our system model, a sensor s is a function

$$s : T \mapsto V \tag{2.1}$$

where T denotes real time and V the set of possible sensor values. V is assumed to be a finite set of unordered discrete states that an entity can be in (e.g., a room entity could include an occupancy sensor that can yield one of two values "occupied" or "empty"). Note that the mapping of raw sensor readings to discrete states may introduce subjectivity, as it often requires interpretation of the sensor data. We let the operator of the sensor implement this mapping, as he has comprehensive information about its context. We do not assume that elements of V follow some order, as we also want to support sensors that sense inherently unordered phenomena, such as sensors that return the current activity of a user.

Each sensor is associated with a type and optionally further structured meta information such as a location. For example, the following function can be used to obtain the type $\in Y$ of a sensor:

$$\text{type} : S \mapsto Y \tag{2.2}$$

A prediction model for a sensor s is a function

$$P_{s,t^0,t^1} : T \times V \mapsto [0, 1] \tag{2.3}$$

The parameter $t^0 \in T$ refers to the time of the first considered sensor reading, $t^1 \in T$ refers to the time when the model has been constructed,

meaning that all sensor values $s(t^0 \leq t_i \leq t^1)$ have been available for the construction of the model. The idea behind constraining the construction of the prediction model to the time window $[t^0, t^1]$ is that sensor values from the distant past are typically bad indicators for the future output of the sensor. Also, using a time window instead of all past data typically reduces the resource consumption (i.e., execution time and memory footprint) of the model construction.

Given a point in time $t > t^1$ and a sensor value $v \in V$, $P_{s,t^0,t^1}(t, v)$ is an estimate of the probability that $s(t) = v$ holds. We call $t - t^1$ the *forecasting horizon*.

An entity $e \in E$ is associated with one or more sensors,

$$\text{sensors} : E \mapsto \mathcal{P}(S) \tag{2.4}$$

whereas we assume that each sensor of an entity e has a distinct type.

2.3.4 Basic Operation

Given the above definitions, we can now outline the basic operation of the search engine. At regular intervals, the search engine crawls the Web of Things. For each visited entity e_i, the search engine downloads and indexes the structured meta information including the prediction models of all sensors(e_i) associated with that entity.

A basic search operation issued at time t would specify the current state of sought entities at time t, where the current state of an entity is represented as a set of sensor types $y \subset \mathcal{P}(Y)$ and a mapping function $\text{val} : Y \mapsto V$ that maps sensor types in y to the requested values. That is, an entity matches the search if for each type $y_i \in y$ the entity contains a sensor s of that type with $s(t) = \text{val}(y_i)$.

To perform a search operation, the search engine first fetches entities e_i from the index which contain sensors of requested types. Next, for each fetched entity e_i a probability p_i is computed that the entity matches the search. As the search is conjunctive (e.g., the entity must contain a matching sensor for each type and value specified in the search), p_i equals the product of the prediction probabilities of the individual sensors (assuming independent sensors):

$$p_i := \prod_{s \in \text{sensors}(e_i) \wedge \text{type}(s) \in y} P_s(t, \text{val}(\text{type}(s))) \tag{2.5}$$

Next, entities e_i are sorted by decreasing values p_i. Beginning with the entities with the largest p_i, the sensors of the entities are consulted

to check whether indeed $s(t) = \text{val}(\text{type}(s))$ holds. Entities where all requested sensor states match are returned as search results until enough matching entities have been found.

Note the difference between sensor ranking and the relevance ranking of traditional search engines. The former tries to optimize the performance of the search engine, while the latter tries to optimize user satisfaction. As detailed in Sect. 2.3.2, we combine both in Dyser: sensor ranking for the internal search process, and relevance ranking when displaying the results to the user.

2.3.5 Prediction Models

As defined in Sect. 2.3.3, a prediction model returns the probability that the corresponding sensor will read a given state at a certain point in time. For this thesis, we consider three different prediction models, ranging from a very simplistic approach to a complex prediction model that considers multiple periodic processes. Each prediction model only depends on past readings of its corresponding sensor. This approach makes the creation of prediction models independent from external data, thus simplifying this process.

2.3.5.1 Aggregated Prediction Model

Probably the most simple prediction model computes the fraction of the time during which the sensor output equals v within the time window $[t^0, t^1]$, that is:

$$P_{s,t^0,t^1}(t, v) = \frac{1}{t^1 - t^0} \int_{t^0}^{t^1} \chi(s, t, v) \mathrm{d}t \tag{2.6}$$

where $\chi(s, t, v)$ is an indicator function that returns 1 if the sensor s reads v at time t, and 0 otherwise:

$$\chi(s, t, v) = \begin{cases} 1 & : \quad s(t) = v \\ 0 & : \quad \text{else} \end{cases} \tag{2.7}$$

For example, if the sensor output was v during the whole time window $[t^0, t^1]$, then the probability computed by the above prediction model equals 1. Note that the output of the prediction model is independent of the actual point in time t of the search. Hence, we call this model the *aggregated prediction model* (APM).

2.3.5.2 Single-Period Prediction Model

A more elaborate model would take into account the time t of the search. For this, we assume that the sensor output is dominated by a periodic process of period length L. The sensor output is expected to repeat after this single dominant period. For example, it is reasonable to assume that the occupancy pattern of a room is likely to repeat every week, that is, L equals one week. If we assume that the time window size $t^1 - t^0$ is an integral multiple of L, i.e., $NL = t^1 - t^0$ for integral N, and that the forecasting horizon $t - t^1$ is smaller than L, we can perform a prediction as follows:

$$P_{s,t^0,t^1}(t, v) = \frac{1}{N} \sum_{1 \leq i \leq N} \chi(s, t - iL, v) \tag{2.8}$$

Here, we consider all points in time t' contained in the time window that have the same offset as t with respect to period length L, i.e. $t' \equiv t \mod L$ such that $t' \in [t^0, t^1]$. Under the assumptions stated above, this is the case for $t' = t - iL$ for all $1 \leq i \leq N$. The output of the prediction model equals the fraction of the instances of t' for which $s(t') = v$ among all t'. As this model assumes a periodic process with a single period, we call this model the *single period prediction model* (SPPM). Note that a spectral analysis of the sensor data or a periodicity indicator as given in [117] can be used to automatically derive the dominant period length L for a given data set $s([t^0, t^1])$. Alternatively, L could be derived from domain knowledge.

2.3.5.3 Multi-Period Prediction Model

In practice, sensor output is often influenced by many periodic processes with varying period lengths. Consider a meeting room that may host a group meeting every Monday and a conference call on a Tuesday every other week. In this example, sensor output is related to two periodic processes with period lengths of one week and two weeks. To support such multi-period processes, we use the following approach. At first, we automatically detect periodic patterns in the time window using a variant of an existing algorithm [118]. As a result, we obtain a list of periodic patterns of the form (l, o, w, p), where l is the period length of the pattern, and o is an offset in the period such that the sensor output $s(kl + o)$ equals w for integer values k with probability p. For example, the pattern (one week, 2, occupied, 0.5) means that every second day

of the week (i.e., Tuesday) the probability of a room being occupied is 0.5.

To make a prediction, we first filter all patterns that match the search time t, that is, we retain a pattern (l, o, w, p) if and only if there exists some integer k such that $kl + o = t$ and $w = v$, where v is the sought sensor value. Assuming N such patterns exist, we perform a prediction as follows:

$$P_{s,t^0,t^1}(t, v) = \max_{1 \leq i \leq N} p_i \tag{2.9}$$

where p_i is the probability of periodic pattern i. In order to mitigate the effects of sudden changes in periodic patterns in the data set, one may specify a lower and upper bound (ω_{min} and ω_{max} respectively) for the number of instances of periodic symbols considered. We call this model the *multi-period prediction model* (MPPM).

2.3.6 Coping with Low-Quality Prediction Models

In order for sensor ranking to work efficiently, prediction models need to map the characteristics of underlying sensors sufficiently well. Prediction models that do not reflect the actual characteristics of a sensor will result in unnecessary communication, degrading the performance of our search engine. Recall that our approach will always return accurate results, despite the quality of indexed prediction models. In order to mitigate the effects that erroneous, outdated, or malicious prediction models may have on the performance of our search engine, we enable the assessment of the quality of indexed prediction models and also their adjustment with respect to sensor ranking.

To assess the quality of a prediction model, we introduce a metric that reflects the ranking error of a sensor in a list of potential results. A query for sensors reading v_Q at time t_Q will cause the search engine to produce a rank list of potential results, sorted by the outcomes of the sensors' prediction models:

$$S^Q = s_1, s_2, s_3, ..., s_m \tag{2.10}$$

The sensor s_1 with the highest probability of reading the value v_Q at time t_Q is ranked first. For a perfect ranking, a request for k results will only require k lookups of sensors' current readings ($s_1 ... s_k$). Any communication with an additional sensor is either caused by a non-matching sensor ranked too high or a matching sensor ranked too low. Ideally, the ranked list of results S^Q is partitioned in matching sensors

(ranked high) and non-matching sensors (ranked low). The ranking is imperfect when S^Q contains at least one non-matching sensor that is ranked before a matching sensor. Since in practice, the search engine will stop contacting sensors as soon as it has acquired the sought number of k results, we formalize the ranked list of sensors S_k^Q that were contacted for a query Q in order to provide k results.

$$S_k^Q = \begin{cases} S^Q & : \ M_k = 0 \\ s_1, s_2, s_3, ..., s_{M_k} & : \ else \end{cases} \tag{2.11}$$

The rank M_k of the kth matching sensor is defined as

$$M_k = \begin{cases} \underset{1 \leq i \leq m}{\arg min}(\chi(s_i, t_Q, v_Q) * i) & : \ k = 1 \\ \underset{M_{k-1} < i \leq m}{\arg min}(\chi(s_i, t_Q, v_Q) * i) & : \ else \end{cases} \tag{2.12}$$

To compute the *ranking error* for each sensor s_i in S_k^Q, we summarize sensors that are ranked either too high or too low. For a matching sensor s_i, we we summarize the amount of non-matching sensors that were ranked higher. For a non-matching sensor, we summarize the amount of matching sensors that were ranked lower. The ranking error is negative when the sensor was ranked too high and positive when the sensor was ranked too low:

$$\begin{aligned} &\mathrm{re}(s_i, v_Q, t_Q) = \\ &\quad \begin{cases} -|\{s_j \in S_k^Q | j > i \wedge s_j(t_Q) = v_Q\}| : s_i(t_Q) \neq v_Q \\ \ \ |\{s_j \in S_k^Q | j < i \wedge s_j(t_Q) \neq v_Q\}| : s_i(t_Q) = v_Q \end{cases} \end{aligned} \tag{2.13}$$

In order to mitigate the effects of systematic misranking, we introduce an *adjustment process* which is based on the ranking error, that is used to adjust the outcomes of prediction models. This is a feedback loop that considers the current ranking error of a sensor s_i in order to minimize its future ranking error. For this, we compute an adjustment term $\mathcal{AT}$, that is based on a sensor s_i, sought reading v_Q, and query time t_{Q_j}:

$$\mathcal{AT}(s_i, v_Q, t_{Q_j}) = \mathcal{AT}(s_i, v_Q, t_{Q_{j-1}}) + \frac{\mathrm{re}(s_i, v_Q, t_{Q_j})}{|S_k^Q|} \tag{2.14}$$

The number of sensors contacted in order to return k results $|S_k^Q|$ is used to normalize the ranking error. The adjustment term for a sensor

s_i computed at time t_{Q_j} for value v_Q is then used to adjust the outcome of that sensor's prediction model at time $t_{Q_{j+1}}$:

$$\hat{\mathrm{P}}_{s_i}(t_{Q_{j+1}}, v_Q) = \mathrm{P}_{s_i}(t_{Q_{j+1}}, v_Q) + \mathcal{AT}(s_i, v_Q, t_{Q_j}) \tag{2.15}$$

The adjusted probabilities of the prediction models $\hat{\mathrm{P}}$ are then used to process a search query at time $t_{Q_{j+1}}$, which will in turn produce new adjustment terms for query time $t_{Q_{j+2}}$. The quality of the adjustment process is depending in the query rate: The more frequent it is invoked, the more accurate its adjustments are expected to be. To prevent inaccurate adjustments, the adjustment term needs to be reset when a new prediction model was created, or when a sensor was not contacted within a certain time frame. The adjustment process works independently of the used prediction model. In fact, it could be used without a prediction model at all.

2.4 Data Set used for Evaluations

In order to evaluate our approach, we require a data set that satisfies our basic assumption, which is the existence of periodic patterns in sensor data. Ideally, it should feature data related to the behavior of people and could also provide an actual use case for our search engine. Since we did not find a data set that satisfied our requirements at the time this evaluation was conducted, we collected a real-world data set using publicly available data. It was gathered over the course of several months from Bicing, a bicycle-sharing system.

The evaluations presented subsequently in this chapter are based on the Bicing data set. Both the service and the data set are discussed in this section.

2.4.1 The Bicing Service

Bicing [119] is a bicycle-sharing service located in Barcelona, Spain. It was started in March 2007 and in spring 2009 (the time of the collection of our data set), it operated about 6'000 bicycles, which could be rent

[5]Image by Borinot bcr, source:
http://en.wikipedia.org/wiki/File:BicicletaBicing.JPG

[6]Image by Hank Chapot, source:
http://en.wikipedia.org/wiki/File:Barcelona_bike_program.JPG

[7]Image by Marc Belzunces, source:
http://commons.wikimedia.org/wiki/File:Furgo_bicing_bcn.JPG

Figure 2.4: Key components of Bicing (clockwise, starting from top left): A Bicing bicyle[5], a rental station with attached bicycles[6], and a truck with a trailer full of bicycles[7] which is used by the operator to redistribute bicycles between the stations.

at 400 stations distributed throughout the city. Users were required to subscribe to Bicing, and the service was restricted to residents of Spain in order to mitigate conflicts with existing bicycle rental services for tourists. Bicyles could be rent at and returned to any of the existing stations. The process was automated, requiring only the use of the personal Bicing card.

Unlike commercial bicycle rental services, *Bicing* aims at extending the public transport service in Barcelona. This is not only reflected by the large number of bicycles and rental stations, but also by the pricing scheme, which enforces short hire periods. At the beginning of 2009, the following rules applied: the first 30 minutes were free of charge, each additional 30 minutes cost 0.50€ up to a maximum allowed rental time of 2 hours. Extending the maximum rental time lead to a penalty fee of 3€ per additional hour and a warning. If the user did not return his bicycle to a station within 24 hours, a surcharge of 150.00€ was applied on his credit card. After three warnings, the provider would exclude the user from further service. The mandatory yearly subscription cost 30€. Additionally, bicycles provided by Bicing feature no locks and can therefore be stored safely only at the Bicing stations. Fig. 2.4 depicts

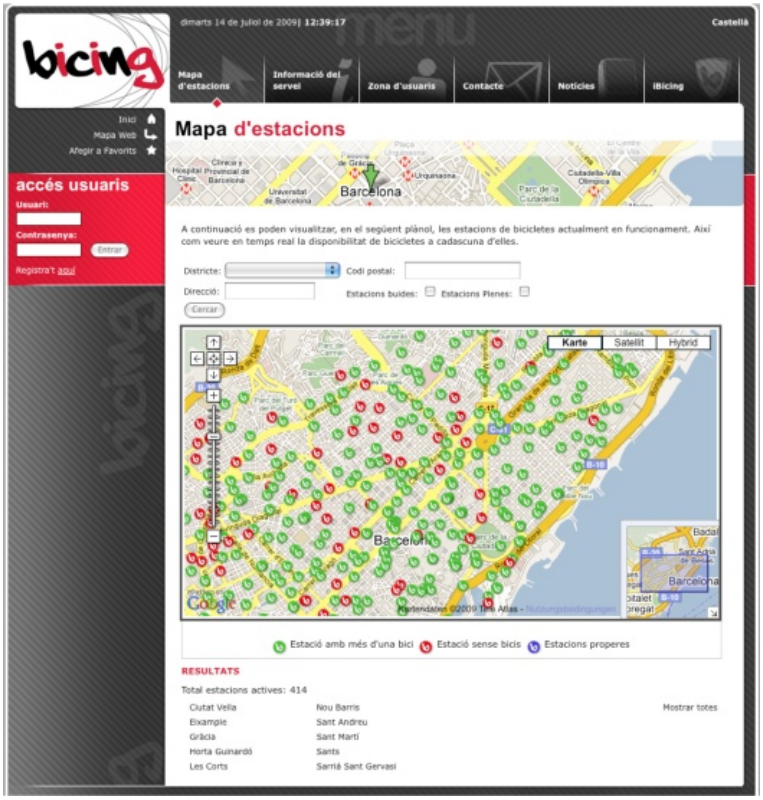


Figure 2.5: Homepage of Bicing at the time of the Study[8].

key components of the system.

2.4.2 Data Set

Bicing provides an interactive map on its homepage, which displays all bicycle stations (Fig. 2.5). For each station, both the current number of bicycles and the current number of free return slots are available. We utilized a simple script which fetched the HTML-code of the respective page every 5 minutes. This process was started on December 6th, 2009.

In a second step, a single log file containing only the station data was produced, based on the gathered files. Missing data was compensated with special markers, ensuring that for each time slot data from all stations is available. This single log file was then cropped to data from January to May 2009. Stations which went in service during this period were excluded from the data set, therefore only stations which were in service during the total considered period of time are considered. This lead to a total number of 385 stations. The "gap" in time caused by the transition to central European summer time (CEST) on March 29th

[8]Screenshot taken on 14.7.2009.

was filled using the data of the preceding hour, in order to prevent conflicts with time-based prediction models.

For our evaluation, a "sensor" is considered to be a rental station, returning the number of currently available bicycles. For this, each rental station is mapped to a virtual sensor that outputs one of six discrete states. The utilized mapping scheme is depicted in Tab. 2.1. The final log file used for the evaluations was created based on this mapping scheme. To limit the amount of sensor data, three consecutive time slots of 5 minutes each were mapped to a single time slot of 15 minutes for the resulting log file. For this, the average number of available bicycles per time slot was considered.

available bicycles	state
0	none
1..5	1to5
6..10	6to10
11..15	11to15
>15	many
(error)	unknown

Table 2.1: Mapping of the number of available bicycles per station to discrete states.

2.4.3 Data Analysis

The processed data was then read into Matlab[9] for analysis. Fig. 2.6 visualizes the average number of sensors reading a certain state during the course of the week, considering the final data set. For example, changes are high that one finds a station with plenty of available bicycles in the early morning hours on a Tuesday, as indicated by the high number of sensors which read the state "many" during those times. Also note that one can clearly identify daily and weekly patterns for the state "none", for instance. What the Figure does not reveal directly is that errors in the data set rarely occur, and if they do, they affect all sensors at the same time.

The aggregated distribution of sensed states is displayed in Fig. 2.7. For each possible state except "unknown", the list of sensors is sorted according to the fraction of total considered time that they output the respective state, in descending order. For example, for the state "none", there exist a few sensors which will always output this state, while the vast majority of sensors will output this state less than 50% of the time.

[9] http://www.mathworks.ch/products/matlab/index.html

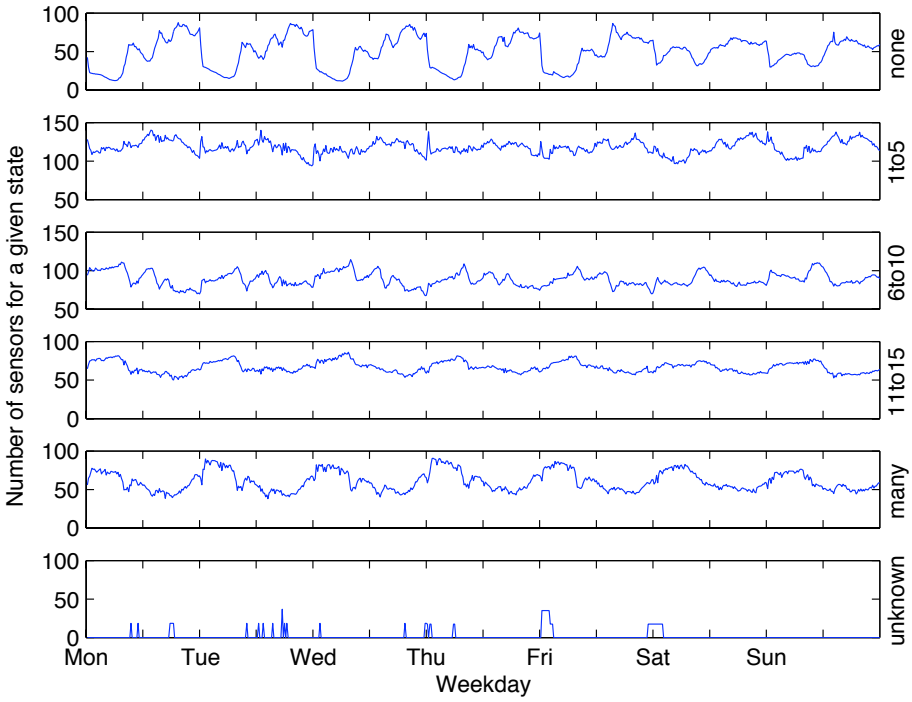


Figure 2.6: Average number of sensors reading a given state vs. weekday.

2.5 Evaluation with Matlab

In this section, we discuss the the simulative evaluation of our approach of searching in distributed real-time data from Sect. 2.3. All evaluations in this section where conducted using Matlab and are only considering the theoretical concepts, thus ignoring any aspects regarding the Web.

2.5.1 Simulation Setup

The simulations we conducted are all based on the preprocessed Bicing data set described in Sect. 2.4.2 and considered queries in a time frame from March 1st to May 31st 2009. The first two months (January and February) of the data set were used to construct the initial prediction models. The maximum forecasting horizon was set to one week, and the size of the time window for the prediction models was set to 8 weeks. All prediction models were periodically re-created when their maximum forecasting horizons were reached. Thus, prediction models "aged" during the course of a simulated week. Search requests for all possible sensor states were placed every 15 minutes throughout the simulation period. The number of required results k was set to 20. All simulations were conducted both with and without the adjustment process.

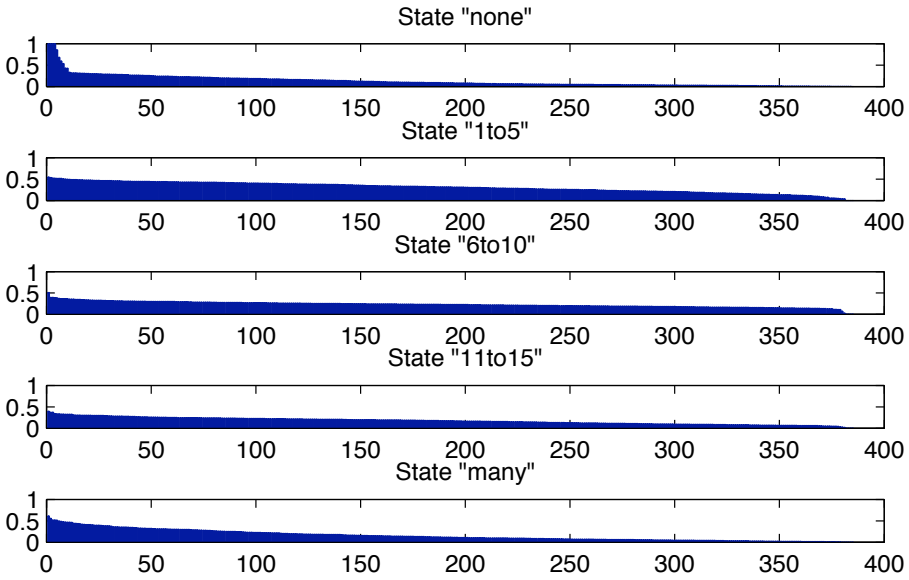


Figure 2.7: State time as a fraction of total time vs. sensors for each state.

The period length for the single-period prediction model (SPPM) was set to one week, as this turned out to be the dominant period length. For the multi-period prediction model (MPPM), ω_{min} was set to 4 and ω_{max} was set to 8. As a baseline, we included the results from a "prediction model" that outputs random values, resulting in a randomized sensor ranking that changes per time slot, state and sensor.

2.5.2 Performance Metric

In order to be able to assess the performance of our approach, we measured the *communication overhead* in our simulations, which is the number of contacted sensors M_k (see Eq. 2.12) divided by the number of requested results k for a given query. A communication overhead of 1 is an optimal result, indicating that no non-matching sensors were contacted. When the number of requested matches cannot be provided, the communication overhead is undefined and will not be considered when computing the average communication overhead. We formalized the communication overhead as

$$o(t,v,k) = \begin{cases} undefined & : \quad \sum_{i=1}^{m} \chi(s_i,t,v) < k \\ \frac{M_k}{k} & : \quad \text{else.} \end{cases} \tag{2.16}$$

where t represents the query time, v the sought state, m the total number of relevant sensors, and k the number of requested results.

2.5.3 Simulation Results

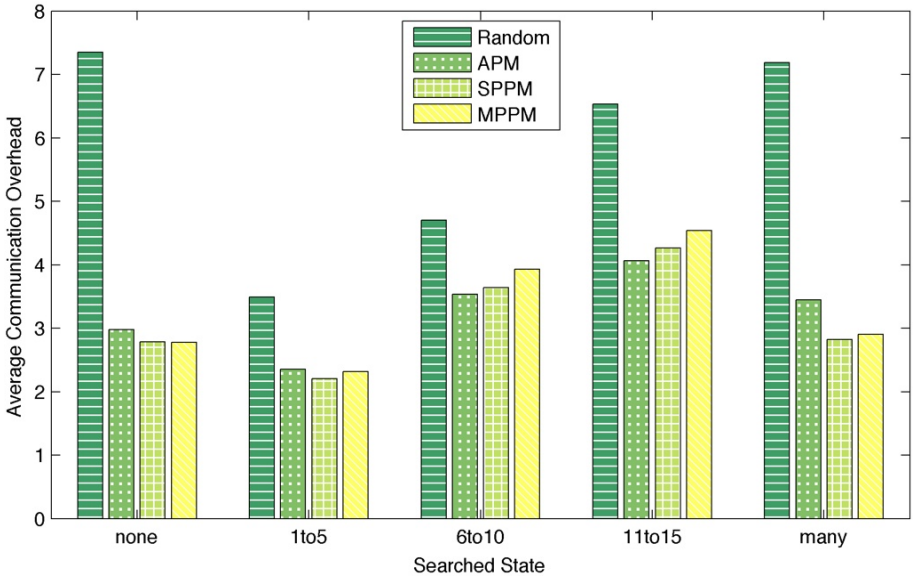


Figure 2.8: Average communication overhead without adjustment process.

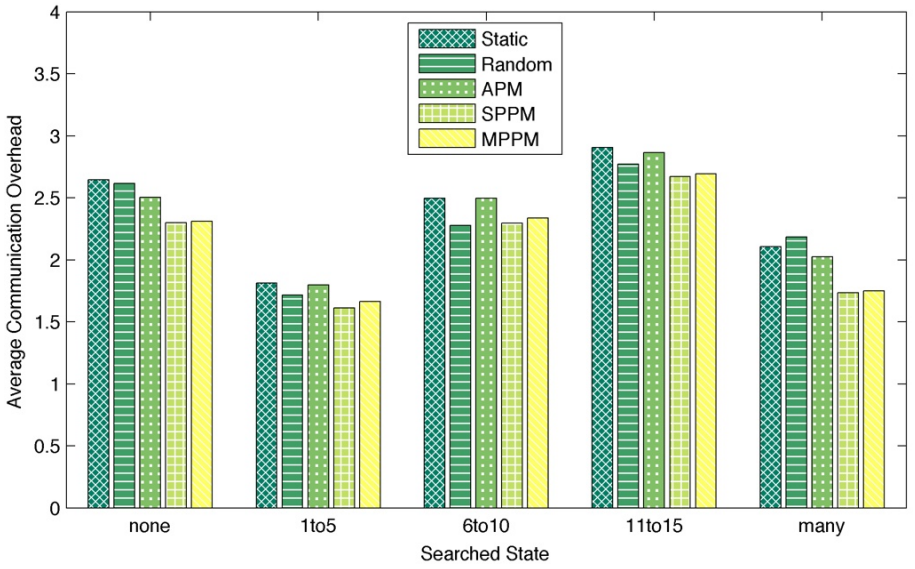


Figure 2.9: Average communication overhead with adjustment process.

The average communication overhead determined in the simulation runs is depicted in Figs. 2.8 and 2.9. Note that the state "unknown" is omitted from the discussion, since it always results in an optimal overhead of 1 for all considered prediction models. This is because either all sensors sense the state "unknown" or none of them do.

For both graphs, we see that the average communication overhead for the individual states varies, and that across all considered prediction models. One factor that affects this variance is the average number of

sensors which read the searched state. A larger fraction of matching sensors usually results in fewer sensors to be contacted in order to find the required number of results. As we can see from Fig. 2.6, on average the state "1to5" is read by the sensors most frequently. It is also the state with the smallest communication overhead. The correlation between the average communication overhead and the average number of matching sensors can be seen best when comparing the data from Fig. 2.6 with the results of the random prediction model in Fig. 2.8. However, the "predictability" of a certain state naturally influences the communication overhead, when considering an actual prediction model. For example, although the state "none" has a low average number of matching sensors, its communication overhead is rather low, compared to the other states, when not considering the adjustment process. This can be attributed to its periodic nature, which can be identified in Fig. 2.6.

2.5.3.1 Without Adjustment Process

When not taking the adjustment process into account, we see that for each state, there is a considerable difference in the overhead caused by the random prediction model and the other prediction models. This is as expected and shows the improvement of sensor ranking over a naive approach, which would contact sensors in arbitrary order until enough results are found. Compared to the other prediction models, the aggregated prediction model produces good results despite the fact that it does not take the factor time into account. This can be explained by two reasons: First, as can be seen from Fig. 2.7, there is a significant number of sensors which read the searched state 40% - 50% of the time, for example, considering the state "1to5". In this case, this should result in an expected average communication overhead of 2 - 2.5, which is confirmed by our simulations. The second reason are irregularities in the simulation data: although one can identify periodic patterns in the simulation data, these are often disturbed with outliers, i.e., the patterns are imperfect. This might partly be attributed to the simulation data, which is inherently ordered and was mapped to unordered states. A small change in the underlying raw sensor data (e.g., the number of free bicycles changes from 10 to 11) may provoke a change of the deduced state. Prediction models which rely on periodicities in the data are susceptible to these imperfections.

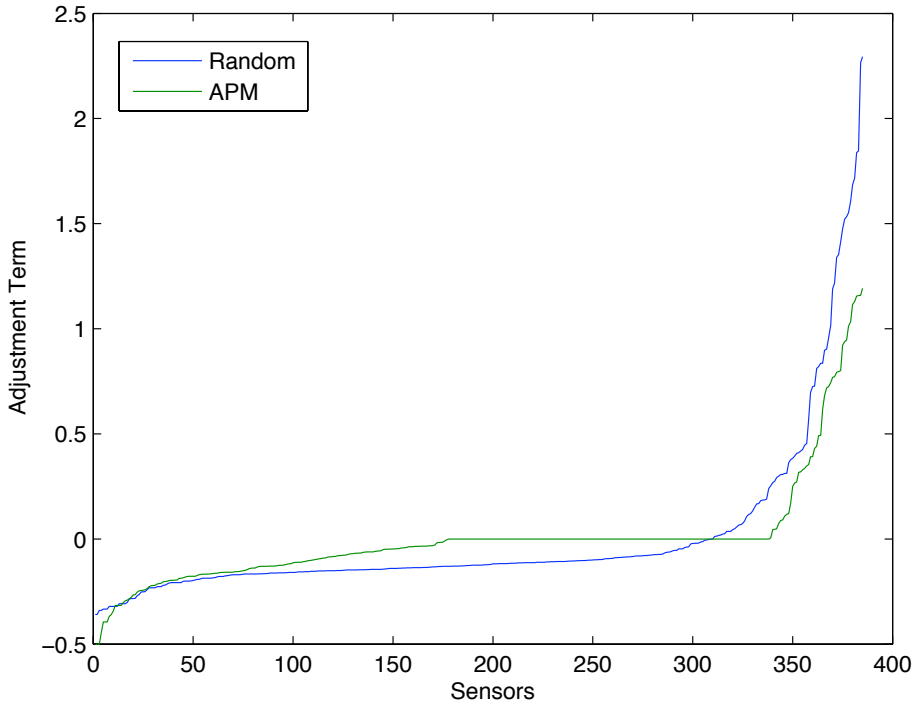


Figure 2.10: Distribution of adjustment terms at the end of one simulated week, for state "1to5".

2.5.3.2 With Adjustment Process

When looking at the simulation that utilized the adjustment process, we see that all results benefit from this approach (Fig. 2.9). While the largest average communication overhead without considering the adjustment process is about 7, it degrades to about 3 when utilizing the adjustment process. This can be explained by the introduced feedback loop, that decreases the prediction results of sensors ranked too high and increases the prediction results of sensors ranked too low. We additionally included a "prediction model" which produces an arbitrary but constant ranking of the sensors, in order to be able to better assess the effects of the adjustment process.

Comparing the different prediction models reveals that the average prediction overhead of the constant and random prediction model is only slightly worse than that of the other prediction modes. This may appear counter-intuitive – prediction models which do not predict meaningful values but result in good rankings. Recall that the adjustment process works best if executed frequently. Since we execute it for every query in our simulation, we update it every 15 minutes per state, and reset it upon the re-creation of the prediction models. For the static prediction model, this results in a behavior that can be

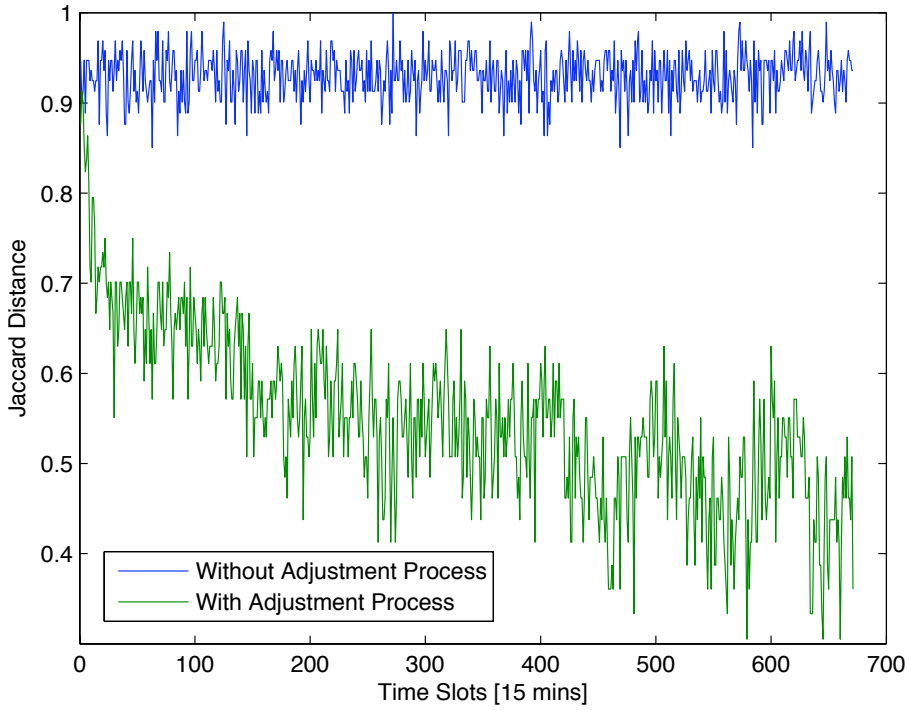


Figure 2.11: Stability of the 50 topmost ranked sensors over one simulated week, using the random prediction model and considering the state "1to5".

compared to caching – sensor predictions are adjusted to reflect the recent state. If the states sensed by the sensors are not changing too frequently, the adjustment process is able to model a ranking and therefore compensate the lack of an actual prediction model. However, this does not explain why the random prediction model with adjustment process performs even better than a static ordering of the sensors. The analysis of the simulation results revealed that with the adjustment process, there are sensors which are not considered at all, when using a static ranking. However, when using a ranking that varies over time (for example, like the random prediction model does) then all sensors will be considered by the adjustment process, which increases the probability that sensors, which continuously read the searched state over a longer period of time are found and adjusted "upwards". Figure 2.10 shows the distribution of the adjustment terms of all sensors at the end of a simulated week for the random and aggregated prediction model. An adjustment term of 0 is a strong indication that this sensor was never contacted. When looking at the distribution of the adjustment terms for the random prediction model, we see that the majority of the sensors was assigned a small negative value, while a minority was assigned a relatively large, positive adjustment term. Since an adjustment term > 1 can outperform any predicted probability, the actual

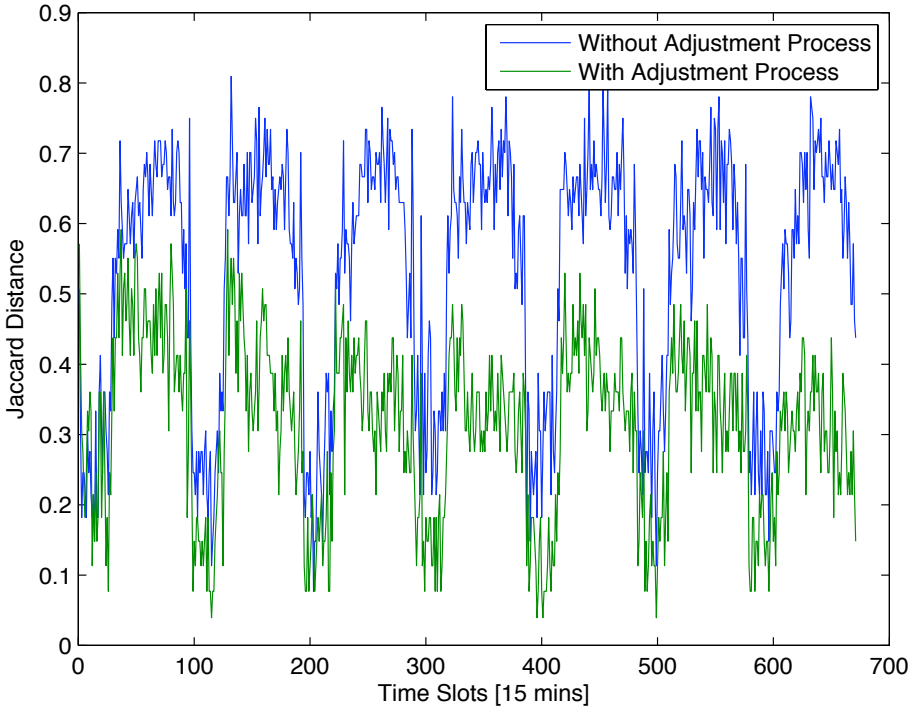


Figure 2.12: Stability of the 50 topmost ranked sensors over one simulated week, using the MPPM and considering the state "1to5".

result of the prediction model is marginalized for such values. The adjustment process generally seems to increase the stability of the top n sensors of a ranking, which should result in a lower ranking error, provided that the sensor states do not change too rapidly. To illustrate the effect of the adjustment process on the stability of the sensor ranking, we calculated the *Jaccard Distances* of adjacent pairs of the 50 topmost ranked sensors over the course of one simulated week.

The *Jaccard Distance* J is used to give a similarity measure of two sets A and B. It is defined as

$$J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|} \tag{2.17}$$

and provides results from 0 (both sets are identical) to 1 (no common elements at all).

The results for the random prediction model are shown in Fig. 2.11 and for the multi-period prediction model in Fig. 2.12. Both graphs clearly indicate that the adjustment process has a significant impact on the stability of the sensor rankings, i.e., it decreases the number of changes within the 50 sensors which are ranked topmost. We also see that the stability increases over time, confirming our assumption that this process works best if executed frequently.

2.5.3.3 Discussion

Based on our simulation results, we identify two major findings: First, the bigger effort introduced by more sophisticated prediction models does not necessarily lead to better sensor rankings. Second, the adjustment process does not only significantly improve the sensor ranking, but also seriously alleviates the influence of the prediction models on the sensor ranking. The best results are provided by a combination of the single-period prediction model with the adjustment process. However, we want to point out that these findings are only applicable to the considered simulation data and should not be generalized.

2.6 A Prototypical Real-Time Search Engine for the Web of Things

We will now demonstrate how our approach of searching in dynamic, distributed data in real time can be applied to the existing Web infrastructure. To this end, we developed a prototypical real-time search engine for the physical world called *Dyser* that is based on a Web of Things.

We need to design basic abstractions for the Web of Things to represent real-world entities and their states. On top of these, we need to define the system architecture for the search engine. One key goal herein is to retain the *open* and *loosely coupled* architecture of the current Web such that everybody can introduce one's own search engine for things – rather than producing a closed system under exclusive control of one party, thus hampering scalability and sharing.

In the Web of Things, entities of the physical world should therefore be represented as Web resources, accessed using HTTP, and should provide (among others) an HTML representation. This allows a seamless integration into the existing Web infrastructure, making it possible to utilize existing applications and services with Web-enabled things [85].

In our model, there are two key elements: *sensors* and *entities*. Each sensor and each entity receives a virtual counterpart, a Web resource, identified by a URL and accessible using HTTP. For all of these Web resources, there is always an HTML representation, which we call the *sensor page* and the *entity page* respectively. In addition to unstructured text they also contain structured information, for example, the type of sensor or its possible readings.

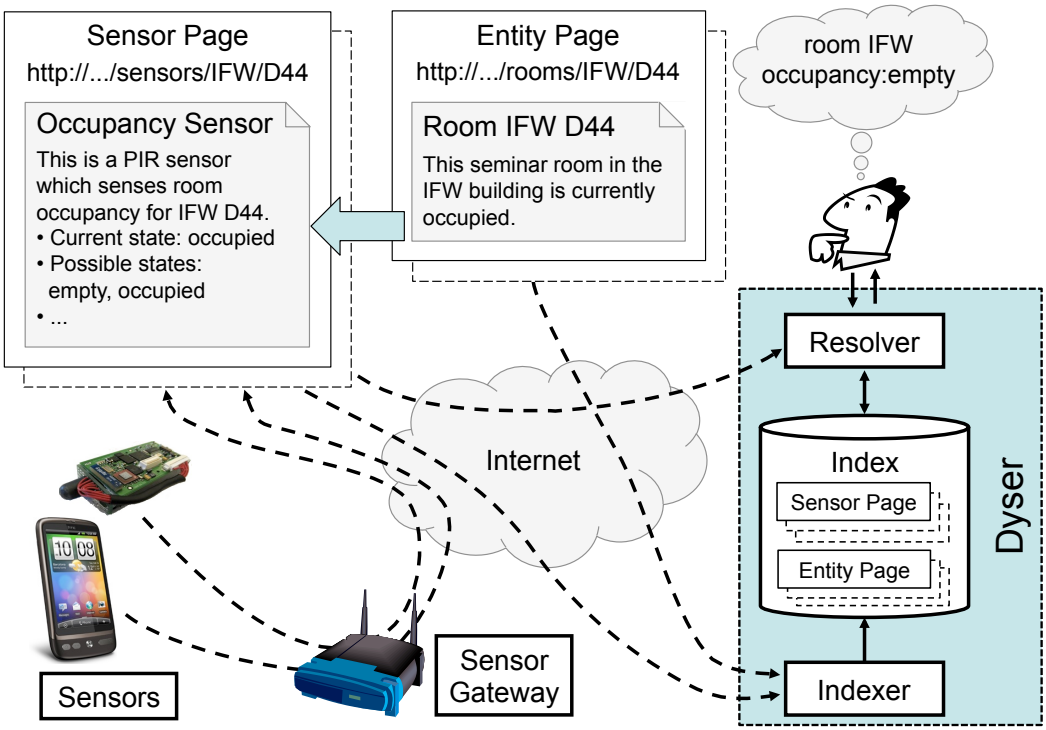


Figure 2.13: Architecture of Dyser (based on [102]).

Our search engine for the Web of Things follows the basic approach of existing Web search engines: it builds up an index of all relevant pages and offers a simple search language to find indexed entities. Note that our pull-based approach does not require sensor or entity publishers to register with Dyser, as sensor and entity pages are found automatically by Web crawlers, which follow hyperlinks.

2.6.1 Design

An overview of the system architecture is depicted in Fig. 2.13. There, *sensors* are connected to *sensor gateways* which are in charge of creating prediction models and publishing sensor information on the Web. Note that sensor gateways may also be implemented as processes running directly on the sensors, (e.g., on a smartphone) if resources permit.

As an example for a real-world entity, we consider a meeting room, represented by an HTML page. This page does not only contain static information like a textual description, but also dynamic information about its real-world state. This information is gathered from an associated sensor which detects whether the room is currently occupied. This sensor is also represented by an HTML page, which contains besides unstructured text also structured meta data about the sensor, such as

its prediction model. In order to associate the occupancy sensor with the meeting room, a simple hyperlink is used to refer to the Web page of the sensor. The state perceived by the sensor is included within the Web page of the entity, and can thus be viewed in a Web browser.

These embedded sensor data can be identified by the indexer, which periodically crawls the Web in order to build an index of relevant pages. Essential information about entities and sensors, including their prediction models, are then stored in the index. When a user issues a search request (in this case "`room ifw occupancy:empty`"), the *resolver* is in charge of handling its execution. For this, it will first reduce the result set to entities which match the static part of the search term ("`room ifw`") and feature the requested sensor type(s) ("`occupancy`"), by querying the index. In a second step, the dynamic part of the search request is handled by executing the prediction models of the specified sensor types for the specified sensor states ("`empty`") and the current time. The probabilities determined by the prediction models of the sensors are then combined to an overall probability for each entity. Beginning with the entity which has the highest probability, the sensors of each entity are then contacted by the resolver to gather their actual state, in descending order of their overall probabilities. As soon as enough hits are found, this process is stopped and the resolver returns its results to the user.

To sum up, prediction models are periodically *created* by a sensor gateway, based on a set of recent sensor states, periodically *indexed* by Dyser, at a rate of days to weeks, and *evaluated* by Dyser during the resolution of a search request.

2.6.1.1 Presentation on the Web

As all Web resources, sensors and entities need to be identified by a URL and accessed using HTTP. While there may be multiple suitable data formats, we focus on HTML, as it can be directly viewed in Web browsers, is indexed by existing search engines, can be hyperlinked with other HTML documents, and is well-known to today's Web users. We denote an HTML page that represents a sensor as a *sensor page*, and an HTML page that represents a real-world entity as an *entity page*.

In order to be able to display sensor-specific information to the user and at the same time provide this data accordingly structured for the indexer, we follow the concept of *microformats* [120]. By "abusing" certain portions of HTML markup, microformats provide the possibility

to semantically tag information included in HTML pages, while still giving the authors complete freedom on how that data is displayed. Note that there are similar concepts such as Microdata [121] and RDFa [122], however, we stick with microformats for their simplicity.

2.6.1.2 Search Language

As we expect that the search for entities of the physical world, based on their dynamic state, will be as common as the search for Web pages or images is today, it is important that the usability of Dyser is comparable to that of today's popular search engines. In particular, the search language used to construct search requests should be usable by the average search engine user without great learning efforts. We believe that query languages like SQL [110] or SPARQL [123] do not fulfil this requirement and are also oversized for our needs. Hence, we aim to gently extend the keyword-based search language utilized by the majority of today's Web search engines. There, one can already include structured data in a search request by using attributes, which are (name, value) pairs denoted as "`name:value`". In order to be able to search for sensor data, we introduce additional attributes to the search language which allow the specification of sensors and their current readings. For example, "`people:few loudness:quiet`" constitutes a search term with two dynamic attributes, *people* and *loudness*. The set of available attributes is defined by the sensor types indexed by the search engine.

2.6.1.3 Sensor Gateways

A *sensor gateway* connects sensors to the World Wide Web and makes their captured states and additional meta-information available on the Web via HTTP. Sensor gateways are also in charge of creating and publishing the prediction models for the sensors they administrate. They may furthermore offer additional functionality, like providing access to archived sensor states. While sensor gateways may run on the sensing devices itself, there may be technical or administrative limitations which require the use of dedicated gateways.

In most cases, sensors do not output a high-level state directly, but capture low-level data from which the high-level state of an entity has to be inferred, which can be performed by the sensor gateway. For example, the GSN middleware [124] may be used for that purpose as it offers a homogeneous interface to a large variety of sensors and can

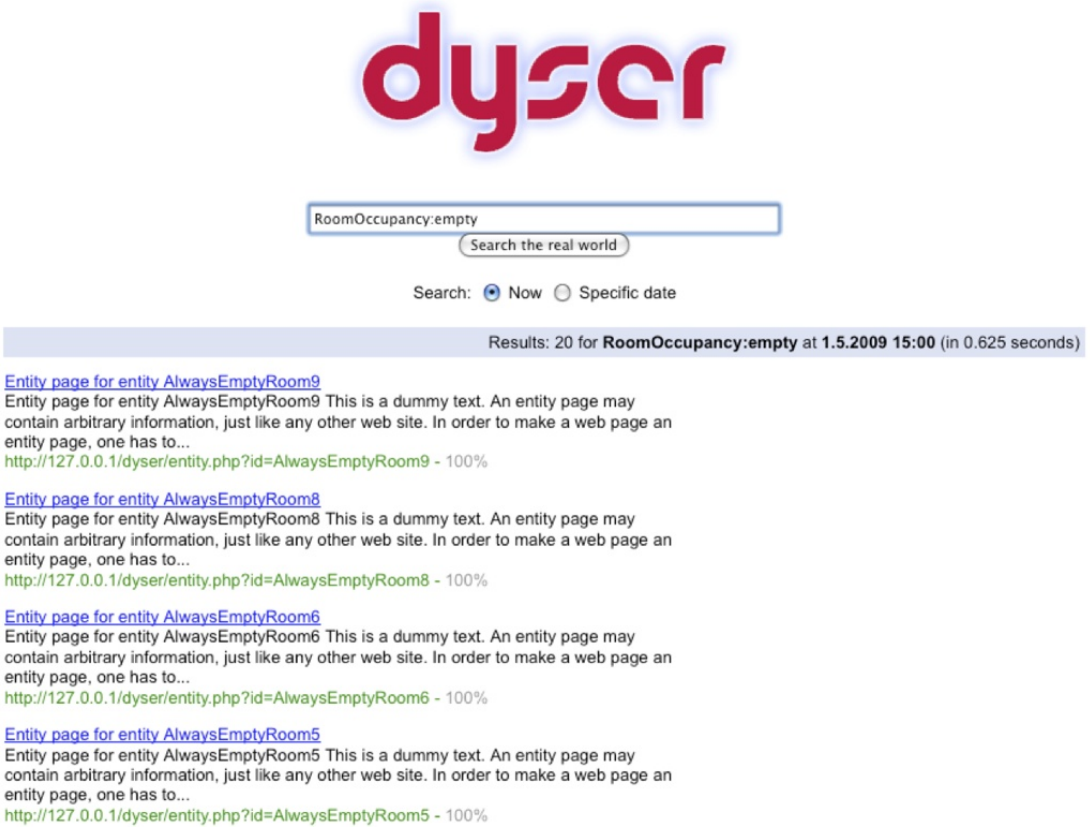


Figure 2.14: Exemplary screenshot of the search results page of Dyser[10].

also perform complex data-stream processing.

2.6.2 Implementation

The implementation of our prototype of Dyser is based on Java and PHP, using SOAP [63] Web services for communication between components.

2.6.2.1 Sensor and Entity Pages

At the time of our evaluation, there was no microformat for sensor information. We therefore designed a simple microformat for this purpose, which is depicted in List. 2.1. Semantic tagging is achieved by utilizing dedicated labels for Cascading Style Sheet (CSS) classes within HTML element tags, e.g., `class="sensor"`. Note that although the depicted example solely uses `<div>` elements, sensor information can be specified using any HTML element, given that it supports the `class` attribute.

In order to create a sensor page, one has to define the following structure: an enclosing HTML element is required whose class at-

[10]Dyser logo courtesy of Matthias Kovatsch.

```
<div class="sensor">
   <div class="id">OccupancySensor7</div>
   <div class="location">IFW D44</div>
   <div class="type">Occupancy</div>
   <div class="currentState">occupied</div>
   <div class="possibleStates">
      <div class="state">empty</div>
      <div class="state">occupied</div>
   </div>
   <div class="predictionModelType">
      SinglePeriodPredictionModel
   </div>
   <div class="predictionModel">
      <!-- JSON serialization, omitted for
           readability reasons -->
   </div>
</div>
```

Listing 2.1: HTML source of prototypical sensor microformat.

```
<a href="http://example.org/sensors/OccupancySensor7"
   rel="http://dyser.org/sensor">occupied</a>
```

Listing 2.2: HTML source of an entity page including a sensor. The hyperlink text *occupied* is not evaluated by the search engine.

tribute is set to `"sensor"`. Inside this element, information regarding the sensor is defined using further HTML elements, whose enclosed text defines the value for the aspect specified by the CSS class names: `id` is the (local) identifier for the sensor at its sensor gateway, `location` provides information about the location of the sensor, `type` identifies the type of the sensor, and `currentState` lists the state the sensor currently detects. `possibleStates` includes a list of `states`, which together specify the list of states this sensor can perceive. In `predictionModelType`, the utilized type of the prediction model is specified, while in `predictionModel`, a serialized version of the prediction model is stored, using the JSON format.

To adapt the visual appearance of a sensor page, one can select the appropriate HTML elements and also define the according CSS classes. As CSS allow not only to change font style and color, but also to hide complete elements or to prepend or append specified text, the embedding of sensor information does not need to have an influence on the depiction of the according sensor page. Fig. 2.15 shows the excerpt of a sensor page, where the microformat of the sensor has been formatted using CSS.

In order to create an entity page, one has to include at least one

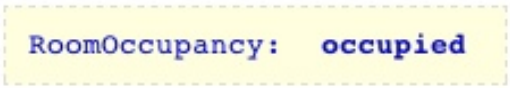


Figure 2.15: Example of the visual appearance of List. 2.1, formatted using CSS.

hyperlink to a sensor page, which has to follow a particular syntax: The attribute *rel* needs to be set to the specific URL

`http://dyser.org/sensor`

in order to denote that the corresponding hyperlink is a semantic link between an entity and a sensor page. Crawlers parsing HTML pages and following the embedded links utilize this information to detect entity pages and their associated sensors. An example of such a hyperlink is depicted in List. 2.2.

Note that in our current approach, sensor types and their states are just text labels, which can be specified at will by sensor publishers. This simplistic approach is intended to provide an open and flexible approach for publishing sensor data. In order to be able to provide well-defined and global semantics for sensor types, one could enhance the concept by outsourcing the definition of a sensor type, including its possible states and further specifications to a separate document. A sensor page would then use a hyperlink to the specification of the according sensor type.

2.6.2.2 Sensor Gateway

The sensor gateway was implemented in Java and features a SOAP Web service which provides access to information regarding the administrated sensors. To facilitate testing of our prototype, the sensor gateway automatically generates both a sensor and an entity page for each sensor it is in charge of and automatically publishes them using a REST interface. The current state of a sensor is modeled as a separate resource below the URL of the sensor: For example, the current state of the sensor

`http://example.org/sensors/occupancy42`

could be accessed at

`http://example.org/sensors/occupancy42/currentstate`.

Note that the suffix to the base URL of the sensor matches the name of the respective subsection of the microformat on purpose. This greatly reduces the overhead of resolving the current reading of a sensor, as only the current state is transmitted instead of the complete sensor page.

Our sensor gateway does not only support physical sensors, but also allows the creation of virtual sensors, based on recorded log files or methods generating synthetic sensor data, for example. If not done by the sensor itself, the sensor gateway will infer a state based on recent readings of a sensor. For each sensor, a history of its perceived states is stored and utilized to create a prediction model.

2.6.2.3 Prediction Models

Besides the aggregated prediction model, the single-period prediction model, and the multi-period prediction model introduced in Sect. 2.3.5, we also implemented the random prediction model from Sect. 2.5, as a baseline for evaluation. It is used to simulate the lookup of current sensor readings in random order.

One important aspect is an efficient representation of the models with respect to memory footprint, as the models need to be transmitted between the sensor gateway and the search engine and stored in the index. For the aggregated prediction model, we only need to transmit one probability value for every possible output state of a sensor. For the other models, we transmit the discretized output of the model for every possible state for a certain forecasting horizon.

2.6.2.4 Search Engine

Like the sensor gateway, the search engine was implemented in Java and features a simple SOAP Web service to pose search requests. This Web service is wrapped by a PHP script, which provides an HTML front end for entering search requests and displaying their results. Since we cannot expect that users are aware of all possible sensor types or of all possible states of a sensor, we provide an auto-suggest mechanism which helps the user to complete the search term by suggesting possible matches. Besides the front end, the search engine consists of three main components:

Indexer We implemented two indexers in our prototype: The first one is using a third-party Web search engine like Google in order to find all entity pages. For this, all entity pages must contain a "magic" string of characters. By searching for this magic string with Google, all entity pages can be found. However, it may take several days or even weeks until Google's Web crawler visits a page and includes it in Google's index, which is impractical for experimental purposes. For this reason, we include an alternative indexer which contacts sensor gateways directly, using the provided Web services, in order to obtain the URLs of all sensor and entity pages. The pages found by either of those methods are then downloaded, parsed and the contents are stored in the index.

Index In our prototype, the index is implemented as a relational database, which is accessed using the JDBC interface, thus allowing for a large variety of different database implementations. We are currently using the MYSQL database with the InnoDB engine as storage backend. In order to speed up the evaluation of the prediction models at query time, we *materialize* the outputs of all indexed prediction models in the database for a given forecasting horizon. That is, the database does not contain the prediction models, but the discretized output of the prediction models (i.e., probability values) for a certain forecasting horizon. Finding the entity with the highest probability of matching sensor outputs is thus realized as an efficient database lookup operation.

Resolver The resolution of a search request is implemented as a chain of filters over sets of entity pages as follows:

i) The given search term is parsed and separated into a static part (i.e., static keywords referring to the entity page) and a dynamic part (i.e., referring to the current output of sensors associated with an entity page).

ii) A third-party search engine like Google is used to find entity pages matching the static part of the search request. For this, a magic character string contained in every entity page is appended to the static part of the search request. As a side effect, we also obtain a *relevance* ranking for each entity page from Google. As for the indexer, we also implement an alternative approach which does not rely on Google, but queries the index of Dyser directly.

iii) Each entity page found in the previous step is checked whether it includes sensors of all types requested in the dynamic part of the search request. Entities which do not contain all requested sensor types are removed from the result set. For the remaining entities, the overall probability that they match the dynamic part of the search is then determined by querying the index holding the materialized prediction models.

iv) The entity pages produced by the previous step are then considered with decreasing probability of matching. For each entity, the sensors associated with that entity page are contacted and their current values retrieved. If the current state of a sensor does not match the state requested in the search term, the entity is removed from the list of results. To speed up processing, multiple sensors are contacted in parallel, using a pool of threads. This process stops when enough matching entities have been found.

v) Finally, all matching entities are sorted according to their relevance ranking obtained during step ii) and presented to the user.

To avoid the overhead of generating large intermediate lists of entity pages, the above steps are performed in a pipelined fashion. As soon as a certain number of entity pages are produced by one of the above steps, they are passed on to the next step. Only if not enough matching entity pages are generated in the last step, the previous steps are triggered recursively to generate more input.

2.6.3 Evaluation

We evaluated the performance of our prototypical search engine in terms of the communication overhead and latency. The evaluation is based on the data set introduced in Sect. 2.4, that is replayed by the sensor gateway instead of using actual sensors. This data set represents an exemplary case for our system as the data is heavily affected by the behavioral patterns of people in everyday life.

2.6.3.1 Setup

The parameters of the simulation setup are identical to that of the Matlab-based simulation in Sect. 2.5. We utilized the search engine implementation with virtual sensors that replay the Bicing data set

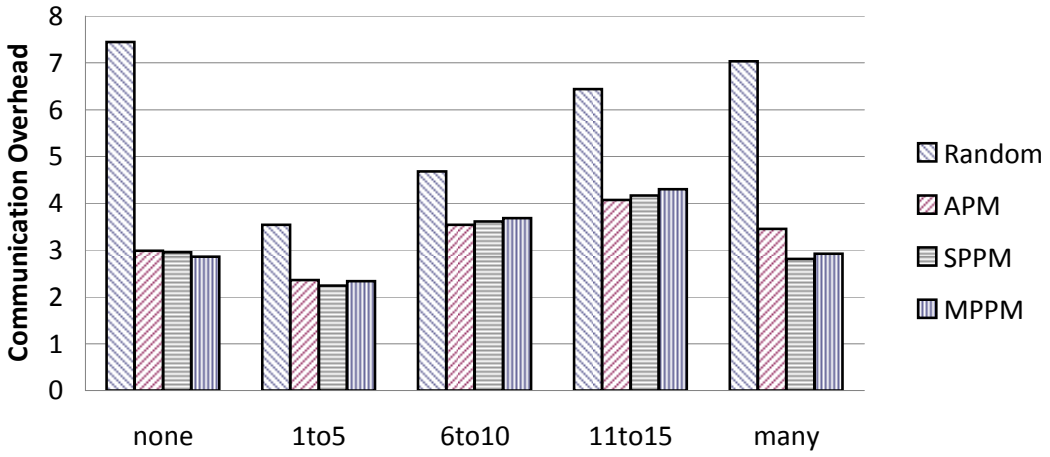


Figure 2.16: Communication overhead when searching for the different states.

described above. In order to perform the simulation, we used a PHP script which accessed both the sensor gateway and the search engine using the provided SOAP web services. All processes were running on the same local machine, which features two dual-core Intel Xeon Core 2 CPUs running at 2,66 GHz. The maximum number of threads for the resolver was set to 8.

Starting on March 1st 2009, all prediction models are created and indexed by the search engine. Then, a query is posed for each possible state and the outcomes are recorded for later analysis. After all states have been queried at the given point of virtual time (i.e., simulation time), the sensor gateway and search engine are instructed to advance their virtual time by one time slot (i.e., 15 minutes) and search requests for all possible states are posed again. If the maximum forecasting horizon of one week is reached, all prediction models are recreated at the sensor gateway and re-indexed by the search engine. This process is continued for 3 months, until the end of the simulation is reached on the beginning of June 1st 2009.

We consider two metrics. First, the *communication overhead* as defined in Eq. 2.16, which is the number of contacted sensors M_k divided by the number of requested results k for a given query. Recall that a communication overhead of 1 is an optimal result, indicating that no non-matching sensors were contacted. Second, we consider the *latency* from issuing a query until the requested number of matches has been found and returned to the user.

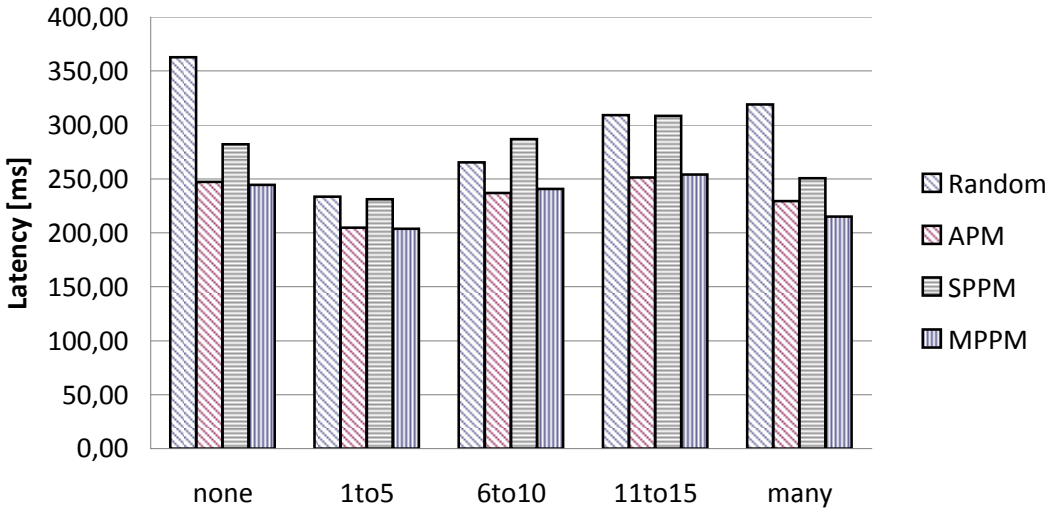


Figure 2.17: Latency when searching for the different states.

2.6.3.2 Results

Figure Fig. 2.16 shows the average communication overhead when searching for the different states. As expected, the results for the communication overhead are identical to that of the evaluation in Matlab, since we used the same simulation parameters.

Figure 2.17 depicts the average latency when searching for the different states, showing a similar trend as the communication overhead. However, the differences between the different models and the improvement over *Random* are smaller, as the latency does also include overhead for database lookups which are shared by all prediction models and which result in a constant latency baseline for all models. This baseline makes up about half of the total latency (about 160 ms for *none* and about 120 ms for all other states), showing potential for further improvements in our prototype. The remainder of the latency is predominantly caused by remote sensor readouts. Note that the latter heavily depends on the parallelism of remote sensor readout operations. In our setup, where the sensor gateway and the search engine are executing on a single computer, this parallelism is limited by the number of CPU cores and hence the latency figures can be considered a worst case that is unlikely to occur in a real deployment where sensor gateways are distributed over many computers, resulting in higher parallelism despite longer average latency for a single remote access to a sensor due to higher round-trip times in the global Internet. A notable artefact in the results are the relatively high latencies for the single-period prediction model. Based on the communication overhead

figures, we would expect similar values for all three prediction models. Analysis indicates that this is an implementation-specific problem caused by memory management issues in the Java virtual machine.

2.6.3.3 Leveraging an existing Web Search Engine

The concept of using an existing Web search engine to find and rank entity pages which have been tagged with a magic string worked to a certain extent: A summary page, which linked to all utilized entity pages was created automatically. After including a hyperlink to this page on some home pages of members of our group, we could see from the log files of our WWW server that Google's crawler periodically visited all listed entity pages. However, not all of the entity pages were actually returned when searching for the magic string with Google. As the entity pages have been generated automatically, they are structurally identical, which might cause Google to remove entity pages from its index. Other popular Web search engines revealed similar behavior when using their Web front ends to search for the magic string of the entity pages. Despite this fundamental restriction, using Google to handle the static part of a Dyser search request worked as expected. Google's SOAP Search API[11] was limited to return only the first 1.000 hits for a search request.

2.6.4 Discussion

In this section, we demonstrated how our approach of searching the physical world can be applied to the Web. In summary, sensors and entities are represented by simple Web pages, that can be crawled and indexed by search engines, just like the traditional Web search works today. The association of an entity with a sensor is achieved by including a simple hyperlink. Structured data is defined using microformats, a concept which enhances HTML pages with semantics. Including the prediction models used by sensor ranking on the sensor pages enables the indexing of such models by our search engine. Resolving the current state of a sensor by the search engine is a simple request over HTTP.

As expected, the simulation results of Dyser show that a higher communication overhead correlates with a larger computation time for a search result. However, the effect on computation time is smaller

[11] This API was deprecated soon after our evaluations, and is now out of service. However, there are alternatives available.

than the communication overhead indicates, as multiple sensors may be queried in parallel.

Note that the process of resolving current sensor readings could be outsourced to the clients, which would relief the search engine from a significant load and thus improve scalability. For this, the search engine would only need to provide the ranked list of candidate results to the client. The client could then contact the candidate results until enough actual results are found, which could easily be implemented using HTML and JavaScript in a Web browser. Another possible advantage of this approach is that network traffic for processing a search request would mostly be generated by the clients. This could relive the network infrastructure as traffic would be more distributed. Also, so-called "denial of service attacks" would be aggravated, as a single search request by a client would not lead to potentially high load on the search engine.

Publishing prediction models publicly on the Web might affect privacy when used with certain sensor installations: these models reveal a "big picture" of the sensed phenomenon at once, which would otherwise have to be composed by an attacker himself, by periodically collecting sensor readings over a longer period of time.

2.7 Related Work

Related work can be categorized in two fields: search engines specifically designed to search the physical world and real-time Web search engines that arose in the recent years. Additionally, there are some other related approaches that we discuss.

2.7.1 Search Engines for the Physical World

MAX is a search engine for the physical world, which allows users to locate tagged objects [125, 126]. For this, everyday objects are expected to be augmented with RFID tags containing a textual description of the object in question. Users can search for objects using keywords and are returned a ranked list of matching objects, including their locations. Locations are specified relative to well-known or easily identifiable landmarks and mimic the description that humans would provide in order to explain the location of an item. An example of a location would be "Peter's bedroom → wooden bookshelf → 2nd shelf" [125].

The architecture is composed of several layers: At the lowest layer are the tagged objects. These can be detected by and communicate with *sub-stations*. These stations are themselves attached to physical objects, mostly those which rarely change their positions (e.g., a bookshelf). Sub-stations also store a description of the object they are affixed to. At the next hierarchy level, there are *base-stations*, which represent immovable entities such as a room. These stations also store a description of their locality (e.g. "Ben's office"). Base-stations are in turn connected to a backbone network, which connects to the MAX server.

When the user enters a query, it is send from the MAX server to all or a subset of base stations, which in turn broadcast the query to their substations and in turn to adjacent tags. The tags answer the substations if their description matches one or more keywords of the query. The sub-stations return matching tags, including their RSSI values, to the base-stations and in turn to the server, which returns the result to the user. Interestingly, dynamic object properties are not specifically addressed by the authors, although their system would in fact work in the same way with dynamic descriptions, as the evaluation of the query is performed at the tags. The authors also provide an extensive evaluation of different query protocols, i.e., strategies for processing the query and show how security and privacy concerns can be addressed.

The authors briefly mention that in order to achieve scalability, the MAX server should distribute the query only to selected base stations instead of flooding the query to all base stations. They specifically mention the possibility to utilize a "Bloom filter for each base station that is updated periodically, and can efficiently rule out all base stations that have no possibility of having the item." Such a filter would be maintained by the MAX server [126]. Note that a Bloom filter does not return false negatives, i.e., results would still be complete as long as there is no mobility of the objects involved.

Microsearch [127] is search engine for embedded devices, which is tailored to the limited resources of such devices. Similar to desktop search engines like Mac OS X Spotlight[12] or Google Desktop[13], Microsearch indexes local data and answers queries with a ranked list of results. The use case presented in [127] is a small embedded device running

[12] http://support.apple.com/kb/HT2531
[13] http://desktop.google.com (discontinued)

Microsearch which is attached to a binder holding several documents. The device itself does not gather information by itself using sensors, instead it requires the user to initially upload content and metadata. Such metadata consists of a list of tuples *(term, value)*, where *term* is a searchable term and *value* specifies its weight. Note that for text-based content, this data may be extracted automatically. The user can then search the contents of the binder by querying the attached node with a list of keywords and the number of results k he is interested in. Microsearch will then return a ranked list of the *top* k results, i.e., matching documents in the binder. The work specifically addresses the requirements of an embedded search engine not only with respect to the limited resources of embedded devices but also with respect to security issues. The author suggest that Microsearch should be combined with physical search engines like MAX or Snoogle (see below).

Snoogle [128] is a search engine for the physical world by the authors of Microsearch. It allows users to search for and locate objects using keywords. The architecture consists of three layers. On the lowest layer, there are *object sensors*, which are sensor nodes attached to physical objects. An object sensor stores metadata and information related to the object it is attached to, e.g., a sensor attached to a CD might store the information from its booklet. Metadata consists of a list of keywords which have been assigned a weight. As in Microsearch, the authors assume that object sensors will need to be manually initialized, by having their data uploaded by a user.

On the next layer, *index points (IPs)* detect object sensors in their proximity, retrieve their metadata and store it locally in an inverted index. Likewise, IPs also update their data when object sensors change their metadata or remove the metadata of a object sensor when it leaves the proximity of the IP. Index points are considered to be immobile and therefore associated with a specific location. The data acquired by Index Points is then aggregated and forwarded over a mesh network generated by the IPs to a *Key Index Point (KeyIP)*, which resides on the third layer. A KeyIP has an aggregated, global view of all object sensors and its metadata.

Snoogle supports local and distributed queries, which are both based on keywords. A local query restricts the search space to the objects in the transmission range of a given IP, and is sent directly to this IP. A distributed query is a global search that involves the KeyIP:

The user's client first directs his query to the KeyIP, which answers which a ranked list of IPs that have objects in their vicinity that best match the query. Based on this results, the client can then perform a distributed search on the returned IPs to obtain the top k results. The authors specifically address privacy and security, data compression and optimization of data structures for flash memory.

Objects Calling Home (OCH) [129, 130] is a research project addressing the locating of physical objects. Instead of relying on a dedicated infrastructure used solely for locating searched objects, OCH is based on the existing infrastructure of cell phone networks and mobile phones. This offers several benefits: cell phone networks are deployed globally, mobile phones are common and usually carried along by their users (so sensing is performed at places relevant for search), and mobile phones could use built-in technology such as Bluetooth or Wi-Fi for detecting tagged objects. Additionally, information from the cell phone network or the phones itself can be used for localization.

The basic idea of OCH is to distribute a search query for locating a well-known item among certain phones of the cell phone network. Phones which receive the query then try to detect the given item for a limited time. If the item is found, its estimated location is then returned as an answer to the search request. In order to achieve scalability, heuristics are applied to distribute the query only to such phones, where there is a sufficient probability of a positive answer. Such heuristics typically mimic the strategy a human would pursue in order to find an item he has recently lost. Examples are searching at locations where the object was last seen, which the user recently visited, or where the user spends a large amount of time. In order to limit the costs of a query, it is limited both in time and in the number of messages sent for its dissemination.

SCPS is a "social-aware distributed cyber-physical human-centric search engine" [131], which has a scope similar to OCH. The work addresses the search for physical objects carried by people. The basic idea is to leverage Bayesian networks in order to predict the locations of people (and in turn, of searched objects). The authors argue that human mobility usually exhibits patterns, which can be predicted to a certain extend. However, external events such as the accidental meeting of a friend on the street or a change of the weather conditions may

break these routines. In order to cope with such non-routine situations, the authors try to model these in their Bayesian network, claiming that they also take into account the social relationship between persons in order to further refine the results. The Bayesian network presented in the paper is rather simple and predicts the location of a person as a discrete state, based on a time slot, and an exceptional event variable.

The architecture of SCPS consists of fixed base stations, and mobile nodes and sensors that both communicate with their nearest base station. The association of objects (i.e., elements that can be searched for) and mobile nodes (i.e., people) is done either manually or automatically by the sensors. The base stations form a distributed hash table (DHT), which is used to store the associations of objects to mobile nodes and mobile nodes to location predictions.

When the user starts a search for an object, the hashed object name is used to retrieve possible holder names, which are in turn used to retrieve possible locations. Based on these predictions, it is tried to contact the object holders and query if they actually carry the searched object.

Distributed Image Search [132] is a search engine for current or past images captured by camera sensor networks. For this, sensor nodes are equipped with a camera and deployed in the field. The user can then pose a query by specifying an image and the system returns sensors that captured similar images. Results are ranked according to their similarities with the query image and only the k most relevant matching sensors/images are returned to the user.

Searching for images in a camera sensor network poses a significant challenge with respect to the computation, communication and energy capabilities of such devices, since sensor nodes are usually resource-constrained devices. In order to save resources, the authors minimize the use of raw images. Images are first represented by 128 byte SIFT[14] vectors and then further abstracted by clustering these SIFT vectors into 4 byte *visterms*. Image matching can then be performed efficiently by comparing the visterms of two images. In general, much effort of the paper is devoted to optimize data structures that are stored in flash memory.

A simplified description of the systems is as follows: Sensor nodes equipped with cameras take pictures and store them in their local flash

[14]Scale-Invariant Feature Transform

memory. The nodes also create, among other data structures, an inverted index linking visterms to images. The sensor network, consisting of the camera nodes, is connected to a proxy that accepts user queries. Once the proxy receives a query, i.e., an image, it computes its visterms and distributes these to all sensor nodes. Each sensor node then processes the query, computing a ranked list of images whose visterms match those of the query. The sensor nodes then report back to the proxy, which will normalize the result lists, creating a globally ranked list of results. This final result list is then returned to the user, which may request thumbnails of full images of results. The system does not only support *ad-hoc queries* for archived images but also *continuous queries* for current images.

Uddin et al. address the search for distributed, dynamic data in the context of network management applications [133, 134]. They focus on searching dynamic properties of network devices, such as routers with a high CPU utilizations. The concept involves the use of distributed search nodes which collect data from network devices and provide a standardized interface for query processing. Queries are distributed to all search nodes, and results are aggregated among search nodes using a spanning tree protocol. The results are ranked according to their similarity to the query, their connectivity in an object graph, and freshness of data. The evaluation is limited to only nine search nodes, but the authors argue that their approach should scale to large numbers of search nodes.

Sensor Ranking Several approaches have been conducted based on our concept of sensor ranking. In [135], a new approach for prediction models to be used with our search engine is presented. There, prediction models based on Bayesian network are created, considering the correlations between pairs of sensors. This concept exploits the fact that co-located sensors often show correlated output. Truong et al. utilize sensor ranking with fuzzy sets for prediction models [136]. They consider content-based sensor search based on queries which specify a range window for sensor readings. This approach does not rely on the periodicity of sensor readings. Pfister et al. introduce a holistic approach for a search engine for a semantic Web of Things in [89]. It builds on some of the concepts of our work, such as sensor ranking and the search for entities with a given state. However, it extends further by leveraging

concepts from the Semantic Web such as RDF and SPARQL, in order to describe sensor semantics, define prediction models, and formulate search queries. Additionally, the system supports the semi-automatic creation of sensor descriptions, by detecting similar sensors (based on work of [137]). Readings from sensors nodes are retrieved through a HTTP gateway to CoAP over 6LoWPAN, or can be scraped live from Web pages. The authors outline a testbed which utilizes sensors located within an office building and at its parking lots. There, users can either perform sophisticated queries using SPARQL or resort to simple keyword-based search. Search results can be displayed on a map. More details on the modeling of prediction models in RDF can be found in [138].

Mayer et al. propose an infrastructure for the Web of Things that supports searching. It is based on a logical tree structure which reflects the locality of involved smart things. Search queries can be limited to a certain scope or number of results, and are resolved by propagating them along the logical structure. Content-based sensor search is not directly addressed, however, there is support for caching of (intermediary) results [139].

Gander, an approach for personalized search of ones immediate surroundings is introduced in [140]. The key concept is the local propagation of search queries by dedicated query routing protocols. This limitation of the search space enables spatiotemporal queries to be efficiently resolved.

IoT-SVKSearch is a real-time search engine for the Internet of Things [141]. It addresses the search for current as well as historic data, based on spatial, temporal, value-based, and keyword-based search conditions. Data from sensors is processed in a distributed index that can be updated in real time. The results of their simulative evaluation indicate short query processing times for content-based sensor search, despite considering large numbers of sensors.

Benoit et al. use approaches from the semantic Web in order to enable the search for real-world objects matching certain static properties [142]. An exemplary use case is the search for devices with a display capability. Content-based sensor search is not addressed.

2.7.2 Real-time Web Search Engines

During the time this thesis was developed, the concept of real-time search on the Web gained significant momentum, also for established search engines and social networks. This reason for this development can be attributed to the growing importance of social networks such as Facebook and Twitter. Users of these services can share their thoughts, interesting links and other pieces of information in real time with friends or even with the rest of the world. As opposed to the decentralized nature of the WWW, such social networks are run by a single authority which has a global real-time view on the activities of all of their users. This removes the hardest problem for searching real-time data on the Web, which is the inherent distribution of such data. Having a real-time data stream comprised of all user postings, the remaining problems are the indexing of the stream and the ranking of the results. The largest social networks currently offer such data streams to selected third parties [143, 144]. Of course, there are also other data sources on the Web which are relevant to real-time search, such as news sites and blogs.

Build around such real-time data, several new companies and products were created, addressing the need for up-to-date search results. However, in the fast-paced development of today's Web, several of these initiatives faced significant change or went out of service. In contrast, only few publications have been found in academic research. We summarize relevant approaches in the area of real-time Web search below.

2.7.2.1 Commercial Approaches

Twitter is a so-called *micro-blogging* service provider and, besides Facebook, probably the most well-known social network. Users can not only tell the world what they are currently doing but also engage in private or public conversations with other Twitter members. The length of a Twitter message (denoted as "Tweet") is limited to 140 characters [145]. Twitter did not feature a search function initially. In 2008, the company bought Summize, a search engine for Tweets and subsequently integrated it with Twitter's infrastructure as Twitter Search[15] [146]. Users can search in the vast number of Twitter messages using keywords and get the latest results in real time. In March 2011, the company claimed to index an average of about 190 million Tweets per

[15] http://search.twitter.com

day and serving 1.6 billion queries per day[16] [147]. Messages are pushed to the Twitter service, where they are archived. Twitter also provides a real-time feed of all public Twitter messages using the XMPP protocol, called the *firehose* [143]. Selected partners can build applications on top of this data stream. For example, Bing and several smaller search engines offer real-time search based on Twitter's firehose data.

Google, the most popular search engine on the Web according to the traffic analysis company Alexa [148], conducted multiple approaches to increase the "freshness" of its results. Google News, launched in September 2002, initially provided "a continuously updated index of news from 4,000 publications around the world." [149]. In contrast to Google's Web search results, whose index was at that time updated in months, Google News was updating its index continuously, thus returning also articles published recently.

In December 2009, Google launched its real-time search feature [150], which particularly addressed the social Web. In order to acquire real-time data, Google partnered with content providers in order to gain real-time access to public updates, using dedicated APIs. Google had agreements with several social networks such as Twitter and Facebook, for example [150, 151]. Real-time search results were not only provided by a dedicated real-time search engine[17], but also included in the search results of its "unified search". However, Google terminated its real-time search feature on July 3rd 2011, shortly after the agreement with Twitter on the use of its "firehose" expired [152]. Although the Twitter feed was just one of many feeds used for Google Realtime, it was considered to be the most important [153]. In 2010, Google moved to a new search index called "Caffeine" that features improved support for real-time updates [154].

A few days before the termination of Google Realtime, the company announced their own social network, called Google+[18] [155], which can be considered a rival to Facebook and Twitter. As of January 2012, Google offers real-time search results from Google+ within its unified search results.

[16] Note the ratio of Tweets per query.
[17] http://google.com/realtime (defunct)
[18] http://plus.google.com

Bing [19] is a search engine from Microsoft (formerly known as live search and MSN search). The authors of [109] mention that the index of the standard Web search engine was updated on a daily basis. Similar to Google, Bing also provides a search feature related to news, which returns up-to date results from major news sources[20]. Bing also featured a real-time search called Bing Social[21], which is based on Twitter's firehose and a dedicated API from Facebook [156]. It returns real-time search results from both of these social networks.

Technorati was a real-time search engine for blogs. In 2009, the site claimed that "Technorati.com indexes millions of blog posts in real time and surfaces them in seconds." [157]. However, the business focus of Technorati has shifted since then and the site is now focussing on advertising. Originally, users were able to provide a hint (signal) to the search engine when they updated their blogs, using a dedicated API call – a so-called RPC ping. However, the site noted that it was no longer using these hints, claiming that "more than 90% of the pings we received were spam and non-blogs" [158].

OneRiot [22] (formely me.dium[23]) was a real-time search engine that originally indexed Web pages visited by users which had a dedicated browser plug-in installed [159]. It was later extended to also include pages shared by users of social community sites such as Digg and Twitter [160, 161]. The focus was to return search results that are currently considered relevant or popular by users of these communities [160, 161]. The service was shut down after OneRiot was acquired by Walmart in September 2011 [162].

Others There exist several other real-time search engines for the Web, some of the are listed below. Topsy[24] is a real-time search engine by Topsy Labs[25] that indexes data from Twitter and Google+. Collecta[26] was a real-time search engine that indexed data from various sources, such as blogs, Twitter and Flickr. The search engine shut down in

[19] http://www.bing.com
[20] http://www.bing.com/?scope=news
[21] http://www.bing.com/social/ (defunct)
[22] http://www.oneriot.com
[23] http://me.dium.com
[24] http://topsy.com
[25] http://topsylabs.com
[26] http://collecta.com

January 2011 [163]. Further real-time search engines include 48ers[27], Icerocket[28], CrowdEye[29] (defunct) and Scoopler[30] (defunct).

2.7.2.2 Academic Approaches

The problem of delivering accurate results despite the dynamic of the Web was identified early. The research mostly focussed on on crawling strategies.

Cho and Garcia-Molina addressed this problem in [164]. They ran an experiment in order to gain insights on the dynamics of the WWW, and based on the data, the were able to model the changes of Web pages as a Poisson process. They also estimate the optimal revisit frequency of a page with respect to its change frequency, which is somewhat counter-intuitive: The authors divide pages into those that are updated at a low frequency and those that are updated at a high frequency. For the first group, the freshness benefits if a crawler visits a page more often as it changes more often. However, for pages that are updated at a high frequency, it is beneficial to the overall freshness of the index to visit them less often as they change more often. This can be explained by the limitation of bandwidth of a crawler, which should not be excessively spent on revisiting highly dynamic pages that may be outdated as soon as they are included in the index. The authors also outline an incremental crawler that aims to improve the freshness and quality of its local collection.

Edwards et al. suggested a Web crawler which does not rely on predetermined models, instead it dynamically adapts to the observed change rates of the Web pages [165].

Risvik and Michelsen studied the aspects of Web dynamics with respect to search engines [108]. At that time, both authors worked at FAST and provide interesting insights on the architecture of the company's Web search engine AlltheWeb[31]. To increase the freshness of the results, they consider not only an optimization of the general crawling strategy

[27] http://www.48ers.com
[28] http://www.icerocket.com
[29] http://www.crowdeye.com
[30] http://www.scoopler.com
[31] http://www.alltheweb.com (now part of Yahoo!)

but also utilizing different crawling strategies for different sites (e.g., for news sites). They also list different forms of cooperations with content providers in order to increase the freshness of the index: (1) Passive notification by having each site serve a central meta-file which may help to optimize the crawling, (2) active notification of the search engine whenever the site was updated, and (3) a combination of the first two approaches using proxy server. Interestingly, both (1) and (2) are common today. (1) has manifested in the SiteMaps de-facto standard [166] or in the form of ATOM [167] or RSS [168] feeds. (2) is the "pinging" of search engines using a Web service call in order to notify them about updates, mostly in the context of blogs. Finally, they also address the processing of updates at the search engine by suggesting several indices with different update frequencies.

Ikeji and Fotouhi proposed an "adaptive real-time Web search engine" in 1999 [113]. Instead of using an index, this search engine is crawling the Web for each search request. To submit a query, the search term, a list of Web pages that act as starting points for the crawler, and a time limit have to be specified in a query language that is similar to SQL. As soon as the search query is submitted by the user, the search engine starts to crawl the Web, beginning at the start points. Pages which need to be crawled are assigned a priority based on the contents of their URL and the priority of their parent page. This is an attempt to optimize the order in which extracted links are visited by the crawler. Note that this solution does not meet our definition of a real-time search engine (Sect. 2.1.4), as the computation of the results may require a considerable amount of time.

Watters follows a similar approach in his master thesis [114], which also restricts the search space to a small portion of the Web. As for [113], the time required to answer a search query can not be considered real time.

2.7.2.3 Discussion

An approach used in all of the outlined examples is the limitation of the search space: Twitter has strict limits not only on the allowed size of messages but also on the allowed publishing rate of messages. OneRiot focuses on a small subset of the Web, which is currently popular among

users of social networks. Technorati only considerd blogs, which are a small subset of the Web. A second approach is to utilize user-specified hints, which may affect the order and frequency in which sites are re-indexed. This is performed by OneRiot and was utilized in a simpler variant by Technorati. Finally, Twitter is using a centralized approach – all data is stored at Twitter's servers, thus the service has a real-time view of all published messages.

While the concept of Twitter and its search engine could also be utilized to publish sensor readings and to search for them in real time, we question that this concept, which requires a central instance aware of all current Twitter messages, will scale to dimensions of an expected Web of Things. When compared to human-generated messages, the volume of sensor-generated content is expected to become magnitudes higher, both in the number of sensors and in the frequency of updates. For the same reason, we doubt that the concepts utilized by OneRiot and Technorati are appropriate to enable a real-time search in the upcoming Web of Things.

2.7.3 Other

Global Sensor Networks (GSN) GSN [124, 169, 170] is a system for the Internet-based interconnection of heterogeneous sensors and sensor networks, supporting homogeneous data-stream query processing on the resulting global set of sensor data streams.

The key abstraction in GSN is a *virtual sensor* that represents either a physical sensor or a virtual entity that takes as input one or more data streams from other virtual sensors and processes them to produce a single output data stream. Virtual sensors can also be based on the output streams of existing virtual sensors. Each virtual sensor in GSN has a unique identifier and can be annotated with meta data (i.e., key-value pairs) to describe the sensor. Such meta data may be used, for example, to specify the location or the type of sensor.

Virtual sensors themselves are hosted in a so-called GSN container, which is essentially a process executing on a computer connected to the Internet. These GSN containers also form a peer-to-peer overlay. To discover a virtual sensor, one can either specify the identifier of the virtual sensor being sought, or provide meta data describing the sensor. In the latter case, a keyword search is performed to find matching sensors. The actual discovery is performed using a peer-to-peer approach. The

meta data for the virtual sensors is stored in a peer-to-peer directory, supporting scalable discovery in large networks [170].

A key limitation of the sensor discovery mechanism in GSN is that it does not support the search for sensors based on their current readings. While data stream processing could be used to find sensors with a given current output value, this would require communication with all discoverable sensors, which would certainly not scale up for use with large numbers of sensors.

SenseWeb, a research project from Microsoft Research, is a platform for collaborative sensing [171, 172, 173]. Users can stream their sensor readings to sensor gateways, which register with the SenseWeb core and provide a SOAP-based API for retrieving sensor information and readings. SenseWeb maintains a central repository of sensor meta data. However, sensor readings are not streamed from the sensor gateways to a central sink. SenseWeb only supports the search for sensors by static meta data such as the name of the sensor or its publisher, but not by their current readings. In addition to keyword-based searches of the meta data, geospatial queries are supported for discovering sensors in a certain geographical region typically described by a polygon [173]. The SenseWeb API allows users to create applications based on real-time sensor data. An example application that provides a geographical representation of registered sensors on a map called SensorMap[32] is already provided.

Christophe et al. also address the problem of search in the Web of Things [142]. They focus on semantic descriptions, suggesting several ontologies for connected objects but also address the problem of matching different ontologies. Effort is also spent in predicting the context of a search request, i.e., whether a human or a program initiated the search. While they claim that humans require "quick results", they do not elaborate on how this can be achieved at a global scale. Real-time search for dynamic content such as sensor readings is not addressed.

Pachube [33] was a commercial service platform that is "built to manage the world's real-time data" [174]. Users could send their real-time data to the platform and create triggers that fired when certain conditions

[32] http://www.sensormap.org/sensewebv3/sensormap/
[33] http://www.pachube.com

were met (e.g., a threshold was passed). Pachube published the data feeds in various formats and also provided different visualizations for this data, such as a dial. Feeds could be searched by "text, tag, username or location coordinates" [175], however, was not possible to search for data streams based on current sensor readings. Pachube was later re-branded as Cosm and Xively, along with a shift of business focus.

Traderbot [34] was a financial real-time search engine which enabled users to find stocks that matched their criteria in real time. Traderbot claimed to be able to search "over millions of real-time trades on 10,000+ stocks" [176]. It went out of service in 2007. Archived versions of the web pages are available[35] which lists - among several other possibilities - the search for stocks based on price range, price momentum, and volume range as possible queries. Users could also create complex custom searches such as "Find all stocks between $20 and $600 where the spread between the high tick and the low tick over the past 30 minutes is greater than 3% of the last price and in the last 5 minutes the average volume has surged by more than 300%." [176], for example.

2.8 Summary

In this chapter, we presented a novel approach for searching in dynamic, distributed data in real time, and its application to the Web. In contrast to existing approaches, our prototypical search engine for the physical world is based on an open architecture which does neither require a global view of the world's state nor a limitation of the search space, while still providing accurate results in real time. Similar to today's Web search engines, our solution is based on publishing data on the Web and making it searchable. Users can then search for physical entities with a certain current state, e.g., for restaurants that are currently well-attended and quiet. Just like for today's Web search, data publishers do not need to register with a search engine in order to make their content searchable, resulting in the same loosely coupled approach that enables multiple search engines to leverage published data.

At its core, our approach is based on the assumption that sensor-generated data, while highly dynamic, may follow certain patterns that make it predictable, at least to a certain degree. For example, the

[34] http://www.traderbot.com (defunct)
[35] http://wayback.archive.org/web/*/http://www.traderbot.com

occupancy rate of a restaurant may exhibit certain patterns during the course of a day, a week, and a year. Building on this assumption, we utilize prediction models that calculate probabilities for future sensor data, based on a sensor's past readings. These models are used by our search engine in order to find physical entities that currently feature a searched state, in real time.

We evaluated our concept using a real-world data set featuring data from a bicycle sharing system over the course of several months both conceptionally in Matlab as well as practically using a prototypical implementation called Dyser. Our results indicate that even simple prediction models can significantly reduce the overhead and time required to find a sufficient number of results. Using Dyser, we also showed how our concept can be applied to today's Web.

Finally, our approach could possibly also be used to "search the future". Given that prediction models work sufficiently well, they could not only be used to reduce the overhead for searching for entities by their current state, but also to find entities which are most likely showing a certain state at a future point in time.

3 A Framework for the Web of Things

In the Web of Things, *sensors* and *actuators* play a central role, as they constitute the physical interface between the virtual and the physical world: They enable us to capture and change aspects of the physical world, possibly in real time. Unfortunately, the current Web architecture does not address some of the key features that are often required when interacting with sensors and actuators, in particular:

- The *historization* of past sensor readings as well as actuator values. The former is often required as a basis for queries, and the latter may be required for debugging purposes.

- The support for *notifications* based on specified events. This obsoletes the need for continuous polling of certain resources in order to detect events and enables reaction in real time upon the occurrence of such events.

- The support for *simple queries*, which enables users to analyze and export data such as past sensor readings.

- The ability to *compose* and run simple application scenarios in a decentralized way.

We address these issues by suggesting new primitives that can be used to extend today's Web architecture. We implement these primitives in a prototypical framework and evaluate them using several application scenarios. The features of the framework could be distributed among connected devices but also among different cloud computing services. This approach does not advocate a central hub but strives to extend today's inherently decentralized Web architecture.

Parts of this chapter have been published in [177, 178, 179].

3.1 Background

In this section, we provide some background on aspects relevant to a framework for the Web of Things.

3.1.1 Sensors, Actuators, and the Web

In the emerging Web of Things, special emphasis is put on *sensors* and *actuators*, as they enable the automated interaction of the virtual and the physical worlds.

Sensors may automatically capture the state of their environment, possibly at high sampling rates and independently from user requests. Sensor readings that are published on the Web may be interpreted by people in order to gather information about past and current aspects of a given environment. For example, maps of noise pollution data could assist in the decision process of finding new accommodations, data from CO_2 sensors can indicate whether a room should be ventilated, and even trivial question such as *"Did I lock the garage door?"* could be answered. This resembles the traditional use case of the Web being used to share information, which is then "manually consumed" by humans. Besides this informative aspect, sensors are used in order to *automate* processes. For example, lights can be switched automatically when presence is detected, an alarm can be set when smoke is detected, and the intensity of a ventilation system can be controlled according to the current concentration of CO_2. For this, sensor readings or data deduced from sensor readings need to be automatically processed.

Actuators enable the manipulation of aspects of their physical environment with high precision and often little latency. Exposing actuators on the Web enables users to remotely control certain parts of the physical world. Naturally, access to actuators is usually restricted to a small number of authorized persons, as their actions have physical consequences. Examples of manual control are the remote control of lights and shades in building automation, pre-cooling or pre-heating of cars, and unlocking of doors. Of much greater potential is the automatic control of actuators over the Web by services. For example, an attendance simulator service could automatically switch on the lights and control the blinds when the occupants of a flat are on vacation in order to avoid burglary. Such a service could be offered by a third party and work with multiple and heterogeneous home automation systems, as long as they provide an interface on the Web.

Combining sensors, actuators, and services available on the Web using software enables the programming of portions of the physical world, and in fact, its automation. Application logic that was previously hardwired can now be implemented in software that may be executed on an arbitrary Internet-connected computer. A simple example is the use of a light switch to control a lamp. Traditionally, the power supply of the lamp is directly and mechanically controlled by the light switch. In modern building automation systems, light switches are sensors that deliver readings that are then processed by automation software that triggers the according actuators, i.e., switching on the configured light. Providing the application logic in software has the advantage that changes of the functionality (such as switching on an additional light) require less effort and are therefore quicker and cheaper to realize. Such systems usually rely on dedicated protocols that all components have to understand, such as KNX[1], and are tightly coupled. In contrast, in a Web of Things, these three components (light switch, control application, and light) do not need to be tightly integrated. In our example, light switches and lights could be based on different communication standards, as long as they provide a Web interface to their functionality. The automation software may be provided by a cloud service or run on a local computer and utilize the exposed Web interfaces of the light switch and the lamp to provide the configured functionality. Additional functionality could be implemented not only by extending the existing automation software but also by other applications. For example, switches could be reconfigured to control the blinds, and lights could be used to signal an emergency by flashing. In summary, this introduces a high degree of flexibility, which eventually turns the physical world into a platform for different applications. Note that there is some similarity to the concept of so-called mash-ups on the Web, which provide an additional value by integrating data from multiple sources available on the Web into a single view that is easily comprehensible to the user.

Such applications may range from the equivalent of plugging in a cable to connect two devices to complex automation scenarios, which strive to make parts of our environment "smart". We assume that in most cases, these applications need to be tailored to specific needs. While there surely is common functionality, different configurations are required for different installations.

[1] http://www.knx.org

The examples in this section have been chosen deliberately to indicate that the Web of Things is expected to have a profound impact on our everyday life and is not merely a tool for experts of certain scientific domains. Unfortunately, the existing Web architecture lacks specific features that arise when working with sensors, actuators, and automation scenarios.

3.1.2 Function-centric vs. Data-centric Access

In today's Web, two fundamentally different paradigms of modeling the access of a remote system are used.

In the *function-centric* approach, accessing remote resources is modeled as a function call (remote procedure call, RPC). For example, to retrieve the current reading of a sensor, one would call

```
y = getSensorReading(531).
```

This would execute a remote function called *getSensorReading* for the sensor with identifier *531* and return the result in the local variable *y*. In the Web, a popular implementation of this approach is provided by SOAP [63], a protocol that runs on top of HTTP (i.e., it utilizes HTTP as a transport layer).

In a *data-centric approach*, access to a remote resource is modeled as accessing a remote variable. The four basic operations of data access are create, read, update, and delete (abbreviated as CRUD), which are implemented by HTTP's standard methods `POST`, `GET`, `PUT`, and `DELETE`. For example, to retrieve the current reading of a sensor, one would call

```
y = GET /sensors/531/reading.
```

This would retrieve the contents of the remote variable */sensors/531/reading* and assign them to the local variable *y*. In the Web, this approach is implemented by HTTP itself. The underlying principle of HTTP is denoted Representational State Transfer (REST) [65].

3.1.3 Application Silos

There is a current trend to provide an API for physical entities on the Web in order to monitor, manipulate, and even program them and their physical environment through their built-in sensors and actuators. Unfortunately, the lack of standards in this area has lead to a

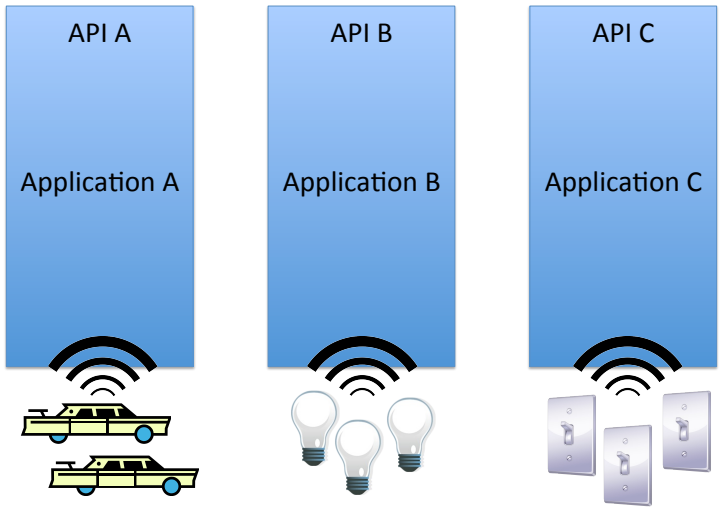


Figure 3.1: Vertical Application Silos.

large diversity of such interfaces, in functionality as well as syntax and semantics. The APIs are usually provided as a Web service by the manufacturers of such products. This leads to vertical *application silos*, where different APIs are exposed for each product or manufacturer. This approach is depicted in Fig. 3.1.

While those APIs are all based on HTTP, which simplifies interoperability, even simple automation operations such as if/then rules require programming skills and, more importantly, an additional server for the code to run. In the above example, assume the user wants to link a light switch and a lamp, which are both managed by different systems. In this case, he has to write a program that interfaces both the API of the light switch and that of the lamp and deploy it on a server. This is a significant effort for a very simple control scenario that is the analogue of running a cable.

3.1.4 Central Hubs

Another approach of exposing things on the Web is to use dedicated hubs that can be used by a large variety of different physical entities and provide a consistent interface, for both users and third-party applications. These hubs may provide additional functionality such as support for different data visualizations, data analytics, and the composition of application scenarios. Things may either connect directly to the APIs of such hubs, or hubs may include physical entities through the APIs provided by their application silos. Fig. 3.2 illustrates the

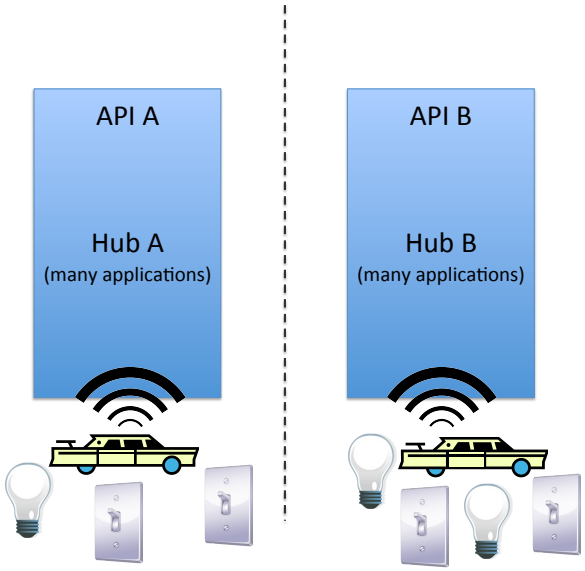


Figure 3.2: Central Hubs.

concept.

While this concept can significantly simplify the effort for the interaction with physical entities over the Web, it does not leverage the Web *per se* for the creation of application scenarios. The features are provided by the hub, which may be a proprietary application, and the application logic does not need to be exposed on the outside. Interacting with devices that are not under the control of the hub may result in the same problems that arise with application silos. In this scenario, the Web is not used as a platform for things but just hosts another Web application.

3.2 Problem Statement

We argue that both application silos and centralized hubs do not fully utilize the potential of the Web. Application silos usually lack certain functionality required to compose application scenarios, while central hubs may provide this functionality but hide the implementation details of realized application scenarios within their system.

The "traditional" Web generates much of its value by using hyperlinks to refer to relevant content as well as to syndicate and render such content in a single Web page. Content may stem from arbitrary parties and be under control of different authoritative domains. Users can inspect the source files of Web pages in order to learn about the techniques used behind the scenes.

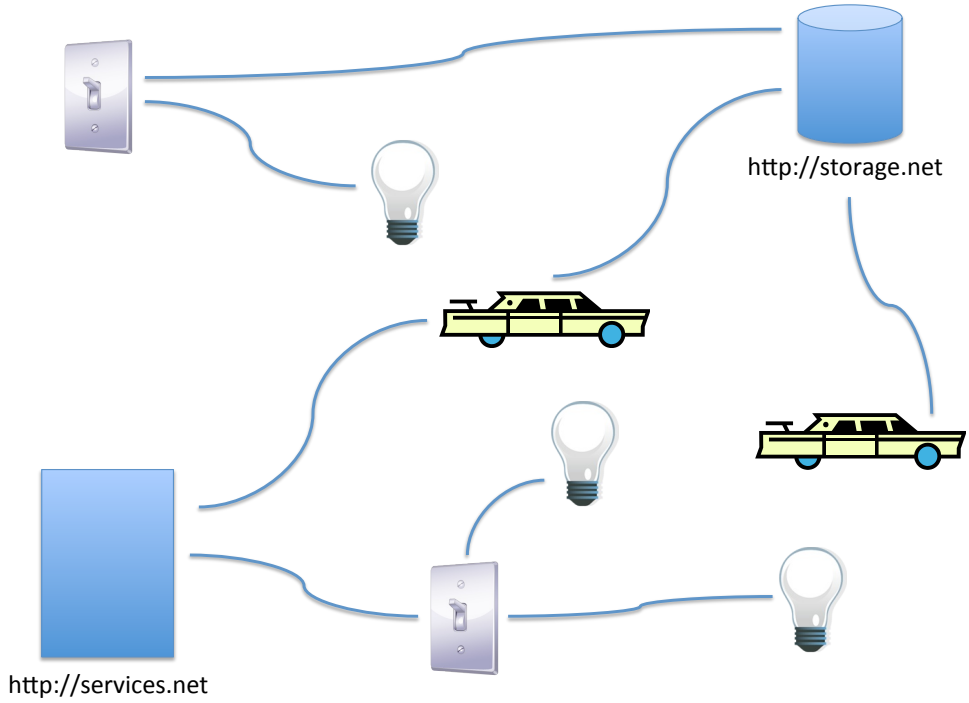


Figure 3.3: The Web as a Platform.

Similarly to the traditional Web, we strive to provide basic functionality required for a Web of Things without the need to realize such scenarios in a single authoritative domain. Additionally, the implementation of realized application scenarios should be exposed in a similar way as in the traditional Web, empowering the user by enabling him to understand the working principles.

3.2.1 Requirements

Based on the background introduced, we deduce several requirements for a framework for the Web of Things, which are listed below.

Support for Sensors and Actuators Naturally, a framework for the Web of Things needs to provide dedicated support for sensors and actuators. These should be "first-class citizens" of the framework, and it should be simple to integrate new sensors and actuators with the framework.

Support for Historical Data The framework needs to support *versioning* of resources. This requirement stems in particular from the need to support sensor readings, which can be represented as resources. For

sensor readings, keeping track of past values is important for determining context, visualizing past readings, and correlating readings with other data, for example.

Subscriptions and Push Notifications The current Web is still mostly based on polling. However, when working with sensor readings, it is impracticable to rely on polling, as it would generate a large communication overhead, does not scale to the expected size of a Web of Things, and does not provide real-time updates. To monitor sensor readings for changes, subscriptions to resources and a push-based notification mechanism are required.

Decentralized Architecture Instead of relying on a central hub that has a global view of all connected sensors and actuators, we argue that a framework for the Web of Things needs to be able to run under different administrative domains while those instances are still able to interact with each other.

Support for Queries (Filtering and Expressions) It should be possible to run simple queries on current and past sensor readings. Example use cases include retrieving the last 5 minutes of sensor data or checking whether a sensor has read a certain value during a given time period.

Support for the Composition of Simple Application Scenarios Since we expect that controlling actuators based on the current readings of a set of sensors will be a common task in the Web of Things, a framework should feature built-in support for the composition of simple application scenarios, such as linking sensors and actuators.

Support for Existing Web-related Infrastructure Since there is already a large infrastructure on and around the Web, such as Web services, resources featuring sensor readings, and applications interacting with Web resources, a framework for the WoT needs to be able to leverage such existing infrastructure to a large extent.

Access Control As for the existing Web, the access to certain resources needs to be restricted. This is particularly important for the Web of Things, as (1) resources interact with the physical world and (2) a

central element of the Web of Things is to expose a large variety of physical entities as resources on the Web. Unauthorized access to a sensor could pose a privacy breach, while unauthorized access to an actuator could enable an attacker to cause severe physical harm.

Traceability In addition to the ability to *control* access to resources, a framework for the Web of Things is also required to *monitor* access to resources. Even authorized clients may take undesired actions, and to make them accountable, access to resources needs to be traceable.

3.2.2 Focus

The focus of this work is to provide support for basic application scenarios in a future Web of Things. In particular, we identified two classes of application scenarios that we particularly want to address:

Data Syndication and Processing Scenarios Such application scenarios address the inclusion of data from possibly multiple and heterogeneous sources (such as sensors and services), which could be under different authoritative domains, into third-party applications.

Simple Control and Automation Scenarios This category of application scenarios covers applications that manipulate aspects of the physical world. This manipulation may be triggered manually or automatically by sensors.

While we strive to address most of the requirements listed above, our solution does not address *access control* or *tracability*.

3.3 Approach

Instead of creating complex Web services for each specialized requirement, we argue that application scenarios should be composed of simple building blocks. The flexible composition of such building blocks enables the creation of a large variety of specific applications[2]. To this end, we believe that exposing elements as Web resources is a beneficial strategy for modularization, as it simplifies access and therefore fosters the (re)use of such components. For this, we follow a data-centric

[2]A concept that has proven its benefits with Unix pipes, for example.

approach and resort to the REST principle [65] of HTTP. URLs play an important role in our concept – we see them as vectors through information space [180] that contain semantic hints for the user.

3.3.1 Design Principle

In order to integrate physical entities with the Web, each physical entity is modeled as a Web resource that serves as its virtual counterpart. Just like any Web resource, it is identified by its URL and can be accessed over the HTTP protocol. Using URLs to identify physical entities enables well-known operations such as bookmarking, sharing on the Web, referencing the resource in documents, and providing "physical hyperlinks" by including them in printed QR codes. Such virtual counterparts can have multiple representations but should always support HTML [94, 95] for the depiction in a Web browser. Physical entities feature a set of properties that describe certain aspects of their instance. For example, a property named *productiondate* could hold the date of something's production. Properties may also represent the current state of an entity. A state may be internal to the entity, such as the amount of time it was powered up, or external, such as the state of its sensors and actuators. Retrieving the current reading of a sensor is therefore in principle the same operation as retrieving data from an arbitrary location on the Web. The control of an actuator is performed by updating one ore more of its properties with desired values. This follows, in principle, the same pattern as updating data on the Web. All properties of the entity are modeled as separate sub-resources that are mapped below the URL of their entity.

Note that exposing functionality as separate resources is a key principle of our approach. Having the possibility to directly address specific variables can reduce the amount of transferred data and simplify parsing. However, one is not required to access several properties of an entity separately, as there is also a single representation that includes all sub-resources.

On a protocol level, we utilize a data-centric approach that follows the REST principle. We limit the use of HTTP methods to `POST`, `GET`, `PUT`, and `DELETE`, which map to creating, reading, updating, and deleting data (CRUD). Additionally, the `HEAD` operation is supported, which returns a response consisting only of the HTTP headers a corresponding `GET` response would return. A `POST` to a base resource creates a

new resource whose URL is returned in the `Location:` header of the response. `PUT` updates an existing resource which is specified by the supplied request URI and stores the enclosed data, or it can also be used to create a new resource at a specified location. `DELETE` removes a specified resource from the system. For each resource, we keep both its creation date and its last modification date. In contrast to HTTP-based protocols such as WebDAV [181], we do not introduce additional HTTP methods, in order to keep the system simple and compatible to a large number of clients.

3.3.2 Typed Resources

All of the resources provided by our framework are explicitly typed. The resource type is analogous to data types in programming languages and defines the functionality that is offered by our framework for a given resource. Resource types are identified by URLs and included in the headers of an HTTP request or response, specified by a header called `Resource-type`[3]. The resource type is orthogonal to the well-known content type, which specifies only the type of representation of the resource.

3.3.2.1 Atomic Types

Atomic types are data types that do not contain other data types. We consider the following atomic types:

Simple Type This is the basic atomic data type that is intended to hold any value that can be understood by users when rendered as string. As such, it resembles a primitive data type in programming languages. Boolean values, numbers, and plain text are all examples that can be stored in a resource of simple type.

Reference Type This type stores a URL. It resembles pointers known from programming languages or aliases known from file systems by redirecting access to another resource. This is implemented by returning the HTTP status code 307 (*Temporarily Redirected*) and the stored value in the `Location` header when content is retrieved using a `GET` request. Browsers and HTTP libraries usually automatically follow this redirection.

[3]Note that such a header was proposed several years ago, but the idea did not catch on [182].

```
{
    "temperature":
    {
        "current": 9.2,
        "target": 6.0
    }
    "open": true
}
```

(a) http://fridge.example.net/

```
{
    "temperature": "http://fridge.example.net/temperature",
    "open": true
}
```

(b) http://fridge.example.net/?level=1

Figure 3.4: Retrieving the properties of an object at varying hierarchical depths.

Binary Data Type Similar to a BLOB[4] in SQL [110], this data type is intended for storing binary data. Since the format is unknown to the framework, it cannot provide features that depend on the interpretation of its content (e.g., different representations).

Note that for atomic types, we somewhat relax our previous definition of `POST` and `PUT` – both can be used to update atomic values.

3.3.2.2 Objects

The object type is the basic element of representing physical (and also virtual) entities. As outlined in Sect. 3.3.1, physical entities may contain a number of properties. Each property is directly addressable, and its URL is constructed by appending its name as a new segment to the URL of its parent. Properties can be structured hierarchically, so the parent may be either an object or another property.

Properties are associated with their parent elements by sharing parts of their URL not only syntactically but also semantically: When the representation of an object is retrieved using a `GET` request, all its properties are included. As properties can be hierarchically structured, this creates a tree structure that is then rendered in the requested format. The number of hierarchical levels that should be returned in the response can be specified using the URL parameter *level*. Elements that exceed the hierarchical level or cannot be serialized in the given format are included as reference. An example of this concept is shown

[4]Binary Large Object.

```
{
    "temperature":
    {
        "current": 9.2,
        "target": 6.0
    }
    "open": true
}
```

(a) http://fridge.example.net/

```
{
    "current": 9.2,
    "target": 6.0
}
```

(b) http://fridge.example.net/temperature

```
9.2
```

(c) http://fridge.example.net/temperature/current

Figure 3.5: Addressing an increasingly specific part of an object.

in Fig. 3.4. Of course, this concept also works at deeper levels of the property tree.

In order to address an increasingly specific part of the object's properties, one can navigate along the URL of the object, down to the specific property. There is some analogy to the refinement of a search query, which also results in increasingly specific results. See Fig. 3.5 for an example. Note that this concept can be used not only to retrieve but also to update properties of an object, using the `PUT` method.

This flexible concept of addressing parts of an object's properties can ease development and increase performance, as it obviates downloading or uploading and parsing a potentially large list of properties when only a single property is of interest.

Object resources are available in the HTML, JSON and Atom formats.

3.3.2.3 Collections

Supporting collections of values (such as sensor readings) is an important aspect of a framework for the Web of Things. For this, we support a collection resource type. It represents a set of items and can contain resources of any type. Resources are added to the collection by `POST`ing them to the collection URL, which will return the newly created URL

of the item in the `Location` header of the HTTP response. Items of a collection can be fetched, updated, and deleted using `GET`, `PUT`, and `DELETE` with the item's URL. `GET` and `DELETE` can also be used on the collection URL.

When an item is added to the collection, it is assigned an identifier that can be used to uniquely address the item within the context of the collection. The resource of each item in a collection is modeled as a sub-resource of the collection URL by adding a segment containing the identifier of the item. For example, when adding the URL

`http://mystuff.example.net/officekeys`

to the collection

`http://drawer.example.net/contents`,

it could return

`http://drawer.example.net/contents/23`

as the URL for the item in the collection.

There is also an index-based access pattern for the items of a collection, based on the order of their creation, starting with the index 1. To be able to differentiate between identifier-based and indexed-based access, indices are prepended by the term "`i:`". For example, the second element added to

`http://drawer.example.net/contents`

can be addressed as

`http://drawer.example.net/contents/i:2`.

This enables the addressing of elements of a collection relative to the first inserted element. In order to enable the addressing of elements in a collection relative to the last inserted element, the keyword `last` is supported as a valid index referring to the last element added to a collection. For relative addressing, it is possible to subtract an integral number $(last - n)$. For example, the penultimate item that was added to a drawer could be identified by

`http://drawer.example.net/contents/i:last-1`.

Collections also feature a dedicated property named `count`, which represents the number of included items. To stick with our example of a drawer, the URL

`http://drawer.example.net/contents/count`

would represent the number of contained items.

A collection can be represented in the HTML, JSON, ATOM, CSV, and TXT formats. Additionally, a representation in the iCalendar [183] format is provided, where each event corresponds to an item in the collection, starting at its creation time and lasting until the creation time of the next item. This provides a convenient method to include time-series-based data (such as sensor events) in calendar applications (see Sect. 3.3.4).

Collections support numerical sensor readings by offering a graphical representation of their values, ordered by their creation date. Additionally, the built-in functions *min*, *max*, *avg*, and *sum* can be used on collections of numerical values to return the minimum, maximum, and average value and the sum of all values. This is achieved by appending the function name as an additional segment to the URL of the collection. For example, a collection of temperature readings identified by

`http://sensors.example.net/temperatures`

can be used to return the minimum temperature reading by requesting

`http://sensors.example.net/temperatures/min`.

3.3.3 Meta-URLs

Resources are often intimately connected with each other. For example, the history of a resource is intrinsically tied to the original resource. To this end, we introduce the *Meta-URL* concept to access such meta-resources of a given resource by appending a well-defined term to the URL of the base resource, separated by the delimiter `@`, which acts as an escape character.

For example, by appending the term `@history` to a given URL, we will address the list of past versions of the resource denoted by the base URL. In our use case,

`http://myhome.example.net/lamp/power@history`

will provide the history of

`http://myhome.example.net/lamp/power`,

i.e., the usage history of the lamp.

This approach allows users to instantly deduce the Meta-URL from a given base URL and put it to use (for example, in a Web browser). In our framework, Meta-URLs are used for accessing version history and manipulating subscribers of a resource, which is discussed in the next sections.

3.3.4 Versioning of Resources

When working with sensor readings, it is often essential to be able to access historic readings of a given sensor. For example, one might be interested in the maximum reading of a given day or plot the readings of the last hour. The historization of data is thus an important aspect.

For that reason, we provide a generic approach for the versioning of resources: Each resource that is managed by our framework keeps track of its past values in a separate resource, which is a collection in type. This resource is identified by appending the term `@history` to the URL of the base resource, creating a Meta-URL. Since the history is a standard collection, it provides all the functionality introduced in Sect. 3.3.2.3. Versioning in our frameworks is a generic approach that is not limited to numeric values but can be used with arbitrary data. For example, the historic data of a temperature sensor that is identified by

`http://sensors.example.net/temperature`

can be accessed at

`http://sensors.example.net/temperature@history`.

Its first recorded temperature can be accessed at

`http://sensors.example.net/temperature@history/i:1`,

its last recorded temperature at

`http://sensors.example.net/temperature@history/i:last`.

3.3.5 Observation of Resources

A key mechanism to enable composition of components is to inform a component about state changes (i.e., events) in another component. This is realized as a simple publish/subscribe mechanism. Similar to

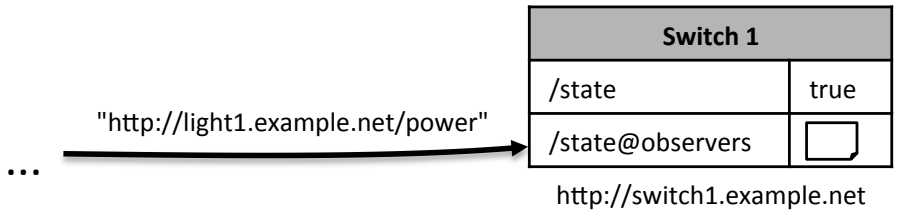


Figure 3.6: Subscribing to changes of a resource is performed by adding an URL to the collection of its observers.

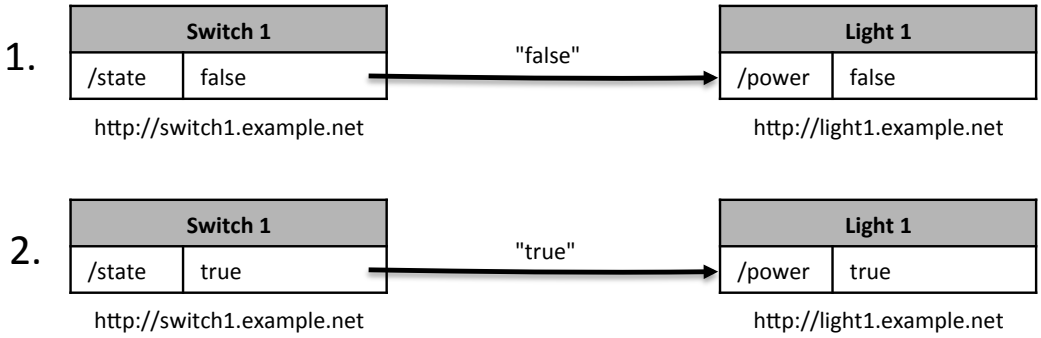


Figure 3.7: On changes of a resource's value, all subscribed resources are updated with its new value. This can be used to connect components (in this example, a light switch and a light) and forms the basis for the composition of application scenarios.

the concept of webhooks [184], our eventing mechanism is based on HTTP callbacks, where a URL registered with a specific *event* is *notified* as soon as this event occurs.

In our approach, an *event* is always the change of data identified by a Web resource. *Observable resources* are resources that support subscriptions to changes in their associated data. *Observers* are resources that are subscribed to one or more *observable resources*. Note that any resource that supports the `POST` operation may act as an observer and that multiple observers may be subscribed to a single observable resource. As soon as the data of an observable resource changes, all registered observers are *notified* by a `POST` request[5] sent to their respective URLs. For each observable resource, there is an associated resource of type collection that represents the list of registered observers. The URL of this collection is a Meta-URL, which can be deduced by appending the term *@observers* to the URL of the observable resource. Using a collection of type resource for the list of observers provides the same functionality as specified in Sect. 3.3.2.3. In particular, a new subscrip-

[5]To increase interoperability, we currently resort to `POST` instead of using the more appropriate `PUT`.

tion can be performed by simply performing a `POST` request containing the URL of the observer on the list of observers (see Fig. 3.6). Likewise, subscriptions can be removed by performing a `DELETE` request for the corresponding entry. It is also possible to retrieve the list of all observers using a `GET` request on the list.

For example, consider an observable resource which reports the state of a door,

`http://myhome.example.net/frontdoor/open`.

As soon as the door is opened, the data associated with this resource changes from "false" to "true", and all registered observers are notified. The list of observers for this sensor can be accessed at the Meta-URL

`http://myhome.example.net/frontdoor/open@observers`.

Notifications materialize as `POST` requests to the subscribers, containing the updated value of the resource that triggered the notification in the payload. For the resource types simple and reference, the updated value is passed directly in the payload. For the resource types binary, object, and collection, a permanent URL identifying the value that triggered the notification is passed in the payload. This is to reduce the overall amount of transferred data, as the subscriber might not act upon the updated value by itself, but instead hand it over to the next component. The approach is comparable to passing variables in functions of popular programming languages, where variables can be passed to a function *by value* or *by reference*.

Let us assume that our above example of a door located at

`http://myhome.example.net/frontdoor/`

is actually of resource type object and that "open" is just one of its properties. To subscribe to any changes in the door's properties, one would register an observer at

`http://myhome.example.net/frontdoor@observers`.

As soon as the state of the door changed, a new entry in the history of the resource would materialize, such as

`http://myhome.example.net/frontdoor@history/832`.

Then, notifications would be sent to all observers that included this exact URL, referencing the entry in the history of the resource that

triggered the notification. Since the reference would be permanent, it could be used at any time to access this specific value of the resource.

Observers may subscribe to multiple resources. In order to be able to distinguish where a received notification originates from, each notification includes the *Referer* header, set to the URL of the observed resource. Note that by introducing intermediary hubs (as in [185], for example), which relay a single notification to multiple observers, we expect the concept of HTTP callbacks to scale to large numbers of observers and notifications.

An important aspect of our approach is that notifications pass data using standard `POST` requests. This enables all Web resources that accept `POST` requests and can interpret the payload format to be subscribed to changes of a Web resource that is managed by our framework. More importantly, this feature enables the simple composition of resources to create basic application scenarios. A minimal example would be the use of a sensor for direct control of an actuator. For this, the resource representing the desired value of an actuator would be subscribed to the resource of the current value of a sensor. Whenever the sensor reading changed, a notification would be sent to the registered resource of the actuator. For example, the power state of a lamp (a binary actuator) could be subscribed to the position of a light switch (a binary sensor). Each flip of the switch would generate a notification that would be sent to the resource of the power state of the lamp, resulting in control of the lamp by the light switch. This approach of composing resources is not limited to sensors and actuators, but can be used to couple arbitrary resources. Our framework also provides support for composing more sophisticated application scenarios, which is discussed in the next sections.

3.3.6 Representations

Besides using HTTP's content negotiation in order to return the correct representation of a resource to the client, we support a manual override that allows the client to specify the requested format explicitly as part of the URL. This has proven to be useful when inspecting resources in a Web browser. The request format is specified by appending its common file name extension to the requested URL. For example, to retrieve the JSON representation for the Web resource

`http://drawer.example.net/contents`,

one would request

`http://drawer.example.net/contents.json`.

3.3.7 Expressions

When working with sensor data, it is often necessary to pre-process or select certain readings of a sensor. For example, one might want to test readings against certain thresholds or select sensor readings of a given time frame. Our framework provides support for basic preprocessing and selection tasks, which also work in conjunction with the versioning and publish/subscribe mechanisms introduced before.

This functionality is realized by expressions, which can be added to the URLs of resources managed by our framework. The expression is then evaluated based on the value of the resource, and the result is returned to the caller. Expressions are not limited to resources representing sensor readings but can generally be applied to resources managed by our framework.

3.3.7.1 Temporal Expressions

All managed resources support expressions regarding the date of their last update. For this, we use a simple syntax that enables us to test whether a resource was updated *before* or *after* a given point in time, or *during* a specified time period. The expression evaluates a resource as either *true* if the condition holds, or *false* otherwise. Time can be denoted as an absolute date or relative to the current time. To state an expression based on a resource's last update time, the keyword *updated* is appended to the URL of the resource, followed by a temporal expression. For example,

`http://fridge.example.net/temperature updated before 01-02-2012 8:00:00`,
`http://fridge.example.net/temperature updated after 01-02-2012 8:00:00`, and
`http://fridge.example.net/temperature updated during 01-02-2012 8:00:00 - 03-02-2012 12:00:00`

check whether the *current* reading of the temperature sensor was taken before the timestamp 1.2.2012 8:00, after the timestamp 1.2.2012 8:00,

or during the time period between 1.2.2012 8:00 and 3.2.2012 12:00[6]. The keyword *during* can be omitted to make the expression shorter.

Instead of providing an absolute timestamp, a timestamp relative to the current time can also be specified by denoting the number of seconds, minutes, hours, or days relative to the current time. This is denoted by specifying a point in time as *x seconds ago*, *x minutes ago*, *x hours ago*, or *x days ago*. For example,

`http://fridge.example.net/temperature updated before 30 secs ago`,
`http://fridge.example.net/temperature updated after 5 days ago`, and
`http://fridge.example.net/temperature updated during 6 hours ago - 30 mins ago`

test whether the current reading of the temperature sensor is older than 30 seconds, was updated less than 5 days ($5 * 24$ hours) ago, or was recorded between 6 hours and 30 minutes ago. For convenience, a timespan that extends back from the current time can also be denoted using the keyword *last* and a time offset:

```
http://fridge.example.net/temperature
    updated during last 30 secs
```

returns *true* if the temperature has been updated within the last 30 seconds. This can also be shortened to

```
http://fridge.example.net/temperature
    updated last 30 secs
```

since the keyword *during* is optional. Finally, there are special keywords for the current day and the day before the current day:

`http://fridge.example.net/temperature updated today`, or
`http://fridge.example.net/temperature updated yesterday`.

The first example will only return *true* if the temperature has been updated on the current day and the second example will only return *true* if the temperature was updated during the day preceding the current day.

[6]Note that the specification of seconds is optional.

3.3.7.2 Expressions on Simple Data Types

Resource values that can be denoted as strings additionally support the direct comparison to a literal. This is achieved by appending "=" (test for equality) or "!=" (test for inequality) to the URL of the simple value, followed by a string or a number that should be compared to the value. The result is a textual representation of true or false. For elements that hold a numeric value, the additional comparators "<", "<=", ">", and ">=" can be used, and they work just as in popular programming languages. For example, the resource

```
http://fridge.example.net/temperature > 10
```

will return *true* if

```
http://fridge.example.net/temperature
```

is currently above 10 ℃, or *false* otherwise (the ">" sign needs to be escaped, which is done automatically by most browsers). This way, simple threshold checks can easily be constructed and are evaluated in the context of the managed resource.

3.3.7.3 Expressions on Collections

Besides the basic functionality of performing temporal expressions based on the date of their last update, collections additionally support expressions based on their included elements.

Expressions on Included Elements Evaluation of expressions based on collections is supported by two directives: the universal quantifier ∀ and the existential quantifier ∃ originating from predicate logic. To test whether a condition holds true for all elements of a collection, the term `forall` is appended to its URL, followed by the condition that is evaluated on all of its elements. Similarly, when the term `exists` is appended to the URL of a collection, followed by a condition, it will return *true* if at least one element of the collection evaluates the specified condition to be *true*. For example, the resource

```
http://fridge.example.net/temperature@history exists > 10
```

will return *true* if the fridge's temperature has risen above 10 ℃ at some time in the past, and the resource

```
http://fridge.example.net/temperature@history/
      forall updated after 60 days ago
```

will return *true* if all recorded sensor readings are no older than 60 days.

Filtering Included Elements Collections in our system support filtering of their elements, obviating the need for transferring possibly large data sets when only a few included items are of interest. Filter expressions are constructed by appending a new segment to the URL of the collection that is prefixed with "`f:`" and followed by the condition that entries of the collection need to fulfill in order to be included in the result. The result is a virtual collection that is computed on the fly and is again of the resource type collection. Filtering can be performed based on the contents of the included elements or based on the time when elements were added to the collection. Filters based on conditions of the values of contained elements utilize comparisons. For example,

```
http://fridge.example.net/temperature@history/f:>10.txt
```

will return the events when the temperature exceeded 10 °C, represented as a chronologically ordered log file with timestamps for each event. Filtering according to temporal constraints can be achieved by using a temporal expression as a filter criterion. For example,

```
http://fridge.example.net/temperature@history
       /f:updated during last 15 mins
```

would return all elements that have been added within the past 15 minutes, and

```
http://fridge.example.net/temperature@history
      /f:updated after 01-02-2012 8:00
```

would return all elements that have been added after 1.2.2012 8:00. Since the result of a filter is a virtual collection, additional filtering can be performed:

```
http://fridge.example.net/temperature@history
     /f:updated during last 7 days/f:>10
```

would return all temperature readings of the last 7 days that exceed 10 degrees.

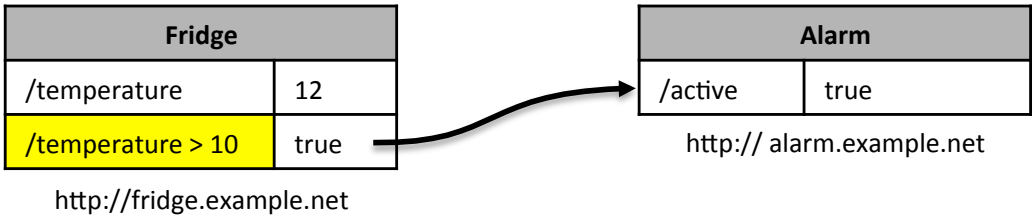


Figure 3.8: Example of the usage of an observed expression (yellow field) to create a control flow.

3.3.8 Observing Expressions

Observers can also be registered with expressions. This provides a convenient method to trigger events, and in turn serves as the basis for simple automation scenarios. For example, let us assume we want to trigger an alarm as soon as the temperature of the fridge exceeds 10 °C. Let us further assume that the alarm can be triggered by sending *true* or *false* to

`http://alarm.example.net/active`.

All we need to do is to add the URL

`http://alarm.example.net/active`

to the collection of observers denoted by

`http://fridge.example.net/temperature>10@observers`.

As soon as the temperature threshold is exceeded, the expression is evaluated as *true*, which is sent to

`http://alarm.example.net/active`,

starting the alarm. As soon as the temperature threshold is undercut, the expression is evaluated as *false*, which is sent to

`http://alarm.example.net/active`,

stopping the alarm (Fig. 3.8).

3.3.9 Including Unmanaged Resources

In order to be able to use features of our framework for unmanaged resources (i.e., resources outside of our framework), we introduce a component called *poller*. A poller will periodically request the value of a specified Web resource at a given interval. The polled value is then

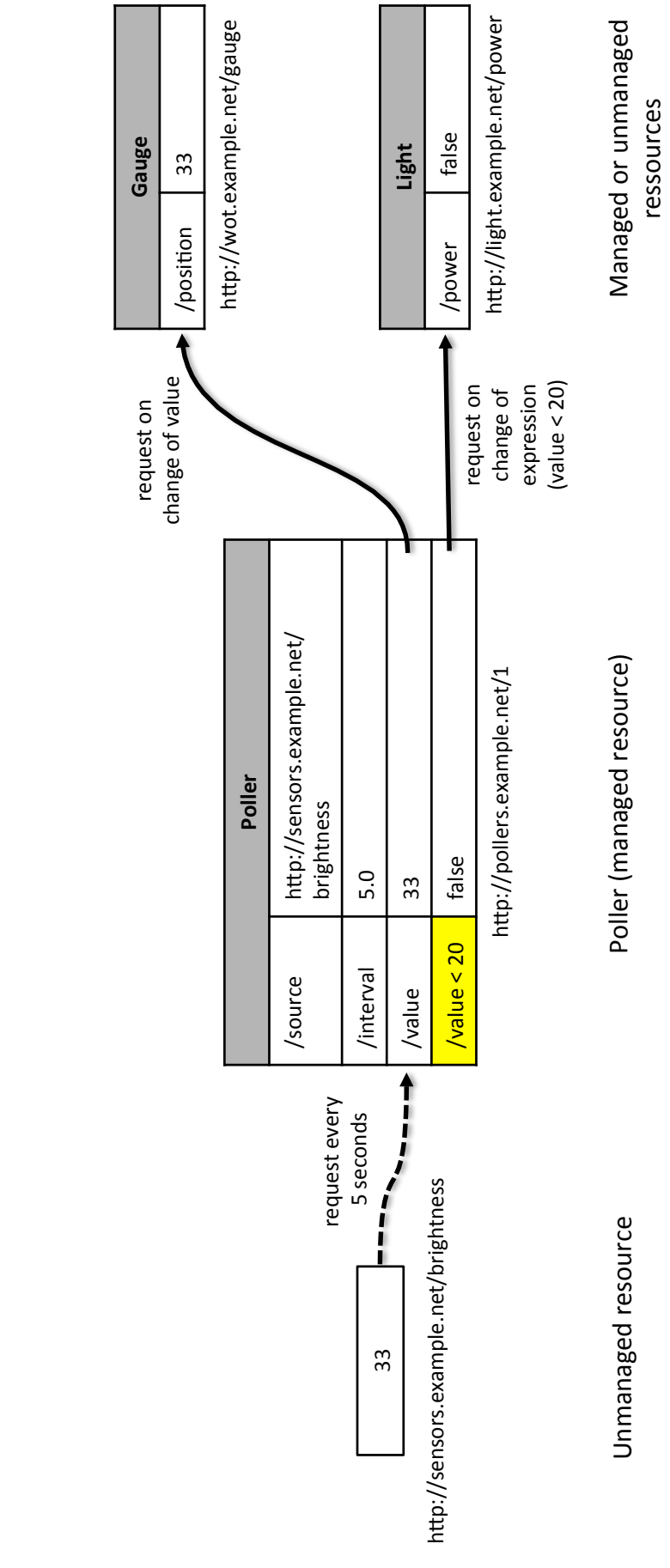
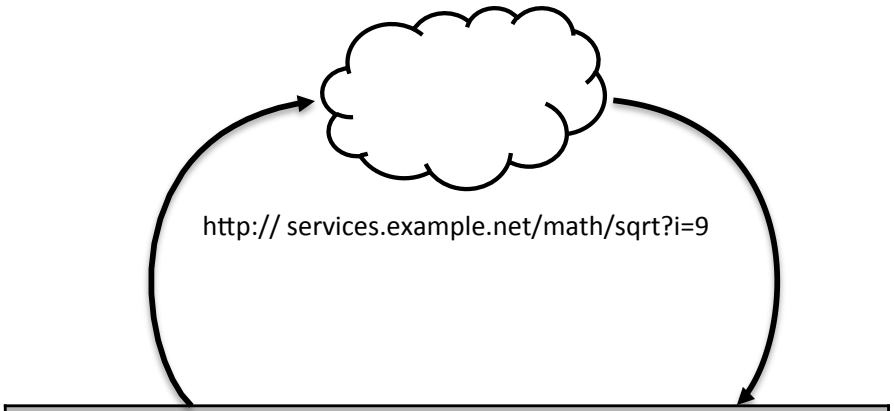


Figure 3.9: Example of a poller that is used to include unmanaged sensor readings within our framework, including the use of an expression (yellow field).



Evaluator	
/input/param	9
/function	http://services.example.net/math/sqrt?i={param}
/value	3

http://evaluators.example.net/1

Figure 3.10: An evaluator encapsulates calls to an external Web service and provides input parameters and result as managed resources.

provided as a sub-resource of the poller, placing it under the management of our framework. This enables all the features of our framework to be used on the value, including versioning, subscriptions, and expressions. Technically, a poller is an object with three properties: The URL of the resource to be polled is stored in the property *source*, the timespan between polling requests is specified in the property *interval*, and the latest polled value is available in the property *value*. An example of the usage of a poller is depicted in Fig. 3.9, in which the support for expressions and observers is also demonstrated.

3.3.10 Performing Computations

Until now, we have discussed only simple approaches for performing computations on resources, such as checking against a threshold. However, for many application scenarios, it is necessary to perform more advanced computations. For this, our framework provides a building block named an *evaluator* that enables the execution of arbitrary computations. An evaluator does not perform computations by itself, but calls a given external function with a set of call parameters and provides the return value as a sub-resource. The evaluator thus encapsulates function calls and serves as the link between our framework and external computations.

An evaluator is realized as an virtual object with a variable set of properties: the property *function* specifies the URL of a Web service that is used to provide computations. Web services that can be used in conjunction with a poller need to be stateless, be invoked using the `GET` method, and accept call parameters as part of the query string. URI templates [186] are used to specify call parameters within the *function* property. Using this notion, the values of call parameters are specified by template variables that are enclosed in braces. For example, assume there exists an external function

`http://services.example.net/math/sqrt`,

which computes the square root of an input parameter called `i`. To compute the square root of 9, one would perform a `GET` request on

`http://services.example.net/math/sqrt?i=9`,

which would return 3. To use this function within an evaluator, one would specify

`http://services.example.net/math/sqrt?i={param}`

as its *function* property. As soon as the *function* property is updated, the evaluator will check its contents for template variables. For each template variable found, it will create an input property of the same name below its property `input`. Let us assume the evaluator is located at

`http://evaluators.example.net/1`.

In our example, there is one template variable named *param*, so the sub-resource

`http://evaluators.example.net/1/input/param`

is created and linked with the value of the call parameter *i* of the specified Web service. Finally, each evaluator has a property called *value*, which stores the result of the last execution of the encapsulated function. Each time the value of an input property of an evaluator changes, a URL is constructed by substituting all template variables in the *function* property with the current values of the input properties of the evaluator. This URL is then called, and the property *value* of the evaluator is updated subsequently with the result of the function call. Our example is depicted in Fig. 3.10.

Since an evaluator is a resource managed by our framework, it features the same basic functionality that is provided for all managed resources of our framework. For example, it is possible to subscribe resources to changes of the function result (the property *value*) or access the history of a property. In particular, it is possible to specify historic values of properties in URL templates. Assume we want to compute the difference between the first and the last recorded values of a numeric sensor reading. We would specify

```
http://services.example.net/math/delta
      ?p1={param@history/i:1}
     &p2={param@history/i:last}
```

for the *function* property of the evaluator. This would generate a property named *param*, whose historic values would be used as parameters for the function calls.

3.4 Implementation

We implemented a prototypical framework of our approach in Java, using the RESTlet framework[7]. Additionally, a simple Web server running on an Android phone was developed to expose some of the phone's sensors and actuators as Web resources.

3.4.1 Functions

For our evaluations, we implemented some basic functions for use by evaluators:

- The boolean functions AND, OR, and XOR support two or more input parameters, whose values are represented textually as *true* or *false*. Likewise, the return value is textually represented. Parameters are labeled `p1`, `p2`, ... `pn`. For example, to compute the boolean AND function for three input values set to *true*, *false*, and *true*, one would perform a `GET` request to

  ```
  http://services.net/AND?p1=true&p2=false&p3=true
  ```

  which would return *false*.

[7] http://restlet.com/products/restlet-framework/

- An image comparison function that compares two images, each specified as a URL and attached as URL parameters named `a` and `b`. It returns a value between 0 (both images are identical) and 1 (maximum difference between images). An example of an invocation (character escaping omitted for reasons of readability):

  `http://services.net/imgcmp`
  `?a=http://images.example.net/foo.png`
  `&b=http://images.example.net/bar.png`.

3.4.2 Resource Factories

Our framework also supports the creation of virtual objects such as pollers and evaluators at run-time. For this, a special collection resource is used that contains only objects of the given type. Creating a poller or evaluator is achieved by executing a `POST` request to the URL of the corresponding collection that contains the parameters in the payload[8]. This approach resembles the factory pattern in object-oriented programming, in which objects are instantiated by a dedicated factory object. For example, in order to create a new poller, one might `POST` its desired configuration to

`http://pollers.example.net/`,

and the new poller might be created at

`http://pollers.example.net/534`,

which is returned to the user.

3.5 Evaluation

We evaluate our framework by implementing exemplary application scenarios from the two categories introduced in Sect. 3.2.2: *data syndication and processing tasks* and *simple control and automation tasks*. The number of HTTP requests typically executed at run-time is then compared to a scenario in which the application logic would not be implemented by our framework but run on a central hub. In such a setup, sensors and actuators would still communicate with the central hub using HTTP, but the entire application logic would be executed completely within a proprietary, monolithic system that has a "global view"

[8] Represented as JSON data or form-encoded data (application/x-www-form-urlencoded).

of all connected sensors and actuators, thus eliminating the need for performing additional HTTP requests. For this reason, it is assumed that application scenarios can be executed more efficiently when implemented on a central hub.

3.5.1 Data Syndication and Processing Scenarios

3.5.1.1 Scenario 1: Syndicate Deduced Sensor State to Personal Calendar

Scenario In this application scenario, data deduced from a sensor are included in a user's calendar application. We use a PIR sensor to automatically detect movements in a meeting room, and we deduce and publish occupancy state from these data in the iCalendar format. Calendar applications can then include these data in their regular view.

Components

- A motion detection sensor, such as a PIR[9] sensor. This is a binary sensor that delivers a boolean output: *true* if it is currently detecting motion, *false* if not. We use this sensor in order to derive the state of occupancy for a room. For this example, we assume that the sensor output is available at `http://pir.example.net/motion` and managed by our framework.
- A managed resource used to store the deduced occupancy state. It is available at `http://variables.example.net/occupancy`.
- A calendar application that can subscribe to calendars published in the iCalendar format. This is a common feature where the application periodically polls the URL of a published calendar and includes it in its regular view. For example, Google Calendar and the iCal application on Mac OS X each support this feature.

Setup In order to deduce occupancy from the reading of the motion sensor, we require the sensor to have detected motion at least once in the last 5 minutes. Since the sensor is managed by our framework, this can be formulated as

```
http://pir.example.net/motion@history
/f:updated last 5 mins/exists='true'.
```

[9] Passive infrared.

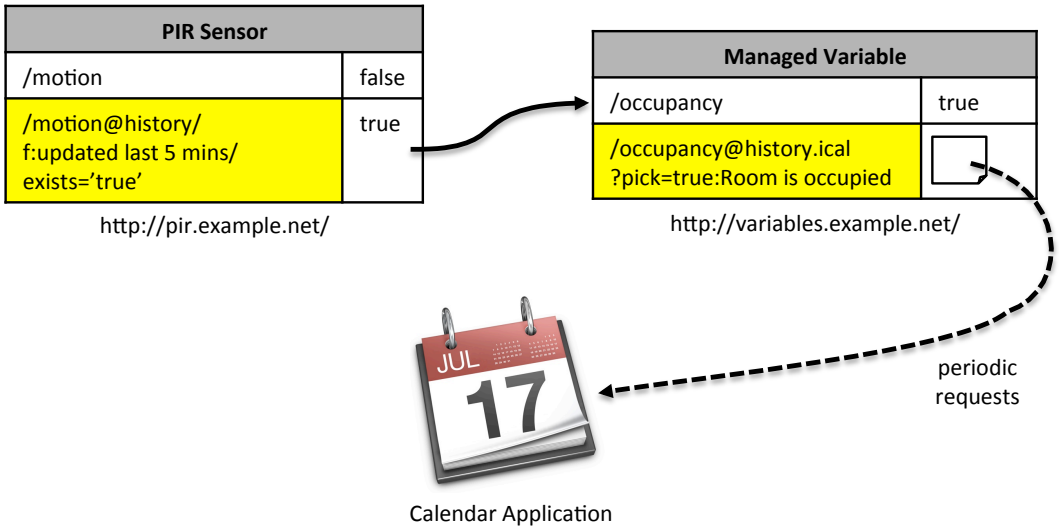


Figure 3.11: Setup of scenario 1 (computed resources are marked yellow).

In order to store the deduced state, the resource

`http://variables.example.net/occupancy`

is subscribed to this URL. To include include the occupancy state of the room in a calendar application, the history of

`http://variables.example.net/occupancy`

has to be returned in the iCalendar format. This can be achieved by appending `@history.ical` to the URL. For a more pleasant view, this representation supports a simple mapping of input states to descriptive states:

```
http://variables.example.net/occupancy@history.ical
        ?pick=true:Room is occupied
```

This picks only the state *true* and creates calendar entries denoted as "Room is occupied". The final URL can then be added to the calendar application. The setup is depicted in Fig. 3.11.

Run-time In this scenario, the occupancy state of the motion sensor is deduced in its context. HTTP requests are generated only when the deduced state changes. Additionally, the calendar application polls at regular intervals for updated data. A screenshot of a calendar application that includes the sensor data is depicted in Fig. 3.12.

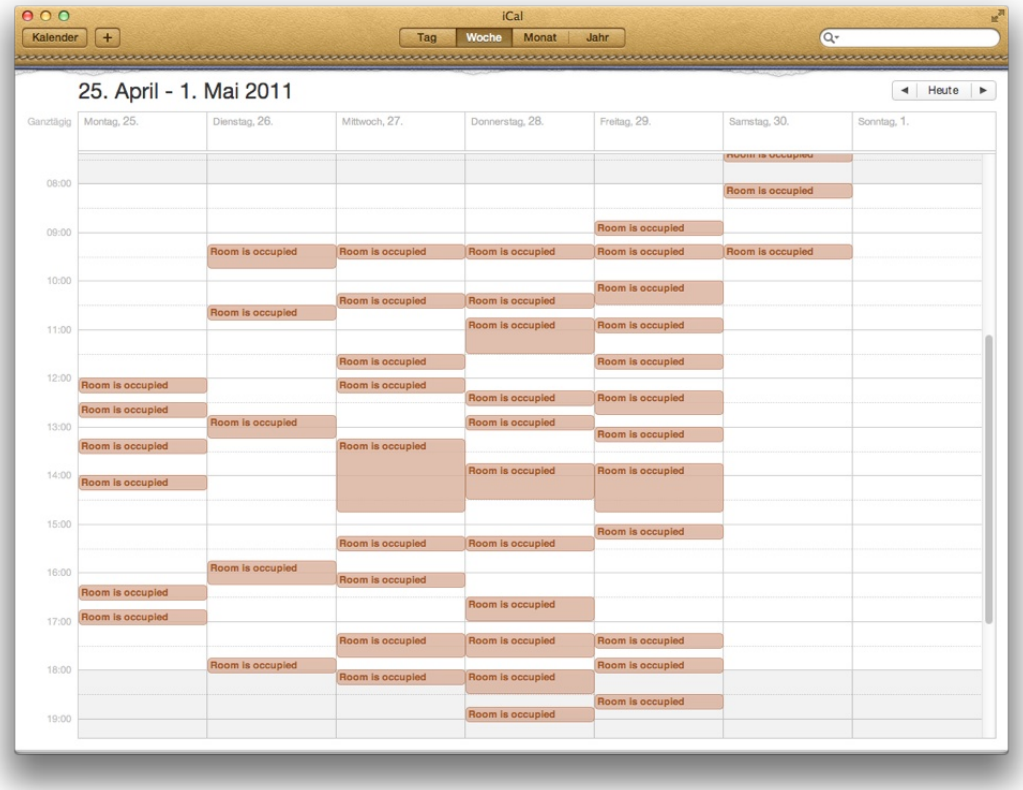


Figure 3.12: Screenshot of a calendar application with included occupancy data.

Discussion Implementing this solution with a central hub would require deduction of the occupancy state within the hub and would in turn require the motion sensor to send motion-related data to the hub. Such data are of much higher dynamic than occupancy data, which in our scenario change at most two times within five minutes. The number of requests sent by the calendar application looking for updates is considered to be identical for both approaches. The number of sent requests is thus expected to be significantly higher when using a central hub.

3.5.1.2 Scenario 2: Include Aggregated Sensor Data on a Web Page

Scenario The objective of this application scenario is to include the current reading of a temperature sensor on a custom Web page. Additionally, for the last seven days, its minimum, maximum and average readings should be included, as well as a graph depicting all sensor readings in this time frame.

Components

- A temperature sensor that outputs a numeric value. It is assumed that the sensor is managed by our framework and provides its current reading at `http://sensors.example.net/temperature`.
- A Web page. For this example, it is assumed that it is available at `http://homepage.example.net/temperature.html` and not controlled by our framework.

Setup Since the temperature sensor is managed by our framework, the data of the past seven days can be acquired from its history using

```
http://sensors.example.net/temperature@history
            /f:updated last 7 days.
```

These data can be aggregated using the built-in functions *min*, *max*, and *avg*. For example:

```
http://sensors.example.net/temperature@history
          /f:updated last 7 days/max
```

The remaining data points can be constructed accordingly. In order to include these data on the Web page, a small JavaScript fragment requests the resources and updates the corresponding parts of the page[10]. To retrieve a graph of the sensor readings for the last seven days, we include an image on the Web page, whose source is set to

```
http://sensors.example.net/temperature@history
/f:updated last 7 days.jpg?width=400&height=300.
```

The setup of this scenario is depicted in Fig. 3.13.

Run-time For each request of the Web page, five requests are performed to retrieve the required sensor data. An exemplary screenshot of the Web page rendered in a Web browser is depicted in Fig. 3.14.

[10]For this, both the browser and the Web server have to support cross-origin resource sharing (CORS), and the server will have to permit access; otherwise the setup will fail due to the same-origin security policy.

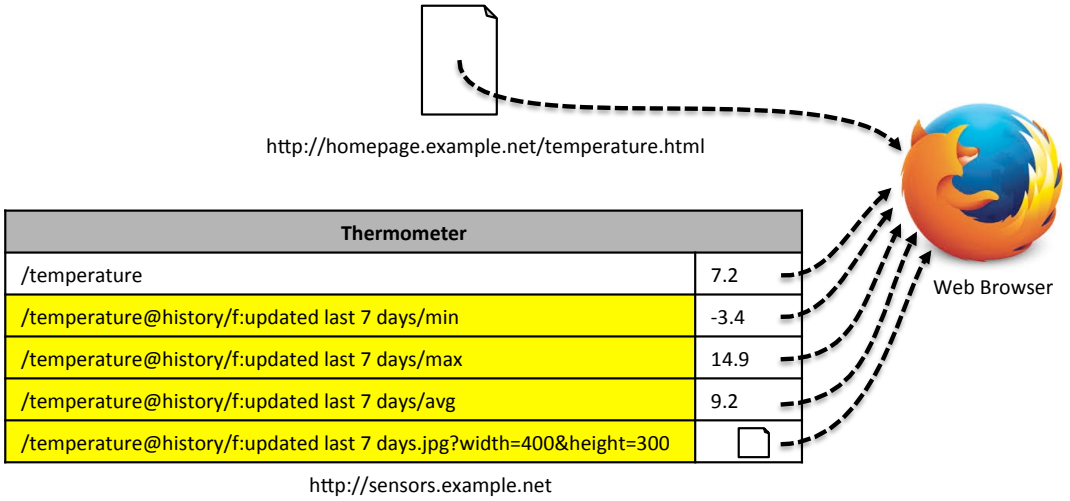


Figure 3.13: Setup of scenario 2 (computed resources are marked yellow).

Discussion The implementation of this scenario with a central hub would require a different setup, in which each reading of the temperature sensor would be sent to the hub. This would create a significant number of HTTP requests just for enabling the use of the sensor data within the hub. The hub would then provide the required functionality such as aggregating data or returning graphical depictions of sensor data. We assume that the interface to the hub that is used for including data on a custom Web page could be optimized. It would require one request to gather all numeric data and a second request to retrieve the graphical depiction of sensor readings.

3.5.1.3 Scenario 3: Turning a Webcam into a Motion Sensor

Scenario The creation of a motion detection system, which is based on a standard webcam. The result of the system is a binary state, returning *true* if motion was detected and *false* otherwise. In that sense, the system provides the same functionality as the PIR sensor introduced in Scenario 1.

Components

- A standard webcam, which publishes its current image at a known URL. In this scenario, it is assumed that the webcam publishes its current image at `http://webcam.example.net/image.jpg`.
- A poller, which creates an image sensor based on the images of the webcam, adding observability and history for its readings. We

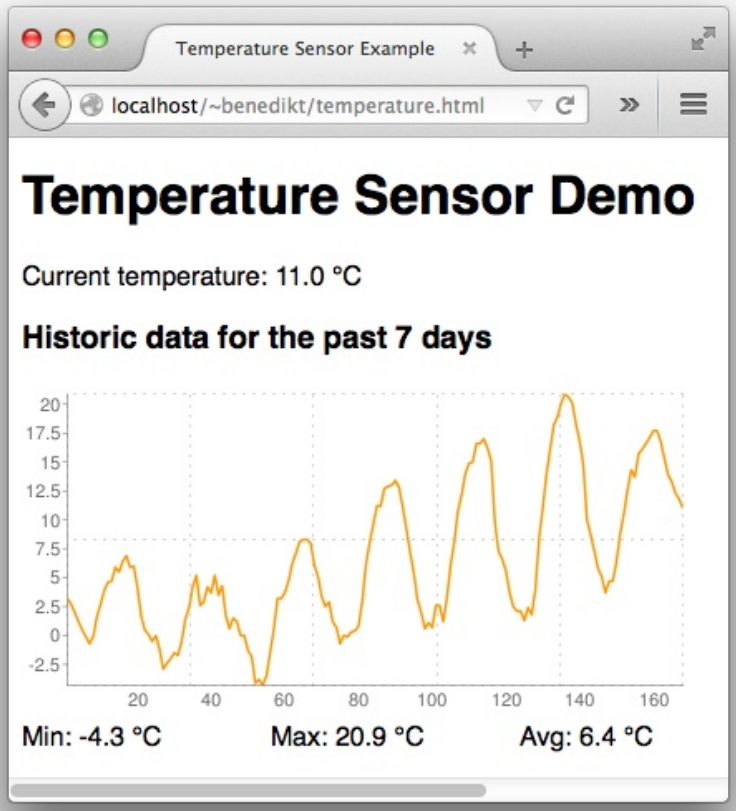


Figure 3.14: Screenshot of Web page with included sensor data.

assume it resides at `http://pollers.example.net/001`.

- The image comparison function introduced in Sect. 3.4.1, which can be accessed at `http://services.example.net/imgcmp`.
- An evaluator, which calls the image comparison function with two subsequent images of the webcam as soon as a new image is present and publishes the result. It can be accessed at `http://evaluators.example.net/001`.

Setup First, we have to configure the poller to periodically poll the image of the webcam. For this, we set its `source` property to

`http://webcam.example.net/image.jpg`

and its `interval` to 0.5. The poller will then download the webcam image every 0.5 seconds and publish the latest image at

`http://pollers.example.net/001/value`.

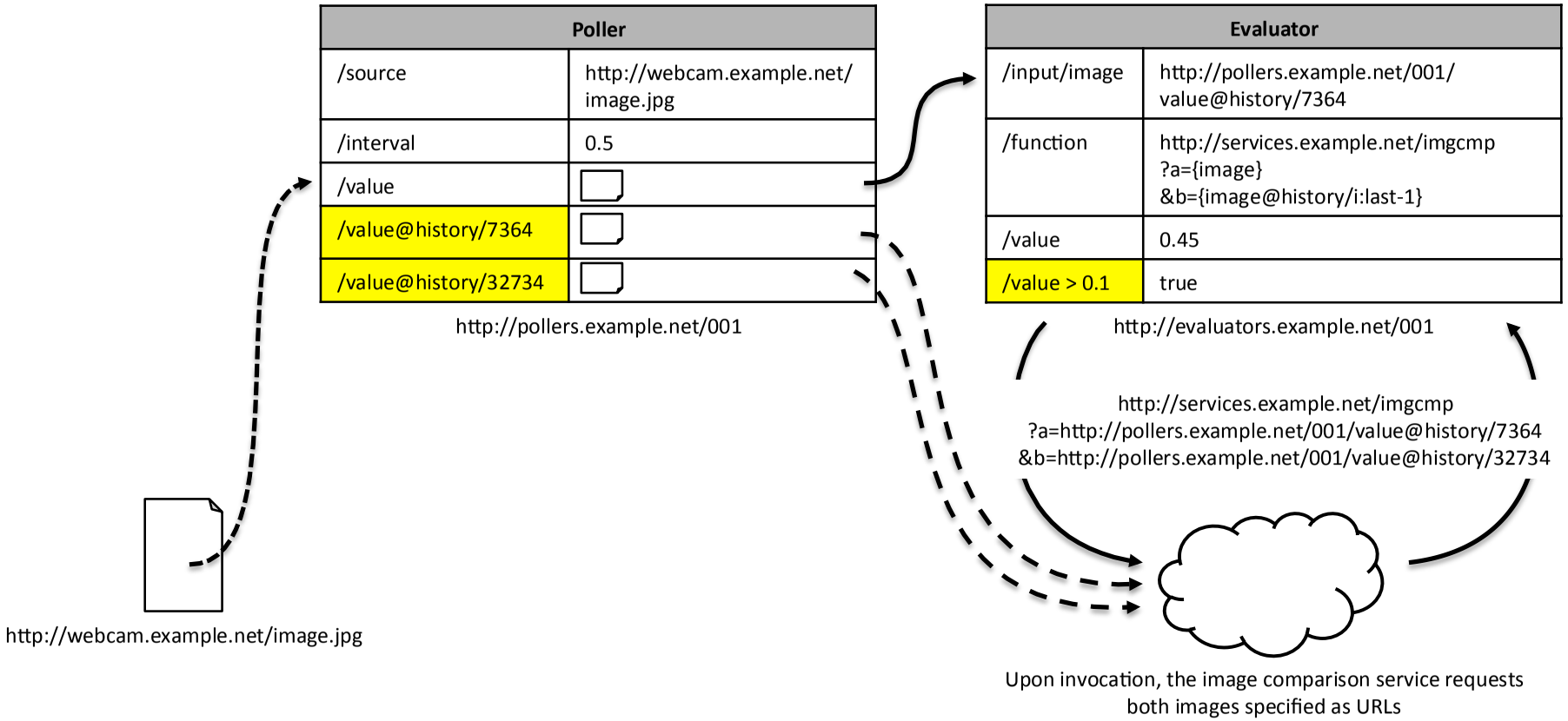


Figure 3.15: A simple application scenario that realizes a motion detection system based on a webcam (computed resources are marked yellow).

Since this is a managed resource, features such as observability and versioning are provided for the *value* property. To configure the evaluator, we need to set its *function* property to a URI template that calls the image comparison service using the URLs of the two latest images of the webcam. This is done by setting

`http://evaluators.example.net/001/function`

to

`http://services.example.net/imgcmp?a={image}`
`&b={image@history/i:last-1}`.

As outlined in Sect. 3.3.10, an evaluator supports versioning of its input resources and the application of versioned resources in the function template. The definition of the function template automatically creates an input property for the evaluator called *image*, which we connect to the output of the poller by registering an observer:

`http://evaluators.example.net/001/input/image`

is added to

`http://pollers.example.net/001/value@observers`.

As soon as the property *image* of the evaluator is updated, the component expands the URI template of the function and performs a `GET` operation on the resulting resource.

Recall that when observing complex resource types like images, notifications do not include serialized data but rather a permanent URL to the corresponding data. Therefore, as soon as the poller is publishing a new image, the evaluator's input property *image* is updated with the URL of the latest image. When this happens, the evaluator will then evaluate the template, by replacing the template variables with the current values. In our example, this might lead to a call like

`http://services.example.net/imgcmp`
`?a=http://pollers.example.net/001/value@history/7364`
`&b=http://pollers.example.net/001/value@history/32734`,

for example (URL encoding omitted for readability). The image comparison service will then request both images from the poller and compute their similarity. The evaluator's *value* property is then updated with the result of the function call, i.e., the similarity value. Note that

this metric reflects an activity measure of the scene the webcam is capturing. Since the *value* property is also versioned, one can access a graph of the activity of the last 30 seconds by requesting

`http://evaluators.example.net/001/value@history`
`/f:updated last 30 secs.png`.

In order to deduce the motion from activity, a simple threshold mapping is applied. As such a threshold is application-specific, the activity graph may help in determining the specific threshold. In our scenario, we select a threshold of 0.1. In order to deduce motion, we append the expression "`>0.1`" to the value resource of the evaluator. This way,

`http://evaluators.example.net/001/value > 0.1`

will return *true* if there is motion detected, and otherwise *false*. The final setup is depicted in Fig. 3.15.

Run-time Every 0.5 seconds, the poller requests the current image of the webcam. This in turn triggers a notification of the evaluator, a call of the image comparison function, and two requests for archived images. This summarizes to five requests.

Discussion The realization of this scenario using a central hub would periodically poll the webcam for images and store all retrieved images within the hub. This behavior is analogue to our poller. All further processing would be performed entirely within the central hub, obviating the need for additional requests. The result would then be accessible via the API of the hub. Per retrieved image of the webcam, the central hub approach therefore requires no additional requests, while our approach requires four additional requests. As two of these four requests contain images of the webcam, the network traffic is significantly higher for our approach, compared to a central hub approach.

3.5.2 Simple Control and Automation Scenarios

3.5.2.1 Scenario 4: Light Control

Scenario This scenario replicates a simple yet common example of a light control, in which a group of lights can be controlled by multiple light switches. Each switch toggles the current state of all lights between *on* and *off*.

Components

- Two or more light switches that output their current switch state as *true* or *false*. It is assumed that these are under the control of our framework and provide their current state at `http://switch1.example.net/state` and `http://switch2.example.net/state`.
- The XOR function introduced in Sect. 3.4.1, which can be accessed at `http://services.example.net/XOR`. Recall that the function requires two or more parameters with textual representations of boolean values.
- An evaluator which encapsulates the XOR function. It can be accessed at `http://evaluators.example.net/004`.
- Three lights that can be switched on and off, named `light1`, `light2` and `light3`. In this scenario, this can be achieved by sending *true* or *false* to `http://light1.example.net/power`, for example.

Setup In order to be able to toggle a light using multiple switches, an intermediary component is required that combines the outputs of the switches. Each time a switch is toggled, the output of the component needs to toggle. This can be achieved by using a boolean XOR operation over all switch states. An XOR operation produces a result of *true* if the number of inputs that read *true* is odd. Each time a switch is toggled, the number of inputs that read true is either increased or decreased by one, thus toggling the result of the XOR operation. In order to include the XOR function, it is wrapped by an evaluator. Each light switch provides one input parameter of the evaluator. For this, we need to specify the property `function` of the evaluator as

`http://services.example.net/XOR?p1={switch1}&p2={switch2}`.

Both switches are then connected to the evaluator by subscribing

`http://evaluators.example.net/004/input/switch1`

to

`http://switch1.example.net/state`

and switch2 accordingly. Finally, all lights that should be toggled by the group of switches are subscribed to the output of the evaluator, the property `value`. This setup is depicted in Fig. 3.16.

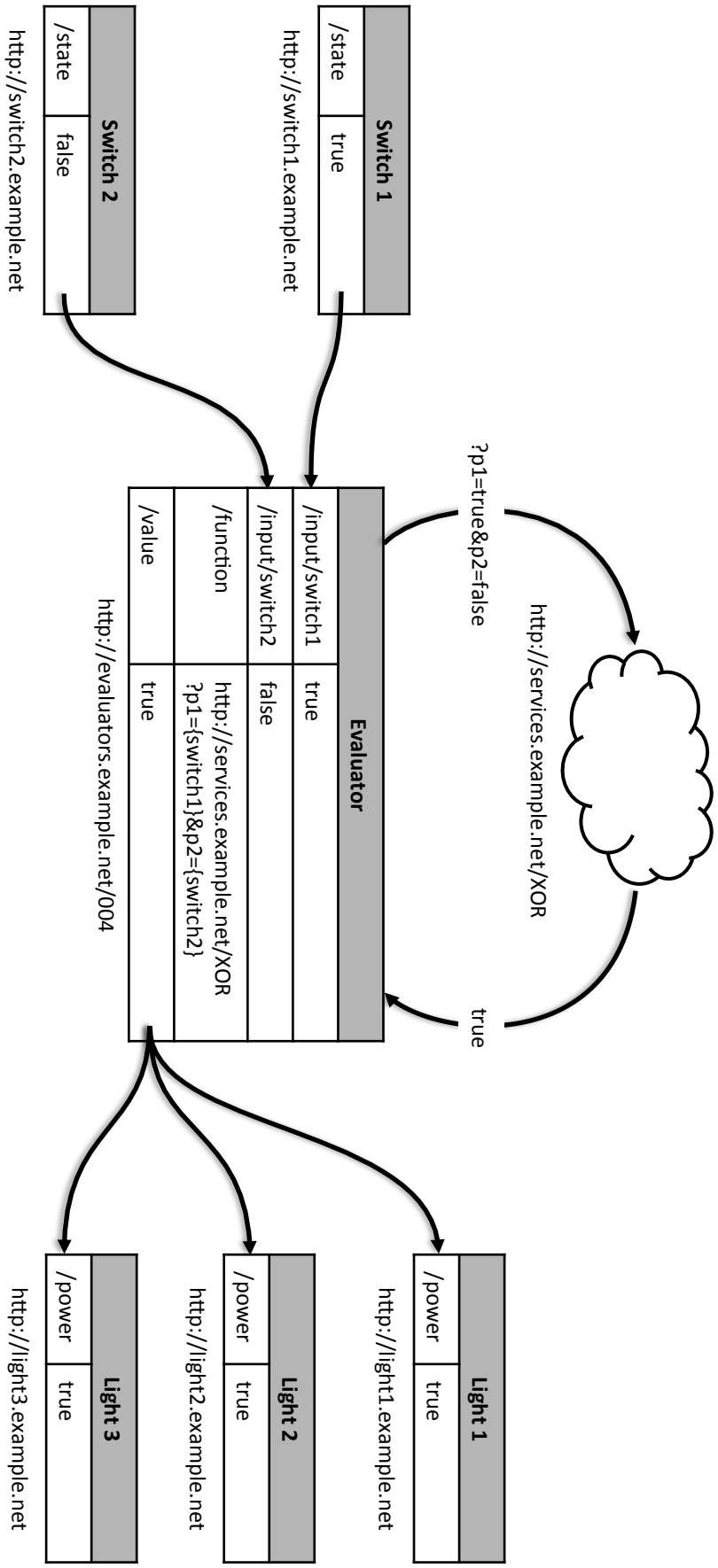


Figure 3.16: Two switches that toggle a group of lights.

Run-time Each press of a button causes a request to the evaluator, a further request to the XOR function, and three requests to update all lights. Thus, a total of five HTTP requests are performed at the press of a button.

Discussion When compared with a central hub, a single HTTP request could be saved per press of a button: The evaluator and the XOR function would be replaced by the hub.

3.5.2.2 Scenario 5: Smart Meeting Room

Scenario In this application scenario, a meeting room is augmented with sensors in order to automate certain tasks: The lights should be switched automatically according to light conditions, provided the room is currently in use. The air conditioning should only work when all windows are closed. Finally, windows that were apparently forgotten to be closed should trigger an alarm.

Components

- A motion sensor (e.g., a PIR sensor), which is a binary sensor that reports *true* if there is currently motion detected, and *false* otherwise. We assume it is managed by our framework and publishes its current state at `http://pir.example.net/motion`.
- A brightness sensor. For reasons of simplicity, we assume it provides values in the range of [0..1] and publishes its current reading at `http://brightness.example.net/value`.
- Three windows, each outfitted with a make contact sensor. We assume that such a sensor reports *true* if the window is closed, *false* otherwise. We assume these sensors are managed by our framework and publish their current readings at `http://window1.example.net/closed` through `window3`.
- Two lights that can be switched on and off, named `light1` and `light2`. In this scenario, this can be achieved by sending *true* or *false* to `http://light1.example.net/power`, for example.
- An air conditioning system which can be turned on and off by sending *true* or *false* to `http://airconditioning.example.net/enabled`.

- An alarm system that can be triggered by writing *true* or *false* to the resource `http://alarm.example.net/on`.
- The AND function introduced in Sect. 3.4.1, which can be accessed at `http://services.example.net/AND`.
- Three evaluators which encapsulate the AND function. The evaluators can be accessed at `http://evaluators.example.net/005`, `http://evaluators.example.net/006`, and `http://evaluators.example.net/007`.

Setup Each of the three functions can be implemented separately.

The automatic lights depend on the motion and brightness sensor. For reasons of convenience, we model the control so that the light switches on quickly and turns off slowly. The method for deducing presence state from motion has already been introduced in Sect. 3.5.1.1 and can be reused here:

```
http://pir.example.net/motion@history/
  f:updated last 5 mins/exists='true'
```

To generate an event as soon as a brightness threshold is undercut but not as soon as it is exceeded, we model it as

```
http://brightness.example.net/value@history/
     f:updated last 5 mins/exists < 0.2
```

This expression results in *true* if the brightness was below 0.2 at least once in the past 5 minutes. The deduced results of the motion and brightness sensors then need to be combined into a single state that reflects the desired state of the lighting. This can be achieved using the evaluator `http://evaluators.example.net/005` with an AND function. Finally, all the lights need to be subscribed to the evaluator's value property.

The automatic activation and deactivation of the room's air conditioning is dependent on the contact sensors of the windows, which detect whether they are open or closed. In order to control the air conditioner, we combine the states of all windows using the evaluator `http://evaluators.example.net/006` with the AND function and subscribe it to all window states. As a result, it will output *false* if at least one window is open. Assuming that the air conditioning can be enabled and disabled at

`http://room.example.net/airconditioning/enabled`,

we subscribe this resource to the output of the evaluator.

In order to detect windows that were left open and automatically issue an alarm, the combined states of all windows and the presence detector are utilized. Since we want to signal open windows only when the room has been empty for 60 minutes, we have to consider the time when the occupancy state of the room was last modified. This can be expressed as

`http://evaluators.example.net/005/`
`input/occupied updated before 60 mins ago`,

which will return *true* if the occupancy state has been steady for more than 60 minutes. The alarm function requires another evaluator with an AND function. The alarm evaluator subscribes to the expression listed above and to the inverted outcomes of the other evaluators. Finally, the Alarm trigger `http://alarm.example.net/on` is subscribed to the outcome of the alarm evaluator. The complete setup is depicted in Fig. 3.17.

Run-time Each state change of a window may generate at most five additional requests: two for evaluating the AND function and controlling the air conditioner. The propagation to the alarm evaluator generates two or three requests, depending on whether the alarm state changes or not. A state change of the brightness sensor generates at most three additional messages: One for evaluating the AND function and two for controlling the lights. A state change of the motion sensor may trigger up to nine additional messages (three for switching the lights, and one or two propagations to the alarm evaluator that each can result in up to three requests).

Discussion A realization of this scenario with a central hub would render all requests to the AND function and between evaluators obsolete, as the desired functionality could be implemented without an additional need for communication. However, the overall number of requests is expected to be significantly higher because occupancy needs to be deduced from raw motion sensor readings inside the central hub. Similarly, all brightness values would be sent to the central hub, and not only changes related to a brightness threshold.

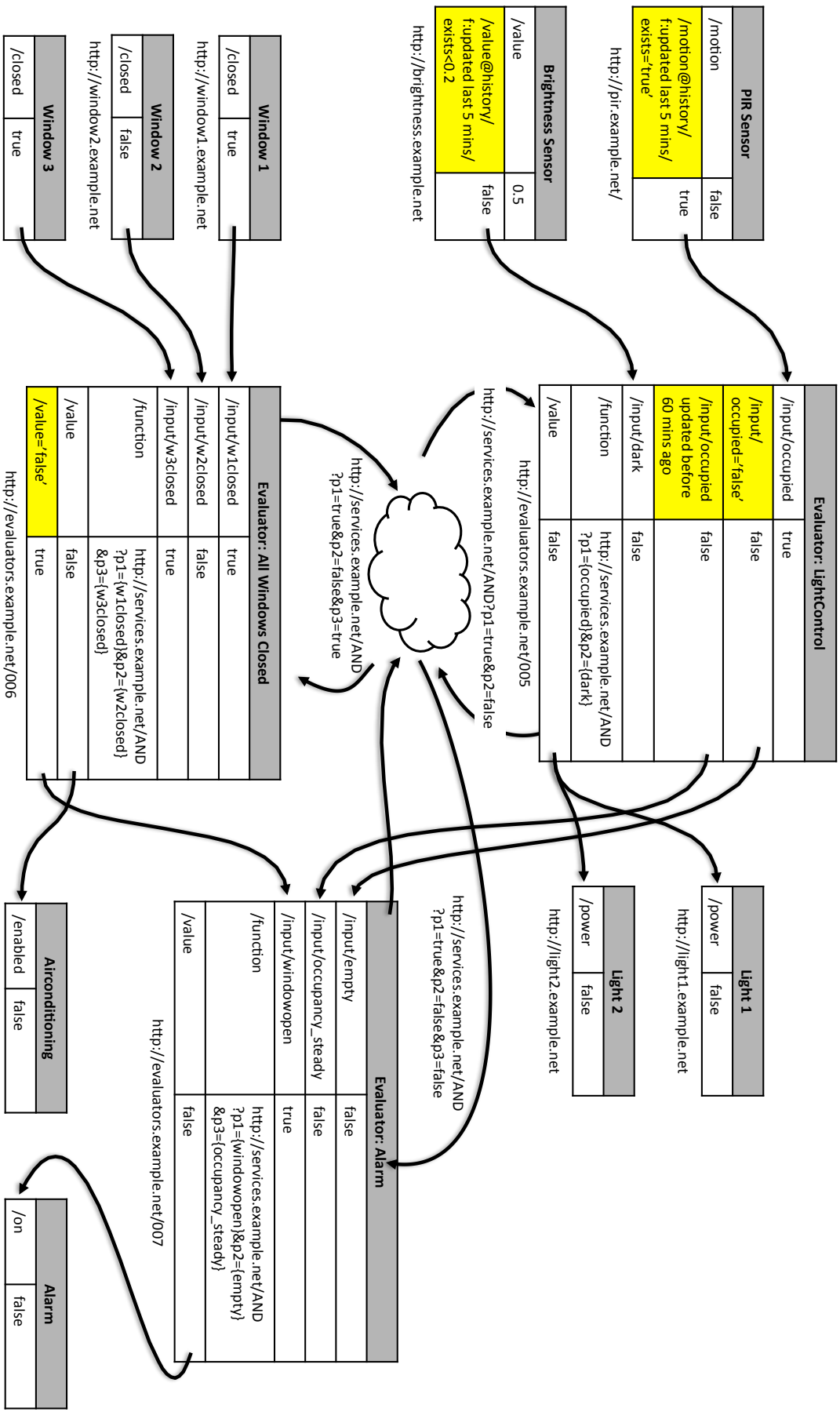


Figure 3.17: Setup of the smart meeting room scenario (computed resources are marked yellow).

3.5.3 Discussion

The evaluation demonstrates that simple application scenarios can be realized based upon the building blocks of our framework. Interestingly, the demonstrated application scenarios show that our approach may actually require fewer HTTP requests during the run-time of an application scenario than a central hub, despite being designed as a decentralized system. This can be attributed to the proposed subscription mechanism, which offers the possibility to subscribe to deduced events instead of to just plain sensor data. A notable exception is scenario 3, in which not only is a higher number of requests performed during run-time, but also the sum of the exchanged payload is significantly increased. This can be attributed to the image comparison method which is stateless and therefore requires both images as parameters per comparison. In contrast, a central hub will keep past sensor states internally and therefore only requires the latest state, i.e., webcam image, to be transferred. A possible solution would be to use HTTP's caching features at functions that receive URLs as parameters, which could significantly reduce the transferred payloads.

3.6 Related Work

Several of the concepts considered in this chapter have already been addressed, which we outline below.

Composition Interconnecting devices and services through HTTP and being able to compose novel services was of early interest [187, 188]. With an increasing number of services available on the Web, this approach gained momentum [85, 86]. Erenkrantz et al. suggest the Computational REST (CREST) architectural style as an extension to REST, which fosters the composition of distributed services [189]. sMAP [190] is a framework that specifically addresses the integration of sensors and actuators in the context of Web applications. In [191], a toolkit for mashups in the Web of Things is introduced.

In [192], a physical mashup editor is introduced that is based on ClickScript[11] and runs in a Web browser. Commercial Web-based editors for composing services for the Web of Things include IFTTT[12],

[11] http://clickscript.ch
[12] http://ifttt.com

Pachube[13] (defunct), and Yahoo Pipes[14]. The UPnP protocol suite[15] simplifies the integration of devices but does not provide dedicated support for composition [193].

Eventing and Publish/Subscribe Today, subscriptions to updates from the Web are usually realized by periodically polling a Web feed containing a list of relevant updates, which are provided in the Atom [167] or RSS [168] syndication formats. Atom also offers publishing capabilities through its publishing protocol [194].

The nature of HTTP's request/response paradigm aggravates the use of real-time notifications to be sent to Web clients. Traditionally, clients usually resorted to polling by periodically requesting a monitored resource in order to check for updates. HTTP provides caching features that allow this check to be performed at the server using the conditional `GET` method [64].

Several approaches have been suggested for realizing real-time notifications in the context of the Web. Netscape included support for server push technologies in its Browser Netscape Navigator 1.1 back in 1995 [195], but it did not catch on. Microsoft suggested GENA for eventing in the Internet in 1998 [196], but it prevailed only within UPnP [193]. Several years later, approaches denoted as Comet (long polling or streaming) [197], Server Sent Events [198], and the WebSocket Protocol [199] emerged, addressing the need for real-time data on the Web 2.0. HTTP callbacks, sometimes called WebHooks [184], have been suggested for asynchronous data exchange between Web sites [200].

A Publish/Subscribe system for the Web is realized by Google's PubSubHubbub, based on Atom and HTTP callbacks [185]. For CoAP [69], there is a suggested extension that enables the monitoring of resources [71], which is quite similar to our approach. Several of the mentioned approaches have already been adapted in the context of the Web of Things [201, 202, 203].

Simple Queries SensorBase was an early Web-based platform for sensor readings [204, 205]. Data could be queried using SQL statements in a Web front end. Subsequent platforms for the "Sensor Web" also provide programmatic methods of filtering sensor data [201]. Stream

[13] `http://www.pachube.com` (superseded by `https://xively.com`)
[14] `http://pipes.yahoo.com/pipes`
[15] `http://www.upnp.org`

feeds [206] are suggested as a Sensor Web abstraction and provide a URL-based interface for filtering sensor readings. In [187], a URL-based naming scheme for control of devices in home automation scenarios is suggested. FIQL, a simple query language intended for filtering Web feeds, is proposed in [207]. SPARQL has been proposed for searching in the context of a semantic Web of Things [89].

The combination of queries and subscriptions is also briefly mentioned in the current draft for observing resources with CoAP [71].

Versioning of Resources HTTP features only limited support for resource versioning[16], which is mostly used to support caching [64]. Versioning of HTTP resources in the context of document management is addressed by WebDAV [181, 208] and CMIS [209, 210]. Support for temporal data based on a time series of sensor readings is an essential feature in platforms for sensor data, where it is addressed to varying extents [171, 205, 201]. As an application interface, Web feeds have been suggested to represent sensor data streams [206].

3.7 Summary

In this chapter, we presented a prototypical framework for the Web of Things, which strives to improve the current Web architecture with unified concepts for interacting with the physical world. It is not based on a central hub, but rather inherently distributed, consisting of several building blocks that can be combined to realize specific application scenarios. The focus is put on two classes of application scenarios: data syndication and processing scenarios, in which data are accumulated and processed from possibly multiple sources, and simple control and automation scenarios, in which aspects of the physical world are being manipulated. The framework specifically addresses some of the key features that are often required when interacting with sensors and actuators:

- The historization of data, such as past sensor readings as well as actuator values.
- The support for notifications triggered by certain events.
- The support for simple queries.

[16] `ETag` and `Last-Modified` response headers.

- The ability to compose and run distributed application scenarios.

The execution of applications scenarios is based on an event-driven, distributed control flow that may span multiple components under different authoritative domains. We evaluated the concepts introduced in several exemplary application scenarios and demonstrated that our inherently distributed approach does not necessarily lead to a communication overhead when compared with solutions that are based on a central hub.

4 Extending the Web Down to Constrained Wireless Devices

In this thesis, we assume that the Web will serve as a universal interface to the physical world. Until now, we have not addressed how the sensors and actuators of real-world entities eventually surface on the Web. The usual approach is to use dedicated low-power communication hardware and protocols over which resource-constrained devices communicate with an application-specific gateway. The gateway then provides an HTTP interface and handles all requests from and to the Web. While this approach is optimized for energy consumption, it impedes the interoperability of arbitrary entities, as such gateways often have to be aware of application-specific protocol mappings.

In this chapter, we pursue a different approach – the direct communication with physical entities over the HTTP protocol, leveraging existing infrastructure and standards. We investigate whether it is technologically feasible for the resource-intensive protocols and data formats used for today's Web (such as TCP, HTTP, and JSON) to be used to communicate with battery-powered wireless devices that are subject to resource constraints. We argue that "connecting" the physical world directly to the Web, thereby avoiding additional infrastructure such as gateways, reduces complexity, enables mobility, simplifies interoperability and therefore fosters a wide and rapid deployment of a Web of Things.

This approach has become possible due to the improvements of available hardware. So-called ultra-low-power IEEE 802.11 transceivers are claimed to achieve an operating time of years on batteries, for event-based interaction, while working with the existing Wi-Fi infrastructure. In this chapter, we use programmable, ultra-low-power Wi-Fi modules that are commercially available in order to evaluate the direct interoperability of physical entities with the Web.

Parts of this chapter have been published in [211].

4.1 Background

In this section, we introduce the basic components, communication patterns, and standards that are relevant for connecting battery-powered wireless devices, which are subject to resource constraints, to the Web.

4.1.1 Embedded Web Server

With the advent of the Web, it became popular to provide a built-in HTTP server in networked devices in order to provide a platform-independent and convenient way to configure and monitor such products. Users were no longer forced to install device-specific configuration software on their PCs but could instead use a Web browser, a standard tool available on almost any platform, to configure, monitor, and control such devices. The user interface consists of standard Web pages. As a side effect, this enabled global access to such devices, at least in theory. In practice, such devices were usually assigned a private IP address and placed behind a NAT[1]. However, the provided interface could also be used by applications; for example, by sending an HTML form using a script instead of a Web browser [83]. Early examples include wireless routers, webcams, and power outlets that could be switched using a Web interface. In a parallel development, the need for standardized inter-device communication for consumer products was addressed by the UPnP Forum[2], whose standards are based on Web standards such as HTTP, SOAP, and XML. Today, many networked consumer products feature both interfaces: HTML pages [94, 95] as a user interface and UPnP [193] as an application interface. However, such devices are usually mains powered and therefore feature sufficient resources for handling HTTP requests.

4.1.2 Battery-powered Wireless Devices

The class of *battery-powered wireless devices (BPWDs)* is of particular interest, as it forms the basis for the realization of the vision of *ubiquitous computing* [5]: As such devices require no cabling, deployment is cheap and simple, and devices can also be mobile. Given a small form factor and sufficient battery life, such devices may be eventually

[1] In this context, a router that performs network address translation (NAT) from possibly many private IP addresses to a single public IP address and vice versa.

[2] http://www.upnp.org/

embedded in everyday objects such as milk cartons, augmenting them with sensing, computation, and communication capabilities in order to make them "smart" [5, 212]. The crucial factor of such devices is their *energy efficiency*: Since replacing batteries may be impossible (e.g., a device sealed in a milk carton), too costly (e.g., a sensor network deployed on a glacier), or simply too cumbersome (e.g., a plurality of devices at home), energy is a scarce resource and low power consumption is therefore of absolute importance. Increasing the capacity of the batteries (and in turn, their physical size) may also not be a viable option, as physical space or weight may be limited due to application constraints. Note that using rechargeable batteries may mitigate this criterion, but their use is often not practical.

In consequence of this scarceness of energy, battery-powered wireless devices are usually *resource-constrained devices*. Components such as CPU, RAM, and the wireless transceiver are all optimized for energy efficiency and therefore feature only limited resources. This is why applications and communication protocols on these platforms are usually carefully optimized.

4.1.3 Connecting Devices to the Web

Connecting devices to the Internet and in turn to the Web allows not only control of the devices itself but also monitoring and control of certain phenomena of the physical world through their sensors and actuators, often in real time. Additionally, such devices are able to utilize the large and growing amount of data and services available on the Web. The "connection to the Web" therefore covers two directions of communication:

- *Web Client → Device:* There is an HTTP-based interface of the device available on the Internet, such as a Web page or a REST API. The interface could be provided by an intermediary service that was in turn connected to the device or by a Web server running on the device itself. This concept makes real-time interaction with the device and, if applicable, its physical environment possible. In this scenario, the device provides its capabilities over the Web, which can be used by third parties for various applications.

- *Device → Web Server:* The device accesses or updates data or utilizes services that are provided on the Web. Access could either

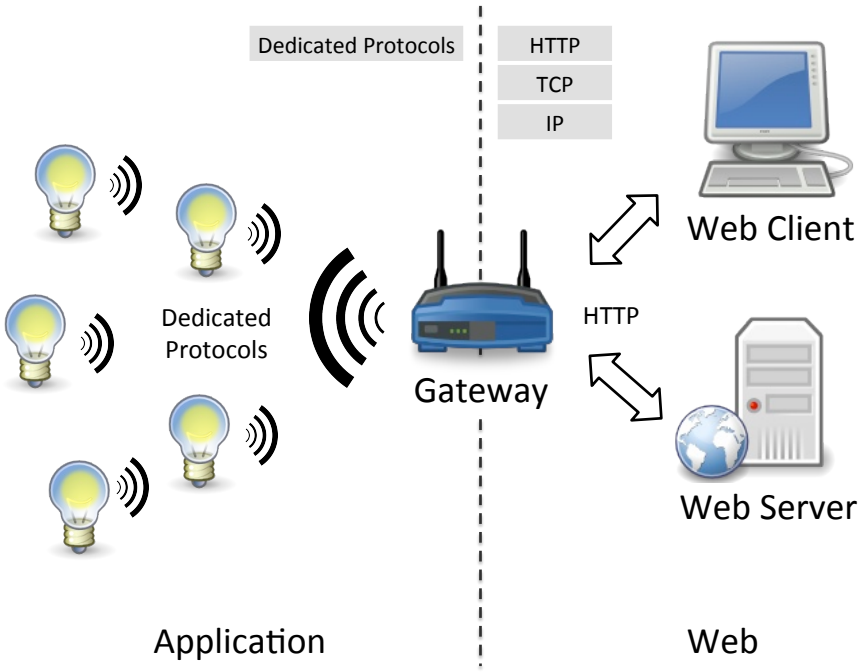


Figure 4.1: Application-specific gateways (icons provided by [213]).

be direct, by using a Web client on the device itself, or be provided by an intermediary service. This concept enables devices not only to leverage the resources available on the Web, but also to report data to an (already existing) Web service, possibly in real time. In this scenario, the device uses the Web as a platform in order to provide its functionality.

Note that these are two separate aspects that can be implemented independently. We say that a device is "connected to the Web" if it satisfies at least one of these two concepts. Implementations of connecting devices to the Web can be categorized into three different categories, which we will discuss below.

4.1.3.1 Application-Specific Gateways

In this approach, which is is illustrated in Fig. 4.1, there is a dedicated application-specific gateway that mediates between the device(s) and the Web. The gateway can provide access to data as well as functionality of its associated devices from the Web. It can also add functionality, such as by providing a history of past sensor readings. While application-specific gateways are usually utilized for this direction of communication, they can also offer functionality that is provided by services and data on the Web to their associated devices. Devices are

not required to address any Web-specific issues; instead they are provided all required application functionality that depends on the Web by the gateway.

This is a classic approach that has been utilized early on. For example, early works in the field of ubiquitous computing used infrared communication for connecting devices, as it is a cheap and low-power technology [7, 8, 214]. Wireless sensor networks (WSNs) use dedicated low-power radio interfaces and application-specific protocols to form a local multi-hop network that is used to send collected data to a dedicated sink node, which is in turn connected to the Internet. An application-specific gateway may then provide access to the collected data on the Web [15, 20]. Other examples include small sensor devices connecting to a smartphone using Bluetooth LE, which then uploads collected data to a Web service [215], and energy-harvesting devices based on the EnOcean technology that are connected to the Web through dedicated gateways [216].

The advantage of this approach is that the strict separation of the device and the Web enables optimizations on both sides. Devices can use dedicated and application-specific communication protocols and radio modules to create a more efficient implementation. Aspects of communication that require interaction with Web resources can be outsourced to the gateway. The gateway software can be run on a high-performance computer that is able to handle high loads from the Web while at the same time preventing devices from overload or malicious access.

However, using this approach, devices and gateways are tightly coupled: Changes of functionality on the devices usually also require changes on the gateway. This introduces complexity and also complicates further developments, as such gateways may be operated by a different party than their connected devices are. Mobility of devices is also impeded, as this requires the deployment of dedicated gateways.

In summary, this approach enables both sides to be optimized (e.g., the application side can be optimized for energy efficiency and the Web side for performance) and is well suited for static application scenarios that do not change functionality or require mobility. The gateway is an inherent part of the application scenario and includes application-specific code.

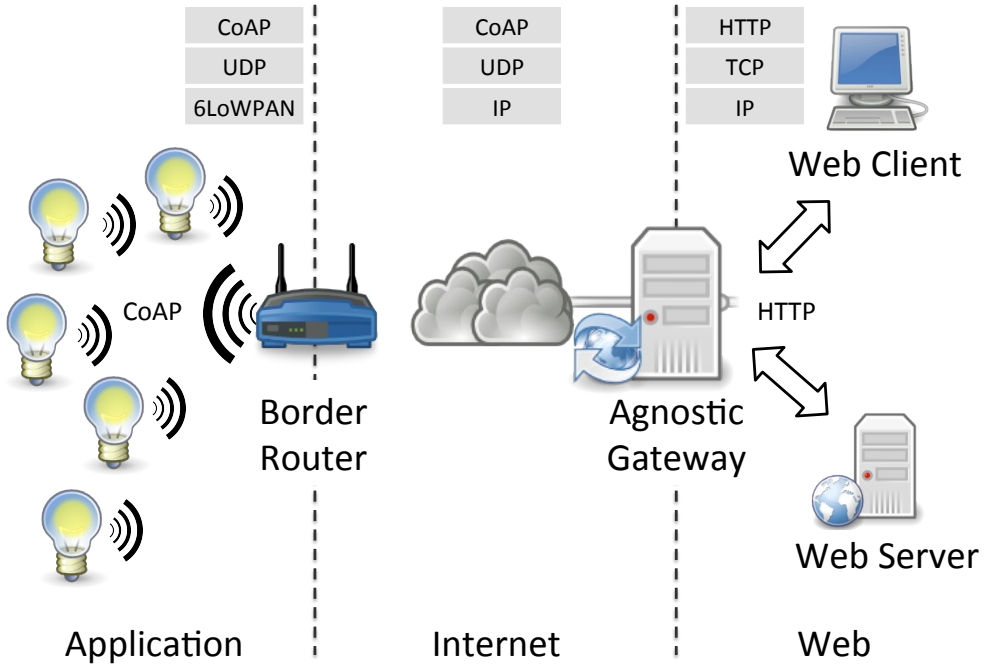


Figure 4.2: Application-agnostic gateways (icons provided by [213]).

4.1.3.2 Application-Agnostic Gateways

The approach using application-agnostic gateways in order to connect devices to the Web is depicted in Fig. 4.2. In this approach, there is also a gateway that mediates between the device and the Web. However, the gateway is *application-agnostic*, so it does not "know" about the specifics of the application scenario and can therefore be generic, i.e., it does not include any application-specific code. Requests from the devices to the Web are automatically translated, as well as the corresponding responses. In a similar way, the gateway provides an automatic translation for requests originating from the Web to devices. This approach requires protocols that are compatible to the HTTP on the device side, as well as a standardized mapping scheme for the gateway.

An example of this approach is CoAP [69], which is designed to run on IEEE 802.15.4 low-power networks and follows the REST architectural style [65], just like HTTP. It is a binary protocol that is more bandwidth-efficient than the verbose and text-based HTTP. It is intended to be compatible to the Web by using application-agnostic gateways [217]. This concept is discussed in detail in Sect. 4.6.5.

Using application-agnostic gateways allows devices to use protocols and radio interfaces that are optimized for low energy consumption

while still providing an interface to the Web. Using generic and possibly public gateways simplifies interoperability. Just like for application-specific gateways, an application-agnostic gateway can run on a dedicated computer that protects devices from overload or malicious access. In contrast, changes in application functionality only need to be implemented on the devices.

However, the use of existing Web resources can be impeded by the required payloads, which are usually formatted in XML or JSON and can amount to significant sizes. A possible solution would require the gateway to automatically translate XML and JSON data to serializations that are more space-efficient and simpler to parse [218, 219]. Mobility of nodes is hampered, since dedicated access points (such as border routers) are required to provide Internet access. Another drawback is that the gateway is still a crucial part in the communication chain between devices and the Web.

To summarize, this concept allows a simple connection of devices to the Web while still offering the possibility to use dedicated low-power radio interfaces on the devices. However, the use of existing Web resources can be aggravated by the limitations of the low-power radio interface utilized on the devices, such as its supported packet size[3].

4.1.3.3 Direct Connection to the Web

The approach of connecting devices directly to the Web terminates HTTP connections only at communication endpoints. Devices run a Web server and/or a Web client to directly provide their functionality to the Web and/or access the Web's resources. This approach is depicted in Fig. 4.3.

Examples of this approach include networked devices such as routers, which provide a Web interface for configuration and monitoring. Interestingly, many routers can also use some services available on the Web, such as DynDNS[4].

Using unmediated communication between the device and servers and clients on the Web enables a high degree of flexibility. Changes in application functionality need only be implemented on the devices.

[3]Maximum transfer unit (MTU).

[4]This service provides a fixed DNS entry for devices that are assigned a dynamic IP address. It provides an HTTP interface for specifying the device's current IP address. Many routers can be configured to automatically register their current IP address using this interface.

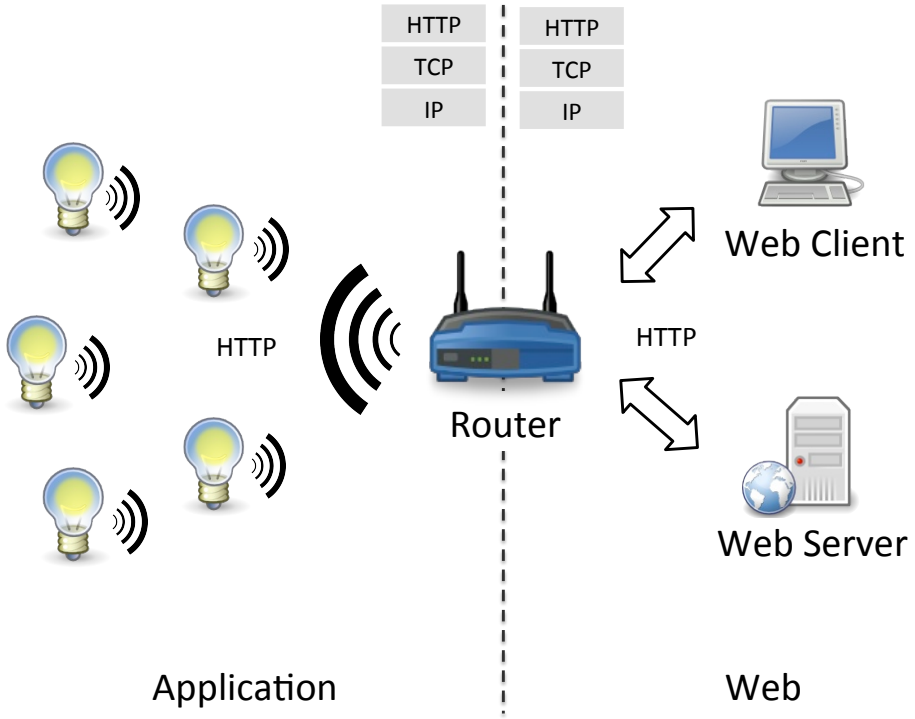


Figure 4.3: Direct connection to the Web (icons provided by [213]).

This approach also simplifies interoperability as the same communication protocol can be used for accessing local as well as remote devices and services. It also simplifies configuration and monitoring, as this is possible by simply using a Web browser.

However, making a device directly accessible on the Web also introduces load and security issues: Such a device can be easily overwhelmed by requests from the Web, and its Web server is constantly exposed to possible attacks. This issue can be addressed by using a reverse proxy, a standard component of the Web architecture that relays requests between the device and the Web. Note that when using IPv4, devices are usually placed behind a NAT and are therefore not publicly accessible, but they can communicate with the Web.

Connecting devices directly to the Web optimizes them for flexible interoperability with other devices and services on the Web, at the cost of reduced energy efficiency.

4.1.4 Towards Unmediated Interoperability

Historically, the use of application-specific gateways in order to connect BPWDs to the Web was well founded, for several reasons:

- Using dedicated communication links and application-specific communication protocols to connect BPWDs enabled a high degree of energy efficiency. Hardware and wireless protocols that could handle the Web architecture in an energy-efficient way were not available.
- The connection to the Web was usually limited to providing data acquired by the devices on HTML pages.
- Devices did rarely depend on data or services available on the Web. If so, they could be handled by a gateway.
- Application scenarios were usually static.

With the advent of an Internet of Things, the ability of devices to interact with each other and also with existing services and applications is becoming an important criterion. Deployed devices are no longer limited to a single application scenario, and application scenarios are no longer limited to devices under a single authoritative domain. For this reason, *interoperability* is becoming increasingly important.

This implies a *paradigm shift* on the nodes, away from locally optimized application-specific communication protocols and towards standardized and generic communication protocols that simplify interoperability. This enables the abandonment of application-specific gateways, which in turn reduces complexity, enables mobility and simplifies interoperability. Since the Internet is currently the largest communication network and the Web can be considered the largest distributed platform, it seems reasonable to leverage this infrastructure.

4.1.5 Using Web Standards on Battery-powered Wireless Devices

The protocol stack used for today's Web is HTTP [64] over TCP [220] over IP[5] [57, 59]. IP abstracts from the underlying link technology and protocols and enables global addressability and connectivity, providing a "best effort" end-to-end packet delivery[6]. While most devices are still running IPv4 [57], with the exhaustion of its 32-bit address space, there is pressure to switch to IPv6 [59], which provides an address space of 128 bits[7]. TCP handles packet retransmissions and provides a reliable,

[5] Note that there may be an additional security layer such as TLS/SSL [221].
[6] *Best effort* is an euphemism for not providing any means of retransmitting lost packets.
[7] This allows for 340,282,366,920,938,463,463,374,607,431,768,211,456 different addresses.

bi-directional end-to-end data stream with congestion and flow control. HTTP handles issues of the application layer such as addressing resources, caching data, and providing transparent compression. The payload of the applications is usually serialized in the XML [96] or JSON [97] formats. Additionally, DNS [222, 223] provides a global, distributed, hierarchical naming service that is realized as a somewhat independent service based on UDP [224][8].

Using the layered TCP/IP architecture in combination with the verbose HTTP on resource-constrained devices such as BPWDs has not been very popular in the past, for a variety of reasons, such as:

- The application scenario required only local communication.
- The high communication overhead of HTTP over TCP/IP was problematic on the given (slow) communication links, such as infrared.
- The communication primitives required by the application scenario were missing and could only be realized inefficiently (e.g., application-level broadcast).
- The application scenario could not benefit from features of HTTP such as caching support or content negotiation.
- The degraded performance of TCP on lossy wireless links [225].

However, in the academic community, there was an early interest in connecting also very small devices to the Internet and to the Web (e.g., [55]).

4.1.6 Using Ultra-low-power Wi-Fi for Battery-powered Wireless Devices

Hardware improvements made ultra-low-power IEEE 802.11 modules possible (e.g., [226, 227]), which are claimed to feature a multi-year battery life when used in certain application scenarios [226]. An advantage of this approach is that we can *leverage existing infrastructures*: IEEE 802.11 access points are truly ubiquitous in urban areas, and a growing number of access points feature public Internet access

[8] However, HTTP benefits from the use of domain names.

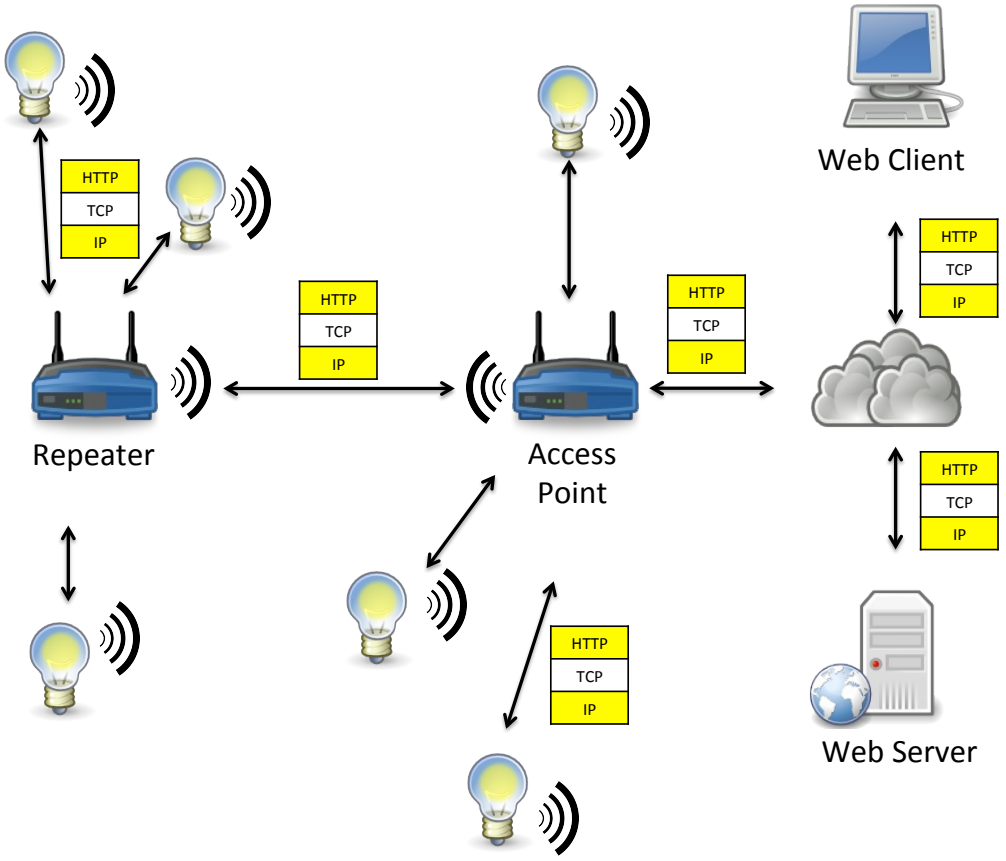


Figure 4.4: Direct interoperability of Wi-Fi nodes with the Web. Since IEEE 802.11 does not support multi-hop forwarding in infrastructure mode, repeaters or additional access points have to be used to increase the area of connectivity (icons provided by [213]).

(e.g., at airports, in parks, etc.). There are also approaches that allow individuals to share their access points[9], which is currently hampered due to legal regulations. There is also Wi-Fi roaming such as eduroam[10], which provides international Wi-Fi roaming for members of participating universities. An interesting aspect of the pervasiveness of Wi-Fi access points is that they can now be leveraged to perform localization, both outdoor as well as indoor [228, 229, 230]. Using IP allows us to automatically configure the device using DHCP, which in turn enables mobile nodes.

However, instead of using application-specific protocols based on UDP, as is suggested by the manufacturers, we are interested in leveraging the potential of a direct connection to the Web. For this, we run an HTTP client and HTTP server directly on the module. Running HTTP over TCP on the node potentially allows us to directly inter-

[9] e.g., http://corp.fon.com/
[10] http://www.eduroam.org/

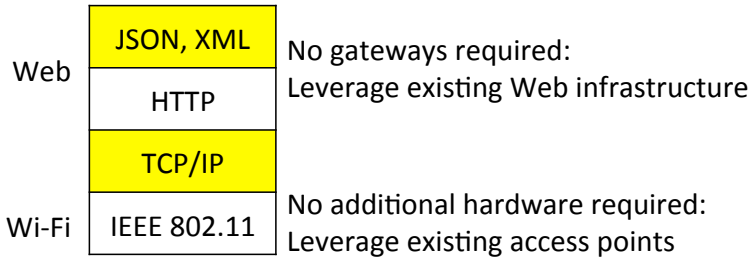


Figure 4.5: Leveraging existing infrastructures.

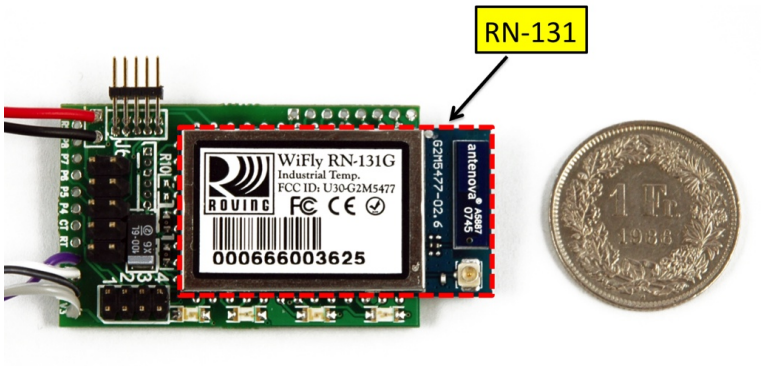


Figure 4.6: The Roving RN-134 evaluation board (green circuit board), which is built around an RN-131. The RN-134 was used for all of the experiments in this chapter. The depicted coin is 1 Swiss franc, which is comparable to that of a US quarter or 1 Euro coin in size.

act with the plethora of existing Web services, Web-enabled devices, and HTTP clients such as Web browsers. This concept is illustrated in Fig. 4.5.

4.2 Platform Utilized

Since IEEE 802.11 was not primarily designed for low-power operation, so-called "ultra-low-power" (ULP) Wi-Fi modules introduce certain features that enable long-lasting operation on batteries. The work in this chapter is based on the RN-131 [226], a low-power programmable Wi-Fi module available from Microchip[11]. Ultra-low-power operation is achieved by supporting short uptimes (due to fast network access and high bandwidth) and a sleep mode that requires little energy yet still supports certain operations, such as real-time monitoring of specific events.

The RN-131 was initially developed by G2 Microsystems under the name G2M5477 [231], before the technology was acquired by Roving

[11] http://www.microchip.com

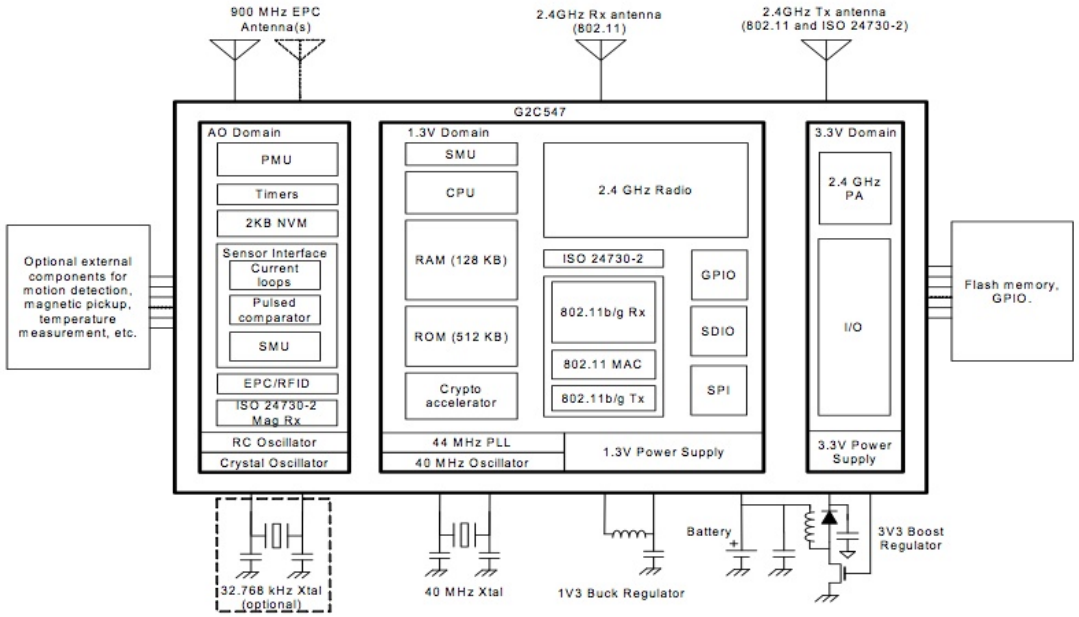


Figure 4.7: Block diagram of the G2C547, a low-power programmable IEEE 802.11 SoC, which is the core of the RN-131 module (Source: [233]).

Networks and in turn by Microchip. The module is a proprietary low-power platform that combines several features (sensor support, RFID, localization, WLAN) from different application domains. We provide a brief overview of the platform in this section, which is based on publicly available information[12] [226, 231, 232, 233, 234, 235].

The heart of the RN-131 is the G2C547 [232, 233], an SoC that features a 44 MHz 32-bit RISC processor, 128 kB of RAM, 2 kB of battery-backed memory, and 512 KB of ROM[13]. It includes an IEEE 802.11b/g transceiver, which handles connection rates of up to 54 Mbit/s and also supports standard authentication methods such as WEP, WPA-PSK, and WPA2-PSK. The SoC also includes a crypto accelerator, which supports security algorithms required for authentication over IEEE 802.11, and a real-time clock. Interfacing with external components is possible over UART, SPI master/slave, and SDIO. There are also up to 15 GPIO pins and 8 analog sensor interfaces. The SoC also supports a UHF (868/902-928 MHz) EPC Gen 2 RFID [236] interface, which can be used to turn the G2C547 into a smart RFID tag. Additionally, real-time location tracking according to ISO 24730-2 [237] is supported by providing a location transmitter at 2.4 GHz and a 125 kHz magnetic

[12]More detailed information is provided in the support documents of the manufacturer, which require the signing of a non-disclosure agreement.

[13]There is a discrepancy between the G2C547 product brief (512 KB ROM) and Roving RN-131 data sheet (2 MB ROM).

receiver.

The architecture of the G2C547 is depicted in Fig. 4.7. Note that there are three power domains: The AO domain is always powered (as long as there is power supply) and contains components that require continuous operation such as the real-time clock or the sensor interface. Based on events generated in this domain, like the expiration of a timer, the other domains are powered up. The AO domain requires only very little power to operate ([233] lists 5 μW). The 1.3 V domain contains most of the functionality of the SoC, such as the CPU, RAM, ROM and supporting sub-systems. Finally, the 3.3 V domain handles I/O and contains the 2.4 GHz power amplifier. The 1.3 V and the 3.3 V domains are only powered up when required, so when the chip is in sleep mode (only AO domain active), the CPU and Wi-Fi interface are not available.

The RN-131 module adds 8 Mbit of flash storage, a small ceramic chip antenna, and a connector for an external antenna. It has a small footprint (20x38x4 mm), exports most of the pins of the G2C547, and requires only a few external components to run. According to the data sheet, the RN-131 can be powered between 2.0 and 3.7 V and drains $15 - 212$ mA of current when in active mode, depending on the specific operation [226]. When in sleep mode, however, the module has a nominal current consumption of only 4 μA, which is comparable to the consumption in sleep mode of mote-like devices [238].

The RN-131 can be operated in two different ways: It can connect to a host CPU and operate as a client, handling all the network-related operations. In this scenario, the user's application runs on the host CPU and communicates with the RN-131 via a serial connection. The other possibility is to run the user's application directly on the RN-131. In order to use this option, one has to develop the applications using a proprietary software development kit provided by Roving Networks. The RN-131 runs eCos[14], an embedded real-time operating system that supports multi-threading. It utilizes the lwIP TCP/IP stack [55] for communication.

For our work, we used the RN-134 evaluation board [239], which integrates the RN-131 module and additional components, such as LEDs and a TTL-to-RS232 level converter (Fig. 4.6).

[14] http://ecos.sourceware.org

4.3 Approach

Our approach is based on the RN-131 Wi-Fi module, which was introduced in the previous section. In order to interface with the outside world, we run an HTTP server and an HTTP client directly on the module, which can communicate over the Wi-Fi link. As energy consumption is significant when the module is awake, it is crucial to keep the node in sleep mode as long as possible when running on batteries.

4.3.1 Web Interface

We rely on the REST paradigm [65] to expose physical entities, including their sensors and actuators, as resources on the Web. Compared to SOAP [63], REST utilizes HTTP as an application protocol rather than as a transport protocol. For this reason, it introduces less overhead and can also directly benefit from HTTP's features such as cache control and content negotiation. Additionally, the resource-centered paradigm of REST maps well to physical resources.

In our approach, every node can run a built-in Web server exposing sensor, actuator, and configuration data as Web resources. These are identified by self-descriptive URLs and can be accessed with HTTP using standard operations such as `GET` and `POST`[15]. For example, in order to read a current sensor value, an HTTP `GET` request is sent to the resource of the sensor. The response includes a textual representation[16] of the current sensor value. This concept works similarly for actuators and configuration variables exposed as resources, except that we can also update them by sending an HTTP `POST` request with the new desired value to their URLs. There is also an HTML front end for human interaction.

Besides the HTTP server, each node runs an HTTP client that is used for communicating events. As soon as a preconfigured event occurs, such as the change of a sensor value, the node sends an HTTP `POST` request to a prespecified URL, containing relevant data (e.g., the updated sensor reading) in the message body. Since this approach is analogous to the callback mechanism used in some programming languages, this concept is sometimes called an HTTP callback or a webhook. The concept of such HTTP callbacks has long been known but

[15] To increase interoperability, we currently resort to `POST` instead of using the more appropriate `PUT`.

[16] Note that this is not restricted to a certain representation.

recently gained popularity [184, 185, 200].

Using HTTP callbacks to communicate events has the advantage that the node may sleep until an event occurs, wake up only to perform the HTTP request, and go back to sleep again, thus saving precious energy. An additional advantage of this approach is that events can be communicated with little or no noticeable delay. Finally, the node may receive valuable information in the HTTP response of the callback, as we will see later.

4.3.2 Sensing

The discussion in this section addresses battery-powered nodes, as nodes that feature an external power supply are not limited in their energy consumption. In order to enable long-running operations when the node is powered with batteries, it is crucial to limit the time it spends awake.

We argue that users will most likely be interested in *event-based sensing* scenarios. In such scenarios, users do not want to monitor raw sensor values but are rather interested in detecting higher-level events, which may then be used to trigger a specific action. Examples of events include the opening of a door, the sudden occupancy of a room, or the crossing of a temperature threshold. An important aspect of event-based sensing is that detected events should be communicated in real time, in order to be able to realize application scenarios that require immediate action upon a given event. For example, a light switch, which is essentially a binary sensor, may be associated with a lamp. Each time the switch is used, it generates an event which is sent to the lamp. In this home automation scenario, users will expect the conventional behavior, which is the immediate control of the lamp with the switch. In order to enable real-time or near-real-time applications, it is crucial that the node transmits the notification of a detected event with minimal delay. As detailed in Sect. 4.2, the platform used in this chapter supports the monitoring for certain sensor events while the module is in sleep mode. Thus it is possible to wake up the node and send a notification only when a prespecified event has been detected. As a side effect, energy is only spent when required. Notifications are HTTP callbacks which are sent to a prespecified URL using simple HTTP `POST` requests. In the context of this chapter, we denote an HTTP callback that serves the purpose of transmitting sensor data or a sensed event

as a *sensor callback* and its target URL as a *sensor callback URL*.

Note that the nodes could also record events or sensor readings over a longer period of time and then upload them altogether. This batch upload could in turn be performed at fixed intervals (e.g., once a day) or at given events, such as the detection of an open[17] access point. While this approach would benefit from the potentially high transmission rates of Wi-Fi, it is out of scope of this thesis.

4.3.3 Actuation

In order to be able to control actuators in real time, the node needs to be awake and connected all the time. This way, the built-in Web server may handle incoming requests that control actuators. Each actuator is modeled as a resource and can be updated just as a standard Web resource can. However, this approach requires an external power supply and faces the problem that nodes are usually given a private IP address within a Wi-Fi network, which hampers global accessibility.

When the real-time criteria can be somewhat relaxed, it is possible to implement an alternative approach. There, the node is usually in sleep mode but will wake up periodically in order to poll a prespecified Web resource for updates. If there is a new value, the corresponding actuator will be controlled using the given value. Note that this approach may also be feasible for battery-powered operation as long as the actuator does not consume much energy (e.g., an LCD screen) or is only powered for a limited amount of time (e.g., a beeper). This approach has the advantage that actuators can be globally accessed even if the node is assigned a private IP address.

4.3.4 Augmenting Things

We consider two fundamentally different approaches for connecting things to the Web: The first approach is to attach a preconfigured node to the physical object and use the sensors and actuators that come with the node to monitor and control the object. This idea of attaching small computing devices post hoc to everyday objects was extensively addressed in the Smart-Its project[18] [212, 240]. The vision is that these devices, denoted Smart-Its, can be attached just like stickers and *"augment [existing objects] with sensing, computing and*

[17] An access point that allows Internet access and does not require authentication.

[18] http://www.smart-its.org/

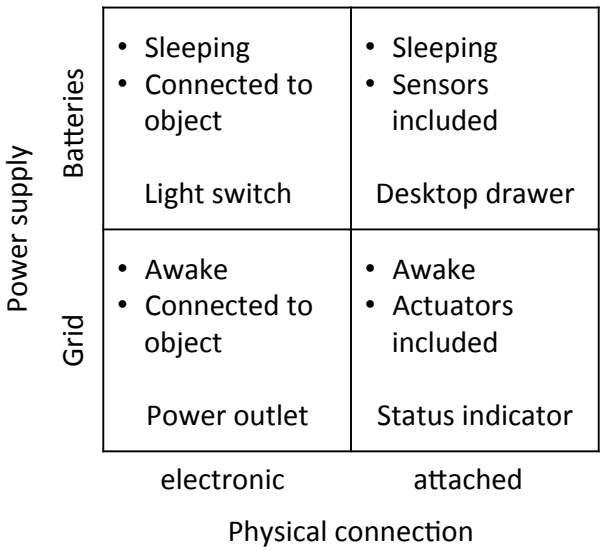


Figure 4.8: Considered design space.

communication capabilities" [212]. For example, a node featuring a motion sensor could be attached to an office chair to automatically detect whether it is currently occupied. This approach has the advantage that these Smart-Its may be commodities that can be installed by untrained users in seconds. However, they cannot use the features of the objects they are attached to.

The other approach is to manipulate the object itself by electrically connecting the node, thereby offering immediate connectivity to the sensors and actuators of the physical object. For instance, embedding the node into a light switch, which is essentially a binary sensor, allows the switch to connect to the Web. This approach enables a much tighter integration of physical entities with the Web. However, embedding a node in an existing device is usually an extensive process that requires special skills and is therefore suitable neither for unskilled users nor for large quantities. However, we consider it out of academic interest – it allows us to test use cases for a future Web of Things.

Orthogonal to this characterization is the question of power supply: Nodes running on batteries have limited energy and will therefore not be able to stay continuously connected to the wireless network. Instead, they will need to persist in a *sleep mode* most of the time and wake up only for certain events. However, if a grid-based power supply is available, nodes will be able to stay permanently connected to the network and thus also be able to control actuators in (near) real time. For example, a node could be attached to a meeting room sign and signal if the room is currently occupied, or it could be embedded in

a switchable power outlet to be able to switch the connected devices via the Web. A summary of the considered design space is depicted in Fig. 4.8.

4.3.5 A Simple Interaction Model for Direct Interoperation

In this section, we introduce a simple interaction model that supports Web-enabled sensors and actuators to interact with cloud services as well as which each other. The model follows a minimalistic approach and implements some concepts of our framework introduced in Chap. 3. We illustrate our model through a simple example consisting of two Web-enabled things: A switchable power outlet, which is a simple actuator and a light switch, which is essentially a binary sensor. Both objects feature a built-in Web server that implements our interaction model.

The power outlet features two resources: the main resource representing the outlet,

`http://poweroutlet/`

located at the root, and a sub-resource representing its actuator, a relay, located at

`http://poweroutlet/power`.

Accessing the root resource returns an HTML representation of the outlet, including a graphical representation of its current switch state (see Fig. 4.9). Accessing the resource of the built-in relay returns a textual representation of its current state: *false* if it is currently switched off, *true* if it is currently switched on. In order to change the state of the relay, one simply needs to `POST` the desired state as a textual representation to

`http://poweroutlet/power`

using the corresponding content-type `text/plain`. This way, the power outlet can be easily controlled using a script on an HTML page, from the command line using a tool like `curl`, or from a program using an HTTP client library.

The light switch features three resources: the root representing the switch located at

`http://lightswitch/`,

a sub-resource representing the state (or sensor) of the switch at

`http://lightswitch/position`,

and a second sub-resource located at

`http://lightswitch/callback`.

Similar to the power outlet, accessing the root resource of the light switch returns an HTML representation including a depiction of the current state of the switch (Fig. 4.10). The state of the light switch is also returned as a textual representation reading *true* or *false*. In order to enable push-based operations, we can read and set the callback target URL using the same approach as for actuators. We can also dynamically adjust the callback target of a sensor based on some external state, such as other sensor readings, time, or location without having to place any additional logic in the light switch.

An important aspect of our approach is that we can link sensors and actuators directly: To control the power outlet with the light switch, one simply needs to update the resource

`http://lightswitch/callback`,

the callback target of the light switch, to the value

"`http://poweroutlet/power`",

which is the URL of the relay of the outlet. Note that this concept of association is comparable to HTML hyperlinks, which are also unidirectional and implemented at the source of the relation. Now each time the light switch is pressed, its state changes and thus an HTTP `POST` request is sent to the actuator of the power outlet, causing it to reflect the current state of the light switch (Fig. 4.11). To realize more complex scenarios, like a light switch that controls multiple actuators simultaneously based on some external condition, one would utilize a service like our framework introduced in Chap. 3, which would receive the HTTP callback from the light switch and distribute it to one or more targets based on some predefined conditions. Connecting sensors and actuators via a gateway or Web service may of course be problematic in certain scenarios: To stay with our home automation scenario, users will probably neither tolerate noticeable delays when pressing a light switch nor the outage of their light controls when a Web service or their Internet connection breaks down.

4.3.6 Monitoring and Run-time Configuration

Deployed nodes should be monitored automatically for correct function and for critical operational parameters, such as battery voltage, in order to detect possible problems in time. It may also be beneficial to be able to remotely change configuration parameters to address deployment-specific issues or even update the application running on the node.

A possible approach would leverage the built-in Web server. However, not every node will be running a Web server, since this requires a constant connection to the Internet and is expensive in terms of energy consumption. Additionally, nodes are usually behind a NAT, and are therefore assigned a private IP address, which makes access from the Internet to the node difficult. Nodes that are battery powered will usually stay in sleep mode and can therefore not provide an HTTP server that is reachable for a reasonable time span.

As a solution, we use the same mechanism used for sensor callbacks: Nodes periodically send *heartbeat* messages to a configured URL (denoted *heartbeat callback URL*) in order to indicate that they are still running. Analogous to a sensor callback, a *heartbeat callback* is an HTTP `POST` request that includes relevant parameters such as the current battery voltage. In the response to this HTTP request, the server may include configuration updates. For example, the server may change the rate of the heartbeats or specify a new sensor callback target. Since the HTTP operation is then finished, the client needs to include the results of the applied updates in the next heartbeat request message. The heartbeat callback can be considered orthogonal to the current application and may be received by a different party than the sensor callback. This way, monitoring of the sensed data and monitoring of the node can be handled by different authoritative domains.

4.3.7 Bootstrapping and Debugging

Since we assume that the node is attached to or embedded within a wide range of entities, we cannot assume that there exists a rich set of input and output capabilities. In fact, there might be no user interface at all. However, the node requires several configuration parameters for its initial setup. Additionally, it may also be necessary to change or query configuration data while the node is operating but has no or only limited network connectivity.

Bootstrapping To be able to operate, the node requires an initial configuration, which covers three domains: (1) the IEEE 802.11 configuration, (2) the IP configuration, and (3) the configuration of the application running on the node.

(1) The IEEE 802.11 domain usually requires the *Service Set Identifier (SSID)* of the access point[19] (the "name" of the access point) and a passphrase, which is used to derive a key[20]. Using this information, the node can search for the specified access point, authenticate itself, and associate with it.

(2) The IP domain can by configured automatically using DHCP [241], which is the de facto standard in today's Wi-Fi networks. Note that Wi-Fi networks usually assign IP addresses from a *private network range*, as they are placed behind a NAT. Without configuration of the infrastructure, the IP address is usually dynamic, which applies also to the DNS name, which may additionally be assigned to the node. However, the node may register its acquired IP address with a dynamic DNS service that provides a fixed DNS address for the node. This DNS address could be printed on the device, making it well known.

(3) If the node does not provide a Web server, i.e., it resorts to sleep mode unless there is a specified event, then the application requires at least a callback URL for its heartbeat messages, which can be hard-coded. The response to a heartbeat message may then be used to configure the required parameters of the application running on the node.

The initial configuration could be performed by placing the node in ad hoc mode, and then running a Web server. Users would then need to connect to the node and could then configure the required settings. Alternatively, the industry standard Wi-Fi Protected Setup (WPS) [242, 243] allows the association of a device to an access point by pressing a button on each of the devices to pair them (among other methods). There are also other possible bootstrapping scenarios that leverage the 802.11 communication channel, such as temporarily changing the settings of the existing access point to values that are printed on the node. Of course, one could also use a cable-based interface for bootstrapping.

[19] Note that the node could also opportunistically use open access points.
[20] e.g., for WPA and WPA2. Note that there are other authentication methods.

Run-time Monitoring and Debugging When the node is operating, configuration data may be changed via the heartbeat responses or by accessing the Web server of the node, if applicable. Additionally, run-time parameters may also be monitored over these interfaces in order to be able to detect possible problems early. However, as soon as the node has no or limited network connectivity, configuration obviously cannot be changed remotely. Additionally, the local identification of the problem cause is aggravated by the limited user interface of the node. Finally, there might also be parameters whose query or update should require physical presence, such as for security reasons.

4.3.7.1 Proposed Solution

A Secondary Communication Channel To alleviate the problem of limited or missing physical user interfaces and to be able to interact with the node even if there is no or only limited network connectivity, we propose to use a *secondary communication channel*. This channel serves as a backup channel that can be used for the bootstrapping, run-time configuration and monitoring, and debugging of the node. It requires neither the communication range nor the bandwidth of the primary channel, as it serves only as a backup link. Furthermore, a connection of the secondary communication channel to the Internet is not mandatory, as local communication may be sufficient for a given scenario (e.g., bootstrapping of a device). Using a wireless communication channel as opposed to a physical interface has the advantage that bootstrapping and debugging of the node may be automated (requiring no user interaction) or semi-automated (requiring only limited user interaction). In fact, this concept has already been investigated in the area of wireless sensor networks. For example, the BTnode [244], a wireless sensor node from ETH Zurich, features two wireless interfaces: A low-power, low-bitrate radio intended as the primary communication channel and an additional Bluetooth interface, which may act as a secondary communication channel. Based on this hardware, several projects investigated how a secondary wireless communication channel can be used for deployment support [245] or for ad-hoc infrastructure access [246], for example.

The EPC Interface as Communication Channel For our work, we leverage the EPC Gen 2 RFID interface of the RN-131 as a secondary commu-

nication channel. The EPC Gen 2 standard operates in the UHF band (860 MHz - 960 MHz) and provides for communication ranges of up to several meters and sufficient communication bandwidth. EPC Gen 2 is an accepted industry standard, that is used for supply-chain monitoring, for example. Compared to other RFID technologies such as NFC, readers are currently more expensive and less widespread. However, there are also handheld readers and even prototypes of EPC-enabled mobile phones [247]. EPC Gen 2 tags may have a user-memory area that can be used to store custom data on the tag. Since the RN-131 may act as a smart EPC tag, it is possible to communicate with the application running on the node using a shared memory communication protocol that operates on the user memory area of the tag. The protocol needs to coordinate the read and write operations of the node and the RFID reader on the shared user memory area. This way, a secondary communication channel may be realized over the EPC interface.

4.4 Implementation

In this section, we discuss some aspects of the implementation of our approach.

4.4.1 Software

We implemented both a single-threaded HTTP server and an HTTP client, which run on the RN-131. Due to resource constraints, we had to carefully optimize the implementation to include only the features required for our concept.

4.4.1.1 Callback Cycle

When powered with batteries, the node has to be kept in sleep mode and wake up only on specific events, perform a sensor or heartbeat callback, and return to sleep mode again. This callback cycle consists of the following steps:

1.) CPU and RAM are powered up and the node boots our application.

2.) The node connects to a pre-configured access point[21].

[21] Note that this step consists of searching, authentication, and association with an access point.

3.) An IP configuration is acquired via DHCP.

4.) The IP address of the callback target is looked up.

5.) A TCP/IP connection to the host of the callback target is created.

6.) The HTTP callback is performed.

7.) The TCP/IP connection is shut down.

8.) The node disconnects from the access point.

9.) The node returns to sleep mode.

This process can also be optimized in order to save network traffic and reduce the time the node spends awake. In particular, we also implemented an optimized callback cycle that caches the following data in the non-volatile memory (NVM) of the node: The wireless channel of the access point and its BSSID[22], the IP configuration, the ARP table, and the DNS table. These improvements can significantly reduce the time required to connect to the callback target, as long as the BSSID or wireless channel of the access point has not changed and the cached data are still within the validity period.

4.4.1.2 Web Interface

As noted in Sect. 4.3.1, the HTTP sever on the node provides not only an interface for other applications (an API) but also an interface for users, based on HTML. Technically, the interfaces for the HTML front end and the API are the same, i.e., the Web front end actually leverages the resources of the API. This reduces complexity and reduces the effort necessary for a developer to include the node in a script, as its API can be learned by investigating the Web-frontend in a Web browser.

As an example, Fig. 4.10 depicts the front end of a Web-enabled light switch, and Fig. 4.9 depicts the front end of a Web-enabled power outlet. Note that the data on these two front ends are actually "live", so that a press of the light switch is visually reflected on its Web-frontend. This is currently implemented by polling the respective resources periodically using JavaScript[23]. In order to save energy, the HTTP server supports caching and delivers the images with a long validity period.

[22] The MAC address of the access point.

[23] There are other approaches, which would however consume additional resources on the server.

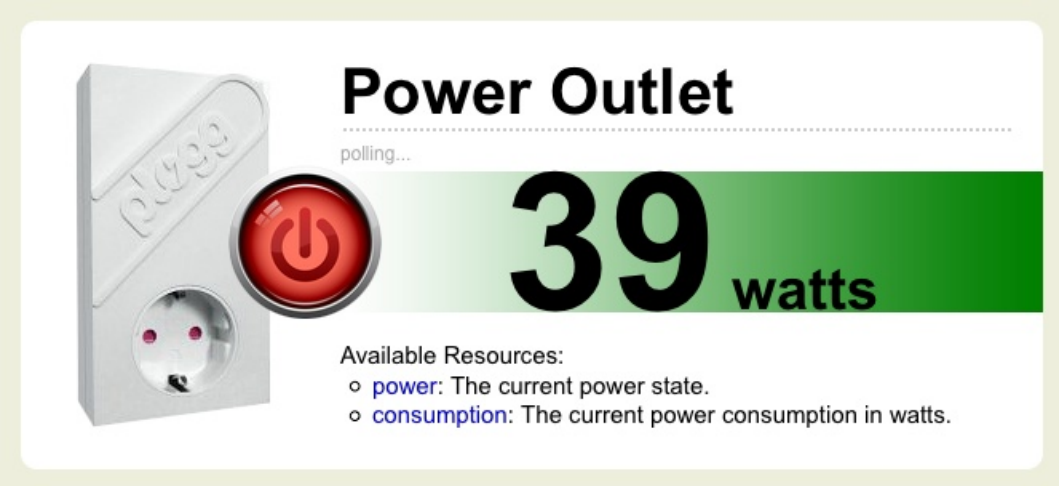


Figure 4.9: HTML front end of the Web-enabled power outlet.

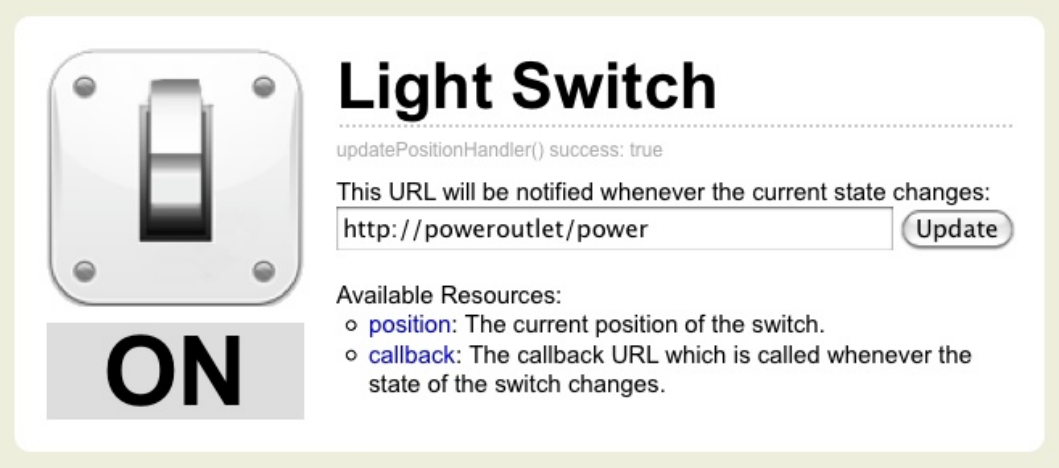


Figure 4.10: HTML front end of the Web-enabled light switch.

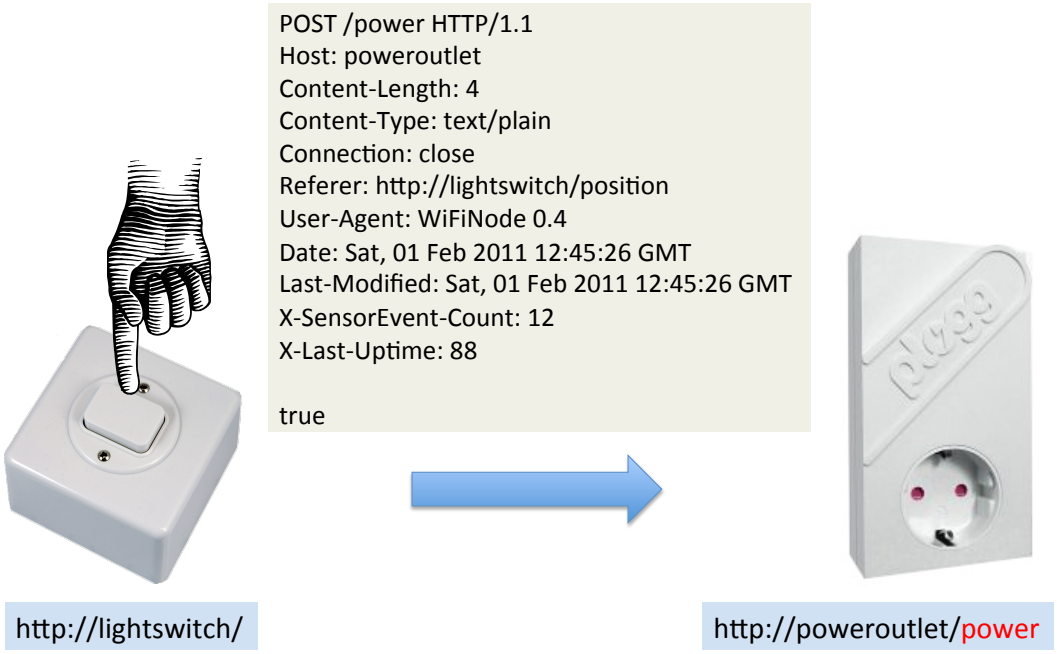


Figure 4.11: Direct communication of a sensor and actuator using HTTP callbacks.

4.4.1.3 Sensor and Heartbeat Callbacks

Due to its minimalistic concept, the implementation of a sensor callback is straightforward. However, we specifically leverage some headers in the HTTP request, which is discussed in the next section. An example of a sensor callback is depicted in Fig. 4.11.

For a heartbeat callback, which potentially includes many different parameters, we use the JavaScript Object Notation (JSON) data-interchange format [97], as it is widely used, supported by many programming languages, has less overhead than XML, and is also simple to understand. An example of a heartbeat request is provided in List. 4.1, and an example of a heartbeat response with configuration updates is provided in List. 4.2.

4.4.1.4 Leveraging Features of HTTP

Leveraging HTTP for communication has the advantage that the protocol already provides solutions for some problems that arise at the application level. Of course, these problems can be solved using application-specific solutions. However, leveraging the features of HTTP increases interoperability and reduces the complexity of the application. In the following, we outline how some application-specific problems can be solved using features provided by HTTP.

Initialization of the Real-Time Clock When the node is reset, the on-board real-time clock (RTC) needs to be initialized. Since sub-second accuracy is not required for our application scenarios, we do not utilize the Network Time Protocol (NTP) [248] but instead leverage the `Date:` header that an HTTP server must return in its response[24]. The server hosting the heartbeat target is used as a time reference. As a heartbeat message is the first message that is sent after a reset of the node, the RTC can be initialized early using the heartbeat response.

Piggybacking Information For debugging purposes, we experimented with including debugging information within custom HTTP headers in the sensor callback request. Since HTTP headers are used for out-band signaling, the interface for application does not need to change. For example, for sensor callbacks, the custom headers `X-SensorEvent-Count`

[24]There are some exceptions, e.g., when the server is experiencing an error or has no clock [64].

```
{
    "message": "HEARTBEAT_EVENT",
    "time_offset": 1336390223,
    "id":
    {
        "MAC_address": "00:06:66:13:af:73",
        "application": "WiFiNode 0.7a"
    },
    "sensors":
    {
        "sensor_state": false,
        "battery_voltage": 3111
    },
    "counters":
    {
        "sensor": 0,
        "heartbeat": 5,
        "restart": 5,
        "limiter": 0,
        "watchdog_reset": 0
    },
    "last_times":
    {
        "search": 11,
        "association": 27,
        "DHCP": 1,
        "DNS": 0,
        "callback": 32,
        "total": 91
    },
    "cumulative_times":
    {
        "up": 7703,
        "doze": 3798,
        "down": 42889,
        "total": 50662
    },
    "AP":
    {
        "SSID": "wifi-tests",
        "BSSID": "00:23:69:1a:73:6b",
        "channel": 9,
        "rate": 24,
        "lastRSSI": -34
    },
    "IP":
    {
        "local": "10.22.19.138",
        "netmask": "255.255.255.0",
        "gateway": "10.22.19.1",
        "dns": "10.22.19.1"
    }
}
```

Listing 4.1: Exemplary contents of a heartbeat callback, which is sent by the node to the heartbeat callback URL.

```
{
    "heartbeat_period": 3600,
    "sensor_callback_url": "http://example.org/actuators/1"
}
```

Listing 4.2: Sample of a heartbeat response, returned by the server to the node. In this example, the server instructs the node to set its heartbeat period to one hour and specifies a new callback URL for sensor events.

and `X-Last-Uptime` include the number of sensor events and the length of the last uptime.

Avoiding Software Update Loops When the application on the node is upgraded using a heartbeat response message, there is the problem of an update loop: The node will download the new application code, reboot, and send a heartbeat message and might still receive the instruction to download the new application code in the heartbeat response. Of course, the server could include the update instruction only as long as the node's application version is older than the provided update or the server could include the version of the offered upgrade in the heartbeat response, so that the node is able to decide if it should proceed to download it. However, this problem can also be solved by leveraging the caching support of HTTP. In particular, a conditional `GET` request is used for downloading the new firmware image with the `If-Modified-Since` header set to the creation date of the image file. This is usually supported by default HTTP server configurations.

Determining the Origin of a Sensor Callback HTTP allows *"the client to specify, for the server's benefit, the address (URI) of the resource from which the Request-URI was obtained"* [249]. This referring address is transmitted in the `Referer`[25] header of the HTTP request. For our purpose of specifying the origin of a sensor callback, we set the contents of the `Referer` header to the URL of the sensor that caused the event.

Timestamping Sensor Readings Since a sensor reading may be acquired a considerable time before it is transmitted – due to network problems, for example – it is crucial to include the timestamp of the sensor reading in the sensor callback. For this, we leverage the `Last-Modified` header, with which we indicate the timestamp of the sensor reading. Note that

[25] Note that the header name is actually misspelled.

the HTTP specification only considers the use of this header in the HTTP response, while we also utilize it in the HTTP request. We additionally include the current time of the node in the `Date` header to allow the receiving server to detect potential time clock drifts.

Supporting Polling Actuators As outlined in Sect. 4.3.3, nodes controlling actuators may resort to sleep mode when the real-time requirement for actuation is relaxed. Using this approach, nodes will periodically wake up and poll a given resource. However, the decision as to whether the actuator should be triggered cannot be solely based on changes of the body of the monitored resource. For example, consider a bell that periodically monitors a resource representing occupancy in a room (true or false). The bell should ring whenever the state of the room changes. Ringing the bell only when the value has changed may not be sufficient, as the resource may have changed from occupied to empty to occupied between two polling requests. For this reason, changes of the `Last-Modified` header of the response have to be considered. This is achieved by using a conditional `GET` request with the `If-Modified-Since` header set to the `Last-Modified` date returned by the last access to the resource. Depending on the returned status code, the node decides whether it should drive the actuator. Note that this is a general-purpose approach to detecting changes of Web resources without the need to analyze their contents, and it should work with any Web resource supporting conditional `GET`s.

4.4.1.5 The RFID-based Communication Channel

As outlined in Sect. 4.3.7, we create a secondary communication channel by leveraging the EPC user memory area of the node as a memory that has shared access by the node and the RFID reader. Both entities are able to read and write data from and to this area. In order to coordinate the communication, we implemented a simple memory-based request/response communication protocol in which the node acts as server and the RFID reader acts as client. Using this protocol, the reader may read data sets from the node, update data sets on the node and execute commands on the node. The node signals whether it is busy and whether an operation was successful or could not be performed. For example, one is able to query the node for its software version and current IP address, set the SSID and passphrase of

the access point configuration, or trigger the execution of a heartbeat callback.

Additionally, the node also signals its general state, such as whether it requires the configuration of an access point. This can be used for automatic bootstrapping of nodes, for example. In this case, as soon as a node that requires certain settings enters the communication range of an RFID reader, it is automatically updated with the requested parameters. To simplify identification, the last 6 bytes of a node's EPC address are automatically set to its MAC address, and all nodes also share a common EPC prefix that enables the detection of their type. The EPC standard supports multiple address schemes; we utilize the General Identifier (GID-96) encoding scheme [250].

In order to communicate with the node over the RFID-based communication channel, a Java program was developed that interacted with the RFID reader using the low-level reader protocol (LLRP) [251]. The program utilizes the LLRP Toolkit Java library[26], an open-source implementation of LLRP.

4.4.2 Hardware

Our analyses are based on a RN-134 board that was physically altered in order to reduce its power consumption to that of the RN-131: The voltage divider and RS232 driver were removed from the board, all LEDs were deactivated, and the board was configured for a supply voltage range of 2.0 V - 3.3 V (VBATT option) [226].

We experimented with three different types of sensors, which were chosen to enable the measurement of activity of an object or a room the node is attached to. The first was a reed switch, which in combination with a permanent magnet creates a contact-less switch, for example to monitor whether a door is currently open or closed. The second was a micro-mechanical "ball-in-tube" motion sensor, which detects whether the attached object is currently moving. The third was a passive infrared (PIR) sensor, which can detect the presence of people in a room, for example. The first two sensors are passive elements that do not consume power by themselves, and the PIR sensor is an active element that consumes about 300 μA. While all of the sensors are binary, neither the platform nor our concept impedes the use of non-binary sensors. Exemplary deployments are depicted in Fig. 4.12.

[26] http://llrp.org

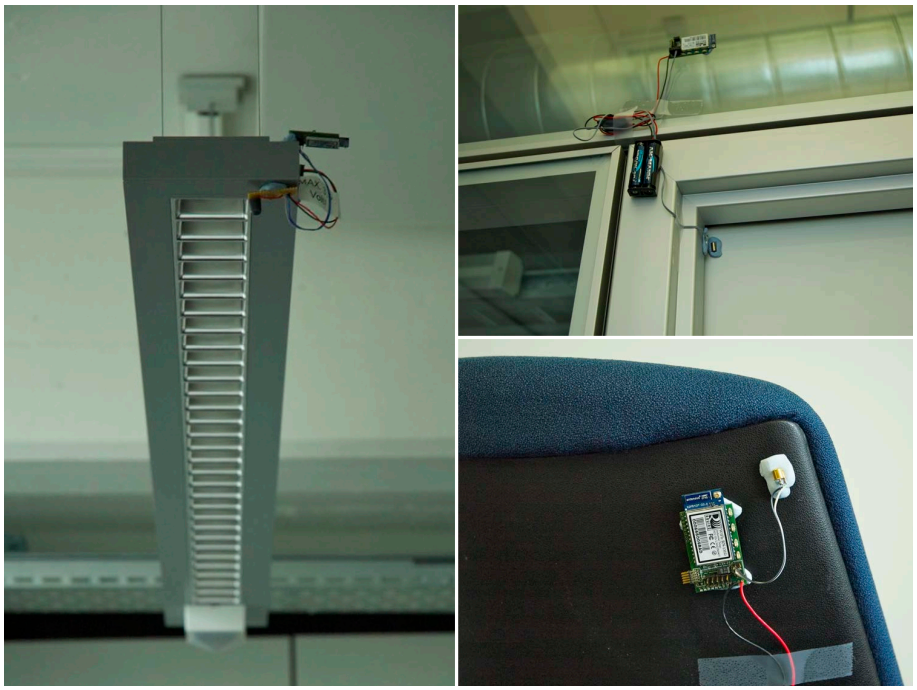


Figure 4.12: Everyday objects augmented with attached ultra-low-power Wi-Fi nodes. Left: PIR sensor attached to ceiling lights (note the existing PIR sensor in the background); top right: reed sensor attached to door; bottom right: ball-in-tube motion sensor attached to the back of an office chair.

We also embedded the RN-134 into several objects (see Fig. 4.13), including a light switch, a power plug based on a modified Plogg[27], which can not only switch a connected consumer load but also sense its current power consumption, and an LED candle, which can act as a subtle notification device. Additionally, we modified a wireless doorbell system, replacing the original communication hardware in the switch and the doorbell with an RN-134.

4.5 Evaluation

We performed several experiments in order to evaluate the practicality of our approach under various conditions. Because a crucial part of our concept is the leveraging of existing infrastructures, which are inherently beyond our control, several experiments were conducted in uncontrolled environments. Even if all components of an experiment were under our control, there is still the wireless communication channel. For the Wi-Fi interface, it is located in the unlicensed ISM band at 2.4 GHz. Besides an increasing amount of Wi-Fi networks, this spec-

[27] http://www.plogg.co.uk

Figure 4.13: Examples of developed Web-enabled Things with embedded ultra-low-power Wi-Fi nodes that were experimented with in the context of this thesis: switch, buttons, power plug, doorbell, and LED candle.

Setup	S1	S2
Location	Office Space	Office Space
Authentication	None	WPA2-PSK
Multiple BSSIDs	Yes	No
DHCP lease time	15 minutes	15 minutes

Table 4.1: Access point setups considered for the evaluation of the power consumption of a callback cycle.

trum is also populated by devices using Bluetooth, IEEE 802.15.4, or proprietary communication standards and also susceptible to noise generated by non-communication devices such as microwave ovens. Since we did not use an anechoic chamber, our experiments are at best *semi-controlled*, as the wireless channel is always beyond our control.

4.5.1 Power Consumption of Callback Cycles

Since the amount of energy consumed by the node when it is in sleep mode is several magnitudes smaller than the amount of energy consumed when it is awake (see Sect. 4.2), the power consumption of a callback cycle is crucial for the lifetime on batteries. As our approach is optimized not for energy consumption but for interoperability, we are interested in its energy requirements.

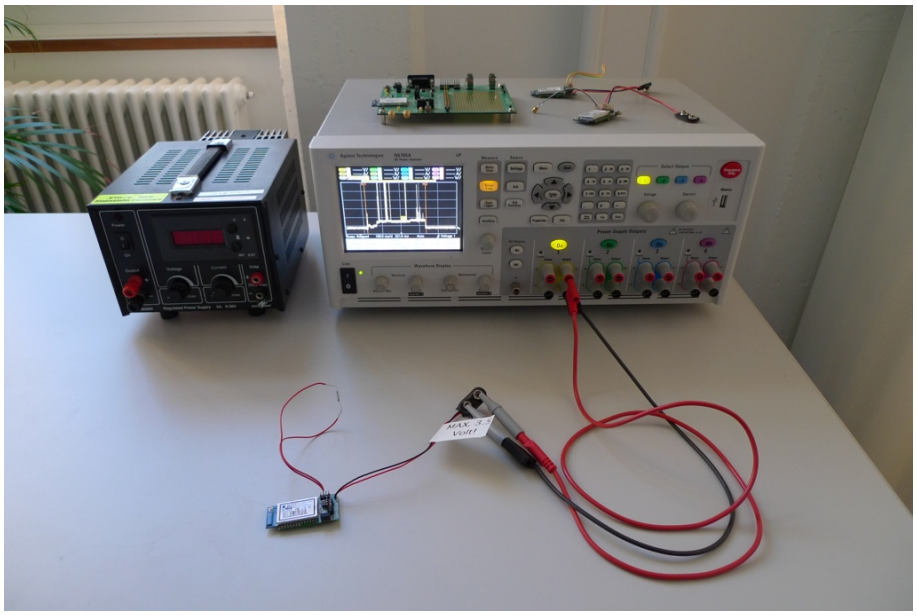


Figure 4.14: Setup for energy measurements. The device used to power the node and measure its power consumption was an Agilent Technologies N6705A power analyzer.

Setup Our analyses are based on a modified RN-134 board as outlined in Sect. 4.4.2. The device used to power the node and measure its power consumption was an Agilent Technologies N6705A DC power analyzer.

The node was programmed to periodically wake up and perform a callback cycle, sending a heartbeat message (similar to List. 4.1) to a PHP script hosted at the Web server of our department and waiting for the response. This server was a productive system hosting the Web pages of the members of the author's department[28]. The software on the node was configured to avoid any unnecessary actions such as printing debug output to the serial console[29]. The connection rate was set to 24 Mbit/s. We evaluated the callback cycle in the optimized and non-optimized implementations (see Sect. 4.4.1.1).

Two different access point setups were considered. In the first setup, we leveraged the Wi-Fi infrastructure of ETH Zurich at the offices of the author's institute, which is an enterprise-level system consisting of multiple access points and dedicated DHCP and DNS servers. In the second setup, we utilized a dedicated consumer-level router with built-in DHCP and DNS servers at the same location. The setup is summarized in Tab. 4.1.

[28] In fact, the script was located on the homepage of the author, so deployment required little effort.

[29] Note that printing data to a serial terminal can require a significant amount of time.

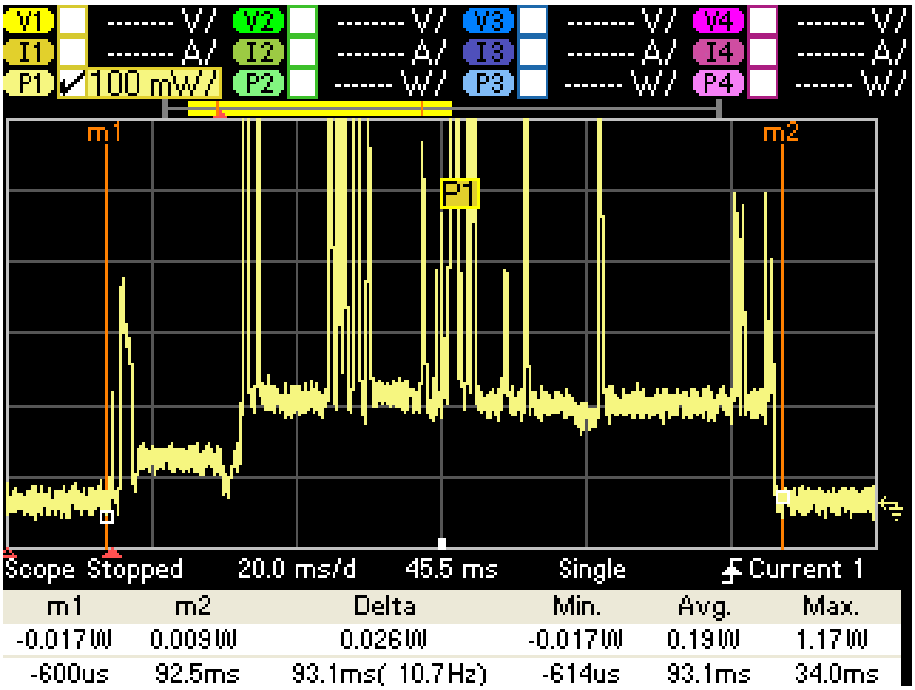


Figure 4.15: Screenshot of the results of a power measurement run. The markers $m1$ and $m2$ have been adjusted manually to match the start and the end of the wakeup cycle.

Methodology For our experiments, the supply voltage was set to 2.4 V, which is comparable to the voltage supplied by two rechargeable AA or AAA batteries. For each setting, we performed 10 measurements of *optimal* callback cycles – that is, we disregarded callback cycles that resulted in failed callbacks or had an outlined uptime. The results can therefore be considered as a lower bound. For each measurement, we let the power analyzer trigger the recording and then manually set the start and end point of the callback cycle using the interface of the N6705A. The power analyzer then calculated the length of that interval and the average power consumed during that period (Fig. 4.15). In order to assess the effects of the network optimizations, we performed the experiments both with and without the optimizations for the callback cycle. Our test setup is depicted in Fig. 4.14, and an example of a recorded energy trace is shown in Fig. 4.15.

Results and Discussion The results of our experiments are listed in Tab. 4.2. We see that the average uptime at S1 with no optimizations is significantly larger than for the other experiments. This is because at S1, the DHCP and DNS servers were part of the infrastructure of our university and were handling requests for larger parts of the network. Most of the uptime was spent waiting for the DHCP lease. The opti-

Setup	Callback cycle	Avg. uptime	Avg. power	Avg. energy	Savings
S1	not optimized	1232.10 ms	0.126 W	155.24 mJ	—
S1	optimized	69.38 ms	0.174 W	12.07 mJ	92.22 %
S2	not optimized	103.50 ms	0.194 W	20.08 mJ	—
S2	optimized	89.16 ms	0.203 W	18.10 mJ	9.86 %

Table 4.2: Energy consumption per callback for an optimal callback cycle. Results are averaged over ten measurements.

mized callback cycle avoided these requests (as long as the cached data were within the validity period) and could therefore significantly reduce the uptime and in turn energy consumption. At S1, the optimized callback cycle required only 7.78 % of the energy of the non-optimized version, which is estimated to be a potential increase of the overall runtime of the factor 12.9[30].

At S2, the DHCP and DNS servers were running on the wireless router. As the response times for these services were already low, the optimizations resulted in only minor improvements of the energy consumption. The optimized version required 90.1 % of the energy of the non-optimized wakeup cycle, which is estimated to be a potential lifetime increase of a factor of 1.11.

The comparison of the results of S1 and S2 with optimizations shows that the use of WPA2 adds an overhead both to the uptime and to the required power. This is expected, as the handshake required for WPA2 requires significant resources. From our experiments, we see that the use of WPA2 increases energy consumption to about 50%, compared to an unsecured access point. We can therefore estimate that the runtime of a node using WPA2 is only about 66 % of that of a node using no authentication.

4.5.2 Performance of HTTP in Semi-Controlled Environments

In order to provide a baseline for our approach of using HTTP for communication over the IEEE 802.11 interface, we tested our implementations of the HTTP client and the HTTP server in a semi-controlled environment (i.e., all parts of the experiment except the wireless channel were under our control). We are interested in the time required for a callback cycle, as this directly correlates to the energy consumption of the node. It also has an effect on the latency, which can be noticed

[30]Ignoring power consumption in sleep mode, self-discharge of batteries, etc.

by the user. The latency is also relevant when the node is providing an HTTP server for interaction with its actuator(s). For this reason, we evaluated both the execution time of a callback cycle and that of an HTTP request being processed by the HTTP server on the node under various conditions.

We used an isolated wireless router in order to avoid traffic on the communication path that was not related to our experiments. The LAN and WLAN segments were bridged, which is the usual setup for wireless consumer-grade routers. We used the optimized version of the callback cycle for all experiments in this section (see Sect. 4.4.1.1).

4.5.2.1 Performance of a Callback Cycle

Setup We evaluated the influence of two factors on the duration of a callback cycle: The communication rate and the security mechanism used to connect to the access point. For this, we tested all the communication rates of 802.11g (6, 9, 12, 18, 24, 36, 48, and 54 Mbit/s). We also analyzed the influence of security mechanisms on the wireless link by running the experiment both without any security features and with WPA2-PSK enabled[31].

For this, we connected a laptop running Mac OS X with an Apache HTTP server[32] to the LAN segment of the router. The node was configured to perform a callback cycle every 10 seconds, sending a heartbeat request to a PHP script running on the laptop. The script logged the contents of the heartbeat message and returned configuration updates when necessary. Communication rates were cycled using these configuration updates in order to alleviate external influences. In order to achieve good signal quality, the node was placed approximately 1 m away from the wireless router.

Results and Discussion The results are depicted in Fig. 4.16 (no security) and Fig. 4.17 (with WPA2-PSK) and show the average time required for 1000 successful callback cycles per communication rate[33]. The total time required is further broken down into the time required for the connection to the access point (this includes association and, if applicable, authentication), the time required for the callback, and the time spent awake for other tasks.

[31] WPA and WEP were not tested, as both were superseded by WPA2.

[32] http://httpd.apache.org/

[33] With the exception of 36 Mbit/s WPA2-PSK, for which only 817 requests were considered.

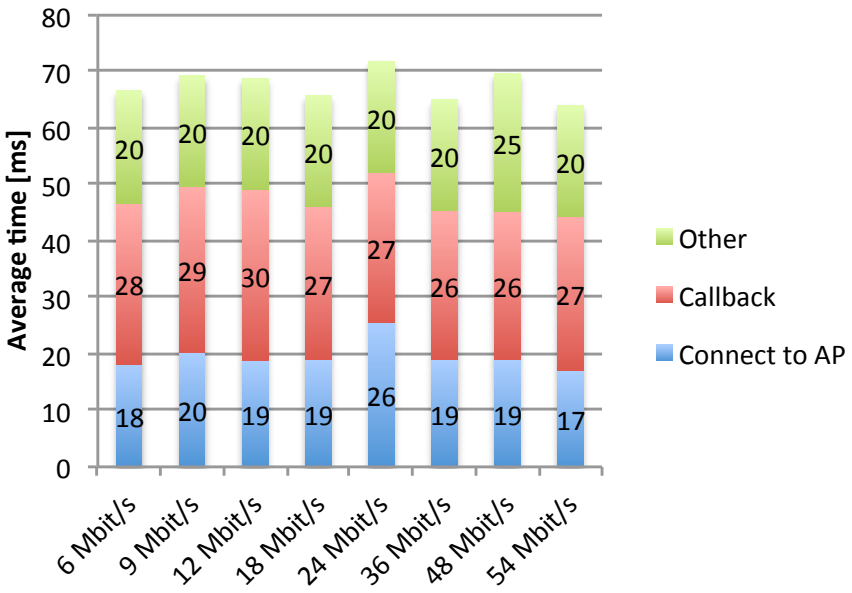


Figure 4.16: Average time required for a callback cycle with no authentication in a semi-controlled environment.

Both figures show that there is no obvious correlation between uptime and communication rate. This can by explained by the fact that in this setup, the transmission time over the wireless channel was not a limiting factor. The time required for the connection with the access point[34], the round-trip time (RTT), and the processing time required for TCP, HTTP, and the PHP script were considerably higher than the time required to transmit a frame on the wireless channel. Since our payload was small, higher communication rates could not produce an effect. However, enabling WPA2-PSK had a significant impact on the overall uptime. This was caused by the additional handshake required for authentication, which manifested in an increased connection time. Interestingly, the callback time was also higher – this might be attributed to the computing overhead required for cryptographic operations. Note that for this experiment, DHCP and DNS were not taken into account, as the validity period was longer than the duration of the experiment. The total averages for all communication rates were 68 ms (total uptime) and 28 ms (callback) for no encryption and 108 ms (total uptime) and 36 ms (callback) for WPA2-PSK.

[34]Which is performed at 1 Mbit/s.

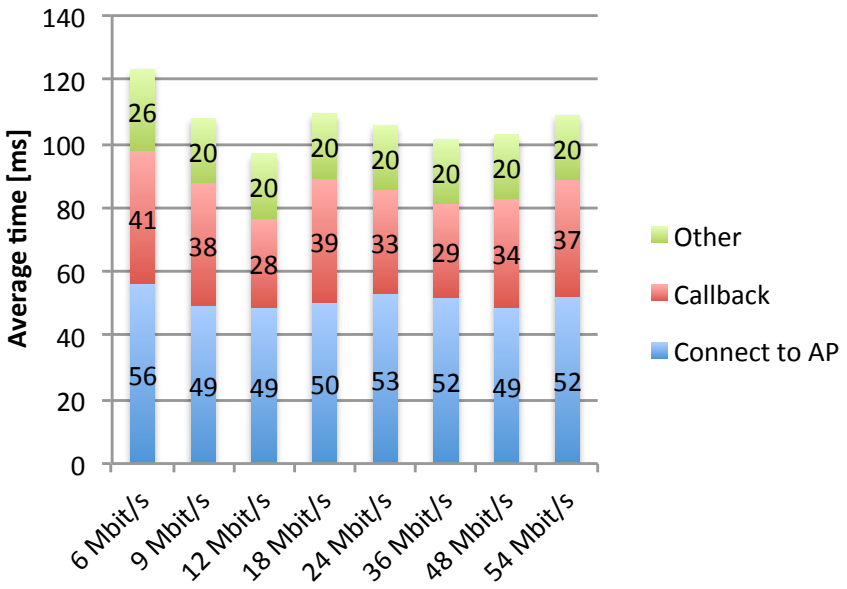


Figure 4.17: Average time required for a callback cycle with WPA2-PSK in a semi-controlled environment.

4.5.2.2 Performance of the HTTP Server

Setup The experiments in this section evaluated the response times of our Web server running on the node. The node was configured with a communication speed of 24 Mbit/s. WPA2-PSK was used to secure the wireless channel. We issued 1000 consecutive requests from a computer connected to the LAN interface of the access point to retrieve a static HTML page served by the node. For comparison purposes, we also performed the same measurements using the laptop from Sect. 4.5.2.1 as a Web server. For this, it was connected to the wireless interface of the access point and running an Apache HTTP server. Both the node and the laptop were placed approximately 1 m away from the wireless router.

Results The average time for processing the HTTP request, measured at the client between the start of the TCP connection and its shutdown, was 24 ms. This qualifies as (near) real-time control of actuators. The Apache HTTP server running on Mac OS X on the laptop required 16 ms on average to serve the static page.

4.5.2.3 Performance of Node to Node Communication

We also tested the communication between two nodes associated with the same access point. For this, one node ran the Web server, while the

Scenario	S1	S2	S3
Location	Office Space	Home 1	Home 2
Authentication	None	WPA	WPA2
Multiple BSSIDs	Yes	No	No
DHCP lease time	15 mins	24 hours	24 hours

Table 4.3: Settings considered for the field test.

other node performed a callback cycle every 10 seconds, which targeted the first node. The client node was configured to sleep between the requests. This setup resembles a home automation scenario in which smart appliances use HTTP for communication. In such a scenario, the delay between a sensed event and its associated action is of particular interest. In our example of the light switch and the power outlet, the user will most likely accept only a short lag between the press of the button and the control of the associated device. In this experiment, we measured the time of the complete callback cycle at the client. The average time for 1000 callback cycles was 98 ms.

4.5.3 Performance of Callback Cycles in the Field

Setup In order to test the performance of our approach in the field, we utilized the existing Wi-Fi infrastructure at three different locations, listed in Tab. 4.3. S1 is the same as in Tab. 4.1 and leverages the Wi-Fi infrastructure of ETH Zurich at the offices of the author's institute, which is an enterprise-level system consisting of multiple access points and dedicated DHCP and DNS servers. The access points at S2 and S3 are single, standard consumer-level routers with built-in DHCP and DNS servers.

The nodes were configured to wake up every 10 seconds and send a heartbeat callback to a PHP script hosted at the Web server of our institute, using the optimized callback cycle. This is the same callback target as in Sect. 4.5.1, i.e., hosted on a server that serves the Web pages of the employees of our department. The size of a heartbeat message (including HTTP headers) was approximately 900 bytes. The preferred connection rate was set to 24 Mbit/s. In order to prevent excessive draining of the batteries due to network problems, we limited the maximum time the node could stay awake to 5 seconds. Because each heartbeat message also included a sequence number, we could detect failed callbacks at the server. A callback cycle may fail for several reasons, including failure to associate with the access point,

Scenario	S1	S2	S3
Total average uptime	128 ms	930 ms	182 ms
Callbacks initiated	297180	74444	241428
Callbacks received	295314	66300	240970
Callbacks received (%)	99.37 %	89.06 %	99.81 %

Table 4.4: Results of the field test.

network problems, and problems at the callback target.

Results and Discussion Table 4.4 provides aggregated data for each of the three scenarios. One can see that the node at S2 performed significantly worse than the nodes at the other scenarios. This is due to the access point used at that location; replacing it with the model used at S3 resulted in performance comparable to S3. This is an important issue, as we cannot assume the existence of access points of a given model when leveraging existing Wi-Fi infrastructure. Note that success rates could be increased by queuing failed callbacks for later transmission, which is currently not implemented.

For a detailed analysis of the average awake times, which is depicted in Fig. 4.18, we considered only data of successful callback cycles. For each scenario, the average time was further split into time required for the connection with the access point, for retrieving the IP configuration via DHCP, for performing the callback cycle, and for other operations. Because we cached DNS entries, the time for resolving the IP address of the callback target was negligible (< 0.2 ms in each scenario) and thus is not depicted. The node performed best at scenario S1, requiring only 98 ms on average for a successful callback cycle. Interestingly, the additional overhead caused by the short DHCP lease time is overcompensated by the short transmission time to the HTTP target, which is hosted at the same site. For S2 and S3, DHCP time has no impact because a lease is valid for 24 hours. However, for these scenarios we see that the callback requires considerably more time, which is caused by the higher round-trip times to the target host. The time required to connect to the access point at S2 was significantly longer than for the other scenarios, because of the mentioned interoperability issues with that access point. The shortest callback cycle times at S1, S2, and S3 were 52 ms, 79 ms, and 97 ms, respectively.

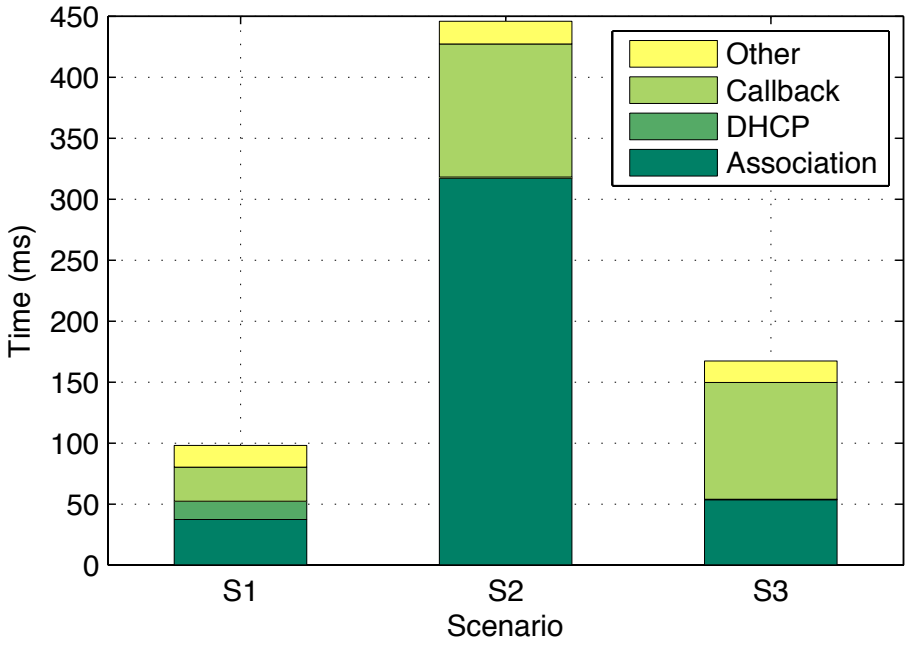


Figure 4.18: Average time required for a callback cycle, considering successful callbacks only.

4.5.4 UHF Communication Channel

Setup We evaluated our implementation of a UHF-based secondary communication channel for bootstrapping and debugging (see Sect. 4.4.1.5) using an Impinj Speedway Reader [252] in combination with a CSL CS777 Brickyard near-field antenna [253][35]. Since the RN-134 does not offer a connector for an UHF antenna, we had to use an improvised configuration: We soldered the antenna wire to the corresponding pad of the RN-134. We tested a professional antenna designed for the 865-928 MHz spectrum (UHF) [254] as well as a bare wire antenna. An exemplary setup is depicted in Fig. 4.19, showing the two antenna types used for the nodes.

On the client side, we used our Java application that communicates with the RFID reader through LLRP [251]. We initialized the RFID reader using Impinj's LLRP extensions, as the communication rates with a generic initialization were unsatisfactory (in this case, only one operation per second was possible per tag). The node itself was put in sleep mode and configured to wake up on RFID write operations.

With this setup, we could successfully communicate with the node over the RFID-based communication channel. We achieved comparable performance within the specified read range of the reader antenna

[35] This setup was readily available at the author's institute.

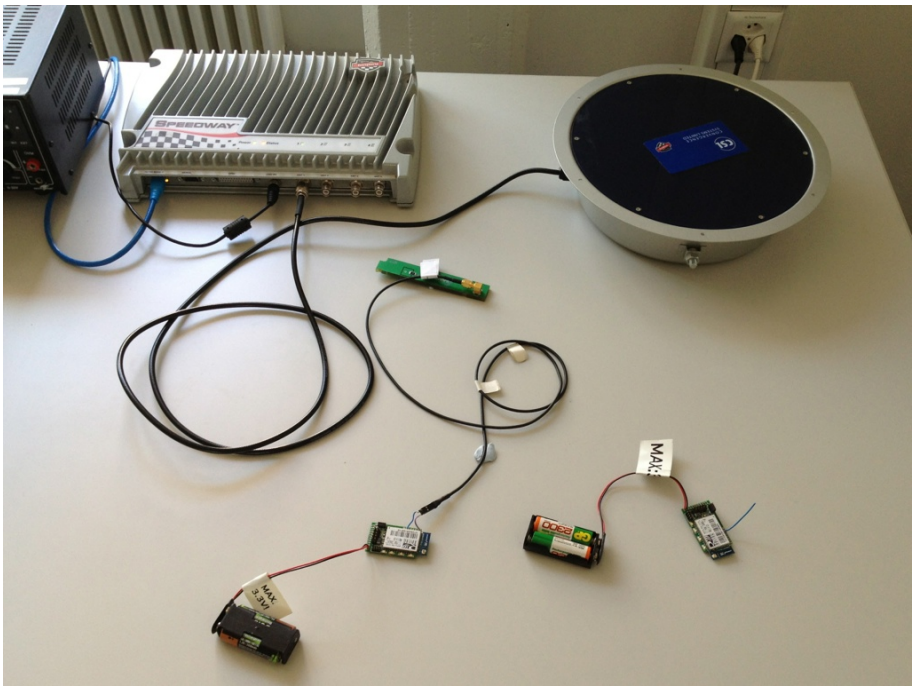


Figure 4.19: Exemplary setup for the test of the UHF-based communication channel. Since the RN-134 does not offer a connector for an RFID antenna, we had to use an improvised antenna configuration. The RFID reader antenna is located in the upper right corner of the image. Both nodes shown in the setup were able to communicate over the RFID-based communication channel from their given locations.

(40 cm), both with the UHF antenna and with the bare-wire antenna. Using our memory-based communication protocol, we could read and write configuration data and execute commands as expected.

In order to evaluate the EPC communication channel, we tested two different use cases: The query of a node's current IP address, and the bootstrapping of a node. Technically, the first use case is implemented by issuing a single command, which requires three read and one write operations to the tag memory from the reader. For this, our Java program issues at least 12 LLRP commands to the reader. Note that the node was given an execution time of 200 ms, after which the reader attempted to read out the IP address from the tag memory. The second use case, the bootstrapping of a node over the RFID interface, consisted of setting both the SSID and the passphrase of the access point and the definitions of the heartbeat callback URL and sensor callback URL. The overall payload to write to the node was 118 bytes. Bootstrapping is a more complex process that consists of four commands each requiring three read and three write operations. With our implementation, the bootstrapping scenario consists of at least 237 LLRP commands that

Scenario vs. Node Configuration	Query of IP address	Bootstrapping
1x RN-134 (UHF antenna)	471 ms	6649 ms
1x RN-134 (bare wire antenna)	480 ms	6620 ms
5x RN-134 (bare wire antenna)	728 ms	10322 ms
1x ALN-9662 RFID tag	455 ms	6288 ms
5x ALN-9662 RFID tag	656 ms	10268 ms

Table 4.5: Evaluation results for communication over the UHF channel. Two application scenarios were considered. The results show the average time required per scenario per node over 100 runs.

are sent to the reader. This large number of requests is mostly caused by the fact that we could only perform write operations of two bytes at a time when using the RN-134. For each of the four commands, the node was given a delay of 200 ms in order to process the request. Note that all the payload data written to the node were validated by our application by reading it out again and comparing it to the desired value. For both scenarios, write and read operations were repeated when they were unsuccessful four times at most.

Both usage scenarios were tested with a single node as well as with five nodes on the reader antenna. In each case, all the nodes were equipped with the bare wire antenna. Additionally, we tested a single node on the reader that was equipped with the professional UHF antenna. As a baseline, we also tested with an ALN-9662 RFID tag [255] that featured 512 bits of user memory area. Since the RFID tag cannot provide the responses required by our communication protocol, we ignored the missing answer codes for these tests. All measurement results were averaged over 100 runs. In all of the tests performed, our Java program first detected available nodes or tags and the performed the selected usage scenario on each node that was discovered.

Results and Discussion The results of our evaluation are summarized in Tab. 4.5. Overall, the measured execution times may seem higher than expected, when considering the small size of payload transferred (4 bytes and 118 bytes). The comparison between using an RN-134 and an RFID tag reveals that despite that fact that we were using an improvised antenna setup, comparable performance was achieved. This also indicates that much of the overhead can be attributed to our implementation and the LLRP communication layer, which currently executes a write operation for every 16 bits of data. Furthermore, reading or writing data to the tag currently requires at least three LLRP

commands to be sent to the reader. Finally, for each command, such as the definition of the sensor callback URL, the node is given a processing time of 200 ms before our software queries for the result. For the bootstrapping scenario, this means that at least $4 * 200\ ms = 800\ ms$ are spent waiting. For these reasons, we are optimistic that the net transfer rate (the *goodput*) can be optimized.

However, from a practical standpoint, we argue that the required execution times are already sufficient. As outlined in Sect. 4.3.7.1, the secondary communication channel only serves as a fallback channel for special purposes or situations. Bootstrapping a node in about 10 seconds seems acceptable, since this operation is usually performed rarely, and other solutions are probably cumbersome, as they require manual intervention. A single command, such as querying the IP address of the node, requires less than one second, which should be sufficient for most use cases. Transferring larger amounts of data from the reader to the node, such as sending a firmware update, is currently impractical with our implementation.

It is interesting to see that the times required for a scenario and a tag type differ significantly, in relation to the number of elements placed on the reader antenna. We attribute this to the amount of metal that is placed on the reader and interferes with the RF field.

We also experimented with several other use cases for the UHF communication channel. For example, our memory-based communication protocol offers a status field that indicates whether the node is missing some of the crucial bootstrapping parameters. We implemented a function in our Java program that automatically detects nodes that lack such parameters and updates these accordingly. This enables automatic bootstrapping: Once nodes are placed next to the reader antenna, they are automatically configured. Note that this works continuously – as soon as a unconfigured node is discovered, it is updated and can then in turn connect to the Wi-Fi.

Besides the auto-configuration, we also found it practical to be able to manually trigger the execution of a heartbeat[36], from which it would then get an updated configuration from the callback target. Similarly, waking the node up and putting it back to sleep could be performed over the UHF channel. If the node is awake, one can ping it and access

[36]Heartbeats are usually sent rarely (e.g., once per hour or once per day). The execution of a heartbeat ahead of schedule can therefore significantly reduce the time required to reconfigure the node.

its Web server, which is sometimes helpful. The identification of the application running on a node, its current IP configuration and the ability to perform a reset (a power cycle) over the UHF channel also proved to be helpful.

Finally, we also briefly tested the UHF communication channel with an Impinj xPortal RFID reader [256], which has integrated and more powerful antennas. Using a node configured with the UHF antenna, we were able to communicate with a single node over a distance of several meters. This enables interesting use cases that could not be implemented with NFC, which is limited to very short distances.

4.5.5 Exemplary Applications

Wireless Doorbell System As noted in Sect. 4.4.2, we modified a wireless doorbell system, replacing the original communication hardware in the switch and the doorbell with an RN-134. Both nodes could be bootstrapped using the UHF interface and were assigned a static DNS address by registering with the NO-IP[37] service. Besides the mimicry of the original functionality (pressing the switch rings the doorbell) using our simple interaction model, we also implemented different, application-specific scenarios.

For the doorbell switch, we tested two similar scenarios, in which it was used with existing Web services that were called upon a press of the button. This is an example of the *event-based sensing* scheme.

In the first scenario, the doorbell switch was programmed to send a push notification to a smartphone. For this, we leveraged Prowl[38], which enables generic notifications to be sent to iOS devices. It consists of an iOS application and an HTTP service with a simple API. Calling the Prowl API from the program running on the node was a simple task. The delay between the press of the button and the display of the push notification on the phone was usually in the range of a couple of seconds. However, this delay can mainly be attributed to the time required by Prowl's service and the underlying push notification system from Apple.

In a similar scenario, we used the doorbell switch to send an email to a prespecified address as soon as the button was pressed. For this, we used the Web services of MailGun[39], which provides a convenient HTTP

[37] http://www.no-ip.org
[38] http://www.prowlapp.com
[39] http://www.mailgun.com

interface for sending emails. The delay between pressing the doorbell button and receiving the corresponding email was usually slightly larger than when using push notifications. This can be explained by the fact that the email system consists of multiple servers from different authorities, which handle the transmission of emails in a store-and-forward manner.

For the doorbell, we tested an example of a *polling actuator* (see Sect. 4.4.1.4). Since the doorbell is running on batteries, running a Web server would result in a short lifetime. Instead, we configure it to wake up periodically and poll a pre-specified HTTP resource for changes. To do so, we leverage HTTP's conditional `GET` feature, which is usually used for caching and widely supported on the Web. The doorbell is a good example of such a polling actuator, as both its actuators (a bell and a flashing light) require power only when an event is to be signaled. We configured the doorbell to periodically monitor the URL of the RSS feed of an existing blog. As soon as a new article was posted on the blog, its RSS feed was updated by the provider of the blog. When the doorbell detected the change, it rang once to notify people in its vicinity of the update. For this scenario, the notification delay was considered less important than battery lifetime, so we configured the node to poll the resource every 15 minutes.

In all of the scenarios implemented and tested in this section, using the UHF interface for bootstrapping and configuration purposes proved helpful, as both the doorbell and the switch had only very limited means of user input (some switches and a button). Also, monitoring the nodes using heartbeat messages proved to be an important tool to detect problems such as low battery voltage or bad Wi-Fi reception early. Leveraging the existing Web services of NO-IP, Prowl, and Mailgun proved to be simple and served as another example of the benefits of leveraging the existing Web infrastructure.

Mobile Nodes We also briefly tested the feasibility of mobile nodes in a large Wi-Fi infrastructure consisting of multiple access points all sharing the same SSID. For this, we utilized the campus-wide wireless infrastructure of ETH Zurich, which provides open access to the intranet[40]. The node was configured to perform a callback cycle every 10 seconds, sending a heartbeat to the same script as in previous evaluations, running on a Web server of our department. Several walks at

[40]For a full access to the Internet, the users need to authenticate.

different parts of the ETH campus were conducted, carrying the node along. The setup worked as expected: As long as there was Wi-Fi coverage, the callbacks were successfully executed. When a location change of the node resulted in a loss of the previously contacted access point, the first callback after such a location change required significantly more time. This was as expected, since all of our optimizations of the callback cycle could not take effect when the access point changed. Since the existing Wi-Fi infrastructure can also be used for indoor localization [230], one could, for example, monitor the conditions and whereabouts of expensive equipment[41].

Monitoring and Configuration of Deployed Nodes Monitoring and reconfiguring deployed nodes using the concept of heartbeat callbacks as introduced in Sect. 4.3.6 proved to be a vital tool. Because the payload was encoded in JSON (see List. 4.1), parsing the data could be achieved easily using readily available JSON libraries. We utilized a simple PHP script that served as a heartbeat callback target and persisted the received heartbeat payloads.

In order create an overview of the status of all nodes, we used another PHP script that extracted key metrics from the heartbeat data of each node and dynamically generated an HTML page rendering such data for all known nodes. Metrics included battery voltage, last RSSI value, number of wake events, last uptime, and average uptime. This overview was very helpful in detecting nodes early that faced some sort of problem. Additionally, recorded heartbeat data could be analyzed to detect trends or patterns.

Reconfiguration of nodes was usually performed manually by storing settings to be updated on the server for each node. As configuration updates were delivered in the response to a heartbeat request, it may require in the worst case the time of a full heartbeat period in order to take effect. During our tests, we usually set the heartbeat period to one hour, which was a sufficient trade-off between battery lifetime, temporal resolution of collected data, and responsiveness to configuration updates. An example of a heartbeat response that includes configuration updates is depicted in List. 4.2. Examples of configuration updates that proved to be valuable include changing the sensor callback URL, the heartbeat callback URL, the Wi-Fi configuration, the LED

[41] Of course, this could also be achieved using the RFID interface of the node. However, a Wi-Fi infrastructure is usually already in place, while an UHF infrastructure is not.

output[42], and the firmware on the node.

To a small extent, we also tested changing the configuration of the nodes automatically. For example, to evaluate the influence of the communication rate on the duration of the callback cycle (see Sect. 4.5.2.1), the script on the server automatically cycled the preferred communication rate of the client by including it in the corresponding heartbeat responses. Note that there is considerable potential in automatic configuration changes. For example, the node could be automatically configured to save energy (e.g., by disabling its LEDs and increasing the heartbeat period) as soon as it runs low on batteries. Also note that this approach is not limited to data acquired from the node but can also consider data available on the Web.

In contrast to all the other configuration changes, which take effect immediately, upgrading the firmware is a three-stage process. First, the node is notified that it should perform a firmware upgrade by returning the command and a URL in the heartbeat response. Based on this URL, the node constructs two URLs by appending different well-known file names (on this platform, a firmware upgrade consists of two separate files). The node then downloads both files using HTTP, activates the new firmware, and reboots. Since a program on the node will usually send a heartbeat message after a reboot, we are immediately notified if the upgrade was successful. Using HTTP for firmware upgrades greatly simplified this process – one simply needs to copy the two parts of the new firmware into a directory that is served by a Web server, and set a configuration update for the node. Since the RN-131 supports the relatively high communication rates of 802.11g, downloading the firmware was a fast operation, despite its size, which can easily reach 64 KB.

4.6 Related Work

We categorized approaches of connecting constrained devices to the Web into three different concepts in Sect. 4.1.3, outlining both benefits and drawbacks. For each of the concepts, we give an overview of related work.

[42]LED output serves as a visual clue both for the performance of the node (e.g., time required for a callback cycle) and for possible problems, such as failure to associate with an access point. However, as the energy required by the LEDs has an impact on battery lifetime, they were usually disabled during deployments.

4.6.1 Application-Specific Gateways

Providing a Web interface to wireless devices via an application-specific gateway is a well-known approach for which we provided examples of related work in Sect. 4.1.3.1.

4.6.2 Application-Agnostic Gateways

Application-agnostic gateways are proxy servers that provide an automatic mapping between HTTP and the protocols used on the resource-constrained domain. They do not require an application-specific configuration and may therefore offer their services to devices running various applications.

Using such gateways for connecting resource-constrained devices to the Web was proposed in 2009 under the name Compressed HTTP Over PANs (Chopan) [257] in the context of IEEE 802.15.4 networks. A similar approach named Embedded Binary HTTP (EBHTTP) [258] was proposed in 2010. Both approaches pursue the idea of an efficient binary transmission format for HTTP and proxies that would translate between this novel format and the original HTTP. Originating from these approaches is the Constrained Application Protocol (CoAP) [69]. CoAP is a binary application protocol for resource-constrained devices that follows the REST principle [65]. Its functionality also includes publish/subscribe. Since CoAP is currently the most popular approach for connecting resource-constrained devices to the Web, we will compare it in more detail to our approach in Sect. 4.6.5.

Note that an automatic mapping between CoAP and HTTP does not address the issue of efficient payloads. Ideally, the gateway could automatically translate between the verbose representations usually utilized on the Web (XML, JSON) and a more efficient representation for the resource-constrained domain. Efforts in this area include the Efficient XML Interchange (EXI) format [218], which is a binary representation for XML, and the Concise Binary Object Representation (CBOR) [219], which enables the representation of JSON data in a compact binary serialization.

For Bluetooth, there are efforts to standardize a mapping for the GATT profile to an HTTP REST API that can be implemented by application-agnostic gateways [259].

4.6.3 Direct Connection to the Web

The idea of connecting resource-constrained devices directly to the Web by running an HTTP server and/or an HTTP client on such devices captured academic interest early on. An early example is the implementation of a Web server in a Java-programmable SmartCard in 1999 [260].

Using this approach in order to sense and manipulate the physical world is specifically addressed in an article from Borriello and Want in 2000 [81]: *"Distributed sensors and actuators connected through the Web's standard protocols and wireless communication media provide a powerful toolkit for developing rich applications affecting all our lives, [...]"* Interestingly, the authors also specifically address the usefulness of binary sensors and actuators: *"[...] At the other end of the Web-server spectrum is the so-called Boolean server, whose sole purpose is to turn on a single bit (in order to, say, turn a light bulb on or off or to sense the state of an electrical switch, perhaps as part of a security system). At the physical-interface side of the embedded Web server implementing bit-level control, or sensing, the system is very simple. But on the network side, the Web server has to use the same protocols as any more advanced server."* In 2001, Dunkels presented an implementation of TCP/IP with a small footprint. In his thesis, he briefly mentions that it was used to run a simple Web server on an 8-bit microprocessor [54].

In the context of IEEE 802.15.4, the authors of [261] use HTTP in order to call REST-based as well as SOAP-based Web services running on sensor nodes. Their results indicate that most calls can be processed in less than one second, in single-hop scenarios.

4.6.4 Other Approaches based on IEEE 802.11

Tozlu et al. also use the Microchip/Roving platform for their work. In several publications, they analyze the influence of security features and different communication rates on battery lifetime and consider the impact of interference and range on the performance [235, 262, 263, 264]. In contrast to our approach, they use UDP for communication and the power save mode of IEEE 802.11. This mode allows the node to stay associated with the access point when it is in sleep mode but requires the node to periodically wake up and listen for data buffered by the access point. Additionally, keep-alive messages are required in

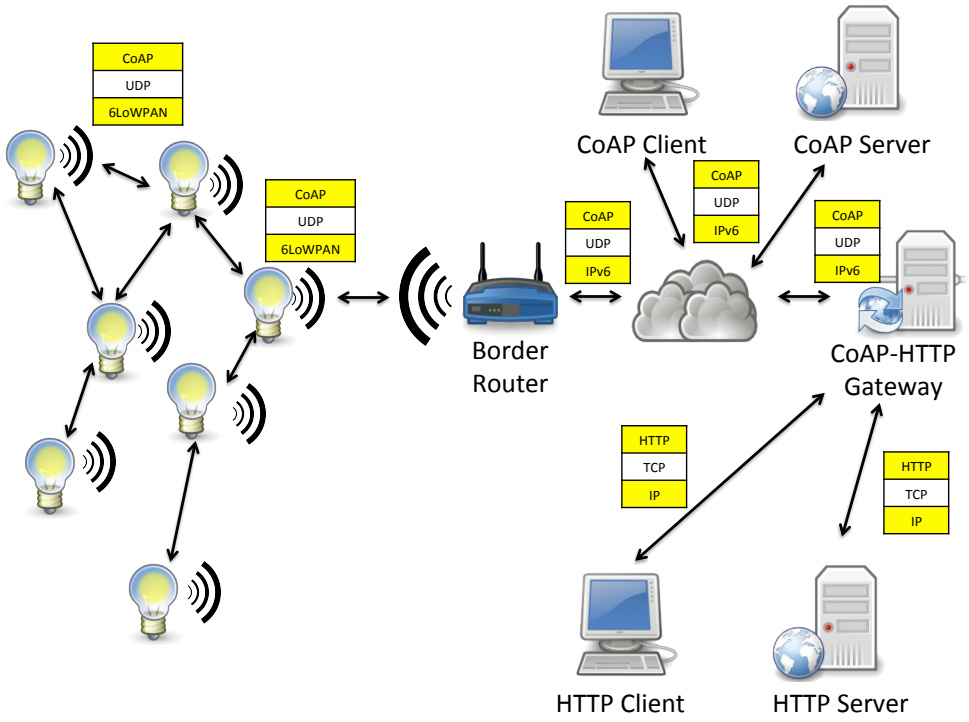


Figure 4.20: Using CoAP to connect physical entities to the Web (icons provided by [213]).

order to prevent de-association by the access point.

4.6.5 Comparison with CoAP over IEEE 802.15.4

An approach to foster interoperability that is currently actively pursued in the research community is based on the IEEE 802.15.4 standard [265], which specifies a low-power, low-bitrate, and reduced-complexity communication protocol intended to enable low-cost transceivers. IEEE 802.15.4 transceivers have been used intensely in wireless sensor nodes. Popular research platforms are the TelosB [17] and the Tmote Sky [18], which feature the MSP430 [266], an ultra-low-power microcontroller, in combination with the CC2420 [267] transceiver.

IEEE 802.15.4 allows for a maximum transmission rate of 250 kbit/s and limits the maximum size of the payload of the physical layer to 127 bytes [265]. Running TCP/IP over the link leaves a payload of about 62 bytes in the best case (TCP/IPv4, no security) [268] down to only 21 bytes in the worst case (TCP/IPv6, with AES-CCM-128)[43]. This payload could then be utilized by HTTP and the application. As HTTP is a verbose protocol, transferring only small amounts of application

[43] According to [269], replacing the 8 byte UDP header with a 20 byte TCP header.

data would most probably require fragmentation of packets.

For this reasons, the practicability of using HTTP over TCP/IP is limited on networks based on IEEE 802.15.4, as the overhead jeopardizes the energy-optimized design and leads to significant response times [270, 261]. A popular approach is to optimize all network layers and to use dedicated gateways to be able to connect nodes to the Internet and in turn to the Web.

As wireless transceivers consume a significant amount of energy not only when sending but also when listening for incoming data, radio duty cycling mechanisms are utilized, which limit the amount of time the transceiver is powered. This of course requires coordination with adjacent network nodes for them to still be able to reliably transmit data. For this, optimized MAC protocols are utilized at the *link layer* (e.g., [61, 271]).

The *network layer* is based on IPv6. An IPv6 packet has a header of 40 bytes, which is mostly attributed to the large address space[44] and requires a supported maximum transmission unit (MTU)[45] of at least 1280 bytes [59]. In order to make IPv6 practical for low-power wireless area networks, 6LoWPAN [269, 272] is utilized to transport IPv6 packets, which provides, among other features, header compression and automatic fragmentation and reassembly. This in turn requires dedicated *border routers* (also called *edge routers*), which connect a 6LoWPAN network with a standard IPv6 network [273].

At the *application layer*, CoAP, the Constrained Application Protocol [69, 70, 71] running on top of UDP is used instead of HTTP over TCP. CoAP is a binary protocol intended for communication with resource-constrained devices that follows the REST architectural style. It has a strong resemblance to HTTP but provides additional features such as support for subscriptions to changes of resources. Interestingly, there is also support for CoAP to use in a Web browser [274]. Considering the payload, standard compression techniques such as gzip may not provide satisfying results [73]. For XML documents, the use of the efficient XML interchange (EXI) format [218] is currently being considered [73]. JSON documents could be automatically converted to the more efficient CBOR serialization [219]. In order to be able to interact with HTTP, it is suggested to use a stateless CoAP/HTTP-Gateway that is agnostic to the application [217]. This scenario is depicted in Fig. 4.20.

[44] Source and destination addresses sum up to 32 bytes.

[45] Link MTU: Maximum packet size in octets, that can be conveyed over a link [59].

To summarize, this is a *bottom-up* approach that is optimized primarily for energy efficiency. In order to operate with existing Web services, it requires various optimizations at several layers of the protocol stack to address the limitations mostly imposed by IEEE 802.15.4. In contrast, the approach that we pursued in this chapter is primarily optimized for interoperability with existing Web services. It is a *top-down* approach that leverages the existing Wi-Fi and Web infrastructures. In order to achieve low-power operation, we rely on special features of the utilized hardware platform, restrict the direction of communication (when running on batteries), and minimize the time the node spends awake.

There are several attempts that compare HTTP and CoAP over IEEE 802.15.4. A good introduction to the topic is given in [275]. [276] provides a simulative evaluation of the performance of HTTP and CoAP based on IEEE 802.15.4. Colitti et al. compare CoAP and HTTP both simulatively and experimentally [277, 278]. Pötsch et al. provide limited practical measurements [74]. In [72], extensive results based on a practical evaluation of CoAP are provided. Finally, [61] provides an extensive evaluation of the performance of UDP and TCP on 802.15.4.

In [235], Tozlu compares the use of UDP over low-power Wi-Fi with UDP over 6LoWPAN. His work is based on the same Wi-Fi platform as used in this thesis. He considers the time and energy required for a node to wake up and send a single UDP datagram, with respect to data rate and packet size. His results are depicted in Tab. 4.6 for packet sizes of 8 bytes and in Tab. 4.7 for packet sizes of 1024 bytes. Interestingly, ultra-low-power Wi-Fi requires less energy than 6LoWPAN, especially when using a high communication rate. In contrast to our callback cycle (Sect. 4.4.1.1), the nodes used the power-saving features of 802.11, which allow them to stay associated to an access point despite resolving to sleep mode. In the setup of Tozlu, the data rate has a significant impact, on both the time and the energy, which is in contrast to our approach (see Sect. 4.5.2.1). This can be explained by the fact that our callback cycle involves the sending and receiving of multiple packets, thus being affected more by the response times of communication parties than by the communication rate, for small payloads. The increased complexity also explains that our approach requires about 5.4x more time and 14.1x more energy (see Tab. 4.2).

Finally, a comparison between CoAP over 802.15.4 and our approach of using HTTP over ultra-low-power Wi-Fi is summarized in Tab. 4.8.

	6LoWPAN	ULP Wi-Fi @1Mbit/s	ULP Wi-Fi @54Mbit/s
Time (ms)	6	12.48	11.3
Energy (mJ)	2.5	1.30	0.55

Table 4.6: Comparison of 6LoWPAN and ULP Wi-Fi for sending a single UDP datagram of 8 bytes (source: [235]).

	6LoWPAN	ULP Wi-Fi @1Mbit/s	ULP Wi-Fi @54Mbit/s
Time (ms)	23.61	25.82	16.58
Energy (mJ)	9.17	8.46	1.28

Table 4.7: Comparison of 6LoWPAN and ULP Wi-Fi for sending a single UDP datagram of 1024 bytes (source: [235]).

4.7 Summary

In this chapter, we investigated how to connect things to the Web by leveraging existing infrastructure and standards. Instead of relying on dedicated low-power radio technology and specialized network-level or application-level protocols, we relied on the ubiquity of IEEE 802.11 and the interoperability of the HTTP protocol. This obsoletes dedicated access points and application-level gateways and in turn reduces complexity, enables mobility, simplifies interoperability, and therefore fosters a wide and rapid deployment of a Web of Things.

Our results show that this approach can be implemented in an energy-efficient manner using ultra-low-power IEEE 802.11 transceivers.

—	CoAP over 6LoWPAN over IEEE 802.15.4	HTTP over ultra-low-power IEEE 802.11b/g
Physical and link layer	IEEE 802.15.4	IEEE 802.11b/g
Frequency band(s)	868 MHz, 902 MHz, 2.4 GHz	2.4 GHz
Brutto data rates	Up to 250 kbit/s (2.4 GHz band)	Up to 54 Mbit/s
Reliability	ACKs & automatic retransmissions	ACKs & automatic retransmissions
MTU	127 bytes	2312 bytes
Encryption	AES 128 bits (other possible)	AES 256 bits (other possible)
Network protocol	6LoWPAN	IPv4 (IPv6 also possible)
Network topology	Mesh, Star	Star
Transport protocol	UDP (with retransmits)	TCP
End-to-end security	DTLS, optional	TLS, optional
Application protocol	CoAP	HTTP
Protocol format	Binary	Plain text
Subscriptions and notifications	Supported	Limited (sensor callbacks)
Connection to the Internet	Requires 6LoWPAN border router	Standard Wi-Fi router
Interoperability with existing Web services	Requires proxy. Payloads should be kept small.	Direct. Supports large payloads.
Infrastructure	Not readily available	Widely deployed
Reachability	Reachable: Global IP address, nodes are constantly online	Not reachable: (1) IPv4 usually behind NAT, (2) nodes wake up only for certain events
Remote configuration updates	Real-time	Delayed (requires polling)
Actuation	Real-time	Delayed (requires polling)
Sensing	Real-time	Real-time
Running server on batteries	Supported	Limited

Table 4.8: CoAP/6LoWPAN vs. our approach; both battery powered.

5 Conclusion

During the course of this thesis project, the Web continued its impressive development. For example, the combination of multiple data sources into so-called mash-ups became popular, the REST paradigm overtook RPC-style SOAP APIs, several new platforms for collecting and sharing sensor data emerged, and standards were improved or appended[1]. Today, utilizing the Web architecture as an interface for the physical world can be considered an accepted paradigm. An increasing number of products are connected to the Internet and offer an interface on the Web, usually both for users as well as for third-party applications. This makes it much simpler for application developers to create novel application scenarios, based on the numerous APIs available on the Web.

However, these APIs, while simple, still feature a high level of heterogeneity. This results in unnecessary manual effort and thus impedes *interoperability*. Driven by the lack of accepted standards for a global Web of Things, a current trend is to connect physical entities to central hubs that provide a uniform API for all of their connected physical entities, creating vertical *silos*. While this may increase interoperability for entities within such silos, it does not for devices from different silos, as these may offer their specific APIs. This development is also concerning since such hubs are usually run under a single authoritative domain, as opposed to the open and decentralized architecture of the Web, which empowers users and makes it so successful. In this thesis, it is argued that a Web for Things should be just as open and decentralized as the World Wide Web has been from its inception. All of our contributions demonstrate that this is technically feasible – addressing different aspects of a future Web of Things.

[1] Such as HTML5 and its various JavaScript APIs, CSS3, and WebSockets.

5.1 Contributions

This thesis provided three major contributions toward a future Web of Things, each covering a different aspect of interoperability. All of our contributions leverage the existing Web architecture and avoid central hubs to make the Web of Things an open and decentralized platform. The contributions can be summarized as follows:

- The development and evaluation of a prototypical *real-time search engine for the physical world*, based on the Web architecture. In contrast to traditional Web search engines, a search engine for the physical world has to support searching for structured and rapidly changing, distributed content in real time. Contrary to existing approaches, our solution is based on an open architecture that requires neither a global view of the world's state nor a limitation of the search space, while still providing accurate results in real time.

- A prototypical *framework for the Web of Things* that simplifies the connection of sensors and actuators to the Web as well as their composition to novel services by providing key primitives identified during the development of several experimental applications. In contrast to existing solutions, our framework does not serve as a central hub but strives to enhance today's inherently decentralized Web architecture.

- A concept and evaluation of connecting everyday objects to the Web using programmable low-power Wi-Fi modules. Contrary to existing approaches that rely on dedicated low-power radio technology and specialized protocols, we leverage the ubiquity of IEEE 802.11 access points and the interoperability of the HTTP protocol. Our experimental results show that low-power Wi-Fi modules can achieve long battery lifetime despite using the verbose Web protocols for communication.

While our contributions provide a step toward an open, decentralized and interoperable Web of Things, there is still a lot of work remaining in order to reach this goal.

5.2 Limitations and Future Work

Naturally, not all of the issues we faced during the course of this thesis could be addressed.

For our search engine, additional evaluations on even larger data sets would be beneficial, as well as a deployment with live sensors and actual users.

For our framework, a large deployment featuring more complex application scenarios could show whether our distributed, event-driven approach scales with respect to the complexity of the application scenarios.

For the work on low-power Wi-Fi modules, a deployment of mobile nodes that sense predefined events and opportunistically use Wi-Fi access points to upload their recorded data to existing Web services, be it in real-time or as a bulk upload, could provide valuable insights. As a plus, the position of the nodes could be determined approximately by using a Wi-Fi geolocation service.

Besides improvements to the existing work, there are also some new directions for future work:

Adding Semantics to the Web of Things The Semantic Web [88] is a promising approach that strives to add semantic annotations to the Web in order to make data universally understandable for applications. Unfortunately, this simple and early concept has still not found widespread adoption in today's Web. Semantic (self-)descriptions of capabilities and properties of physical objects may be the missing link that is required for true interoperability. It could enable heterogeneous devices to automatically operate with other devices and services in novel ways. In this regard, the Web of Things may be the catalyzer for the Semantic Web, creating a Semantic Web of Things [89, 87].

Extending the Web down to Wireless Energy-Harvesting Devices In this thesis, we demonstrated that extending the Web down to resource-constrained ultra-low-power Wi-Fi modules is beneficial for certain application scenarios. However, despite providing sufficient battery lifetime for event-based sensing, batteries need to be changed eventually. When considering a scenario in which a large number of such nodes is deployed, this can require significant manual effort. Combining low-power Wi-Fi modules with an energy harvesting device such as

a photovoltaic cell and a supercapacitor may provide enough energy for maintenance-free operation while still offering the benefits of leveraging existing infrastructures. Using energy harvesting with low-power Wi-Fi modules is challenging, as these require significantly more energy than current approaches of energy-harvesting wireless devices, which use dedicated low-power radio links and protocols [279, 280, 281].

Considering other transport layers than TCP for HTTP In Chap. 4, we demonstrated how Internet-connected devices benefit from a secondary communication channel. The protocol we implemented for the communication between the UHF reader and the Wi-Fi module is based on request/response cycles, just like HTTP. It is obvious that one could also have used HTTP over this link, which would enable much of the existing code of the node's Web server to be re-used. Unfortunately, HTTP was much too verbose for our implementation of a shared memory protocol. However, there are efforts that successfully run HTTP over NFC [282]. Running HTTP on top of UDP (denoted as HTTPU) is also potentially interesting for certain use cases, as it supports broadcast and multicast and features less overhead than TCP. HTTPU was proposed early and is currently used within UPnP [193, 283]. In the context of a global Web of Things, revisiting this approach could be promising, such as for handling notifications.

Spontaneous and Opportunistic Use of Physical Resources In a future Web of Things, applications could opportunistically make use of sensors and actuators to perform a certain task. For example, in order to determine the temperature in a room, all devices that are currently located in that room and feature a temperature sensor could contribute. In the field of sensor networks, *opportunistic sensing* is well known and often used with mobile devices such as smartphones [284]. However, in the Web of Things, the quantity and heterogeneity of devices will be much larger than in such closed sensing applications, and there is also the possibility of *opportunistic actuating*. One simple example would be a universal light switch that could be implemented as a key tag and that switches all the lights in the room where it is currently located on or off. When combined with semantic approaches, one could also realize higher abstractions such as "make my surroundings quiet".

Standards for the Web of Things There are currently multiple efforts to standardize parts of the Internet of Things [285, 286, 287, 288]. Regarding the Web architecture, the new HTTP/2 standard [289] missed an opportunity to include support for building blocks for a Web of Things, such as publish/subscribe, notifications, and discovery. Standardization efforts focused on protocol efficiency, not on functionality. If the further development of HTTP does not address the requirements of a future Web of Things, a fragmentation into several application-level protocols such as CoAP [69, 70, 71] seems inevitable.

Bibliography

[1] Friedemann Mattern. Ubiquitous Computing. Lecture slides, ETH Zurich, Zurich, Switzerland, 2015. Available from: http://www.vs.inf.ethz.ch/edu/FS2015/UC/slides/01-Intro-Vision.pdf.

[2] Neil A. Gershenfeld. *When Things Start to Think*. Henry Holt and Company, 1999.

[3] Gordon Moore. Cramming More Components onto Integrated Circuits. *Electronics Magazine*, 38(8):114–117, April 1965.

[4] Markus Weiss, Adrian Helfenstein, Friedemann Mattern, and Thorsten Staake. Leveraging smart meter data to recognize home appliances. In *Proceedings of the 10th IEEE International Conference on Pervasive Computing and Communications (PerCom 2012)*, pages 190–197, Lugano, Switzerland, March 2012.

[5] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):66–75, September 1991.

[6] Roy Want, Andy Hopper, Veronica Falcão, and Jonathan Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems (TOIS)*, 10(1):91–102, January 1992.

[7] Andy Harter and Andy Hopper. A Distributed Location System for the Active Office. *IEEE Network*, 8(1):62–70, January/February 1994.

[8] Michael Beigl, Hans-W. Gellersen, and Albrecht Schmidt. MediaCups: Experience with Design and Use of Computer-Augmented Everyday Artefacts. *Computer Networks*, 35(4):401–409, March 2001.

[9] Michael Beigl, Tobias Zimmer, and Christian Decker. A Location Model for Communicating and Processing of Context. *Personal and Ubiquitous Computing*, 6(5-6):341–357, December 2002.

[10] Jon E. Froehlich, Eric Larson, Tim Campbell, Conor Haggerty, James Fogarty, and Shwetak N. Patel. HydroSense: Infrastructure-Mediated Single-Point Sensing of Whole-Home Water Activity. In *Proceedings of the 11th International Conference on Ubiquitous Computing (UbiComp 2009)*, pages 235–244, Orlando, FL, USA, September 2009. ACM.

[11] Sidhant Gupta, Matthew S. Reynolds, and Shwetak N. Patel. ElectriSense: Single-point Sensing Using EMI for Electrical Event Detection and Classification in the Home. In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing (UbiComp 2010)*, pages 139–148, Copenhagen, Denmark, September 2010. ACM.

[12] Javier Hernandez, Mohammed (Ehsan) Hoque, Will Drevo, and Rosalind W. Picard. Mood Meter: Counting Smiles in the Wild. In *Proceedings of the 14th ACM International Conference on Ubiquitous Computing (UbiComp 2012)*, pages 301–310, Pittsburgh, PA, USA, 2012. ACM.

[13] Xuan Bao, Songchun Fan, Alexander Varshavsky, Kevin Li, and Romit Roy Choudhury. Your Reactions Suggest You Liked the Movie: Automatic Content Rating via Reaction Sensing. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp 2013)*, pages 197–206, Zurich, Switzerland, 2013. ACM.

[14] Rodrigo Fonseca, Omprakash Gnawali, Kyle Jamieson, Sukun Kim, Philip Levis, and Alec Woo. The Collection Tree Protocol (CTP). TinyOS Enhancement Proposal 123, February 2007. Available from: http://www.tinyos.net/tinyos-2.x/doc/pdf/tep123.pdf.

[15] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '02)*, pages 88–97, Atlanta, GA, USA, September 2002. ACM.

[16] Jason Hill and David Culler. A wireless embedded sensor ar-

chitecture for system-level optimization. Technical report, UC Berkeley, 2002.

[17] Crossbow. TelosB. Data sheet, January 2006. Available from: http://www.willow.co.uk/TelosB_Datasheet.pdf.

[18] Moteiv Corporation. Tmote Sky. Data sheet, June 2006. Available from: http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf.

[19] BTnode rev3 – Product Brief, March 2006. Available from: http://www.btnode.ethz.ch/pub/uploads/Documentation/btnode_rev3.24_productbrief.pdf.

[20] Geoffrey Werner-Allen, Konrad Lorincz, Mario Ruiz, Omar Marcillo, Jeff Johnson, Jonathan Lees, and Matt Welsh. Deploying a Wireless Sensor Network on an Active Volcano. *IEEE Internet Computing*, 10(2):18–25, March-April 2006.

[21] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steven Glaser, and Martin Turon. Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN 2007)*, pages 254–263. IEEE, ACM, April 2007.

[22] Gyula Simon, Miklós Maróti, Ákos Lédeczi, György Balogh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, and Ken Frampton. Sensor Network-based Countersniper System. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, pages 1–12, Baltimore, MD, USA, November 2004. ACM.

[23] Emiliano Miluzzo, Nicholas D Lane, Kristóf Fodor, Ronald Peterson, Hong Lu, Mirco Musolesi, Shane B Eisenman, Xiao Zheng, and Andrew T Campbell. Sensing Meets Mobile Social Networks: The Design, Implementation and Evaluation of the CenceMe Application. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys '08)*, pages 337–350, Raleigh, NC, USA, November 2008. ACM.

[24] Eiman Kanjo, Steve Benford, Mark Paxton, Alan Chamberlain, Danae Stanton Fraser, Dawn Woodgate, David Crellin, and

Adrain Woolard. MobGeoSen: Facilitating Personal Geosensor Data Collection and Visualization Using Mobile Phones. *Personal and Ubiquitous Computing*, 12(8):599–607, November 2008. Available from: http://dx.doi.org/10.1007/s00779-007-0180-1.

[25] Andong Zhan, Marcus Chang, Yin Chen, and Andreas Terzis. Accurate Caloric Expenditure of Bicyclists Using Cellphones. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems (SenSys '12)*, pages 71–84, Toronto, Canada, November 2012. ACM.

[26] Chengwen Luo and Mun Choon Chan. SocialWeaver: Collaborative Inference of Human Conversation Networks Using Smartphones. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems (SenSys '13)*, pages 20:1–20:14, Rome, Italy, November 2013. ACM.

[27] Martin H. Weik. A Fourth Survey of Domestic Electronic Digital Computing Systems. Technical Report 1115, Ballistic Research Laboratories, Aberdeen Proving Ground, Maryland, January 1964. Available from: http://ed-thelen.org/comp-hist/BRL64.html.

[28] James E. Tomayko. Computers in Spaceflight: The NASA Experience. NASA Contractor Report 182505, March 1988. Available from: http://history.nasa.gov/computers/Compspace.html.

[29] IC Insights. MCU Market on Migration Path to 32-bit and ARM-based Devices, April 2013. Available from: http://www.icinsights.com/data/articles/documents/541.pdf.

[30] Ragunathan (Raj) Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-Physical Systems: The Next Computing Revolution. In *Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC)*, pages 731–736, Anaheim, CA, USA, June 2010. IEEE.

[31] CPS Steering Group. Cyber-Physical Systems - Executive Summary, March 2008. Available from: http://precise.seas.upenn.edu/events/iccps11/_doc/CPS-Executive-Summary.pdf.

[32] Edward Ashford Lee and Sanjit A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. LeeSeshia.org, 2011. Available from: http://leeseshia.org/releases/LeeSeshia_DigitalV1_08.pdf.

[33] John A. Stankovic, Insup Lee, Aloysius Mok, and Raj Rajkumar. Opportunities and Obligations for Physical Computing Systems. *Computer*, 38(11):23–31, November 2005.

[34] John Eidson, Edward A. Lee, Slobodan Matic, Sanjit A. Seshia, and Jia Zou. Distributed Real-Time Software for Cyber-Physical Systems. *Proceedings of the IEEE*, 100(1):45–59, January 2012.

[35] Miroslav Pajic and Rahul Mangharam. Embedded Virtual Machines for Robust Wireless Control and Actuation. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 79–88, Stockholm, Sweden, April 2010. IEEE.

[36] Jonathan Fink, Alejandro Ribeiro, and Vijay Kumar. Robust Control for Mobility and Wireless Communication in Cyber-Physical Systems With Application to Robot Teams. *Proceedings of the IEEE*, 100(1):164–178, January 2012.

[37] Patricia Derler, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Modeling Cyber-Physical Systems. *Proceedings of the IEEE*, 100(1):13–28, January 2012.

[38] Yilin Mo, Tiffany Hyun-Jin Kim, Kenneth Brancik, Dona Dickinson, Heejo Lee, Adrian Perrig, and Bruno Sinopoli. Cyber-Physical Security of a Smart Grid Infrastructure. *Proceedings of the IEEE*, 100(1):195–209, January 2012.

[39] Siemens AG, Munich, Germany. *SIEMENS GSM Module M1 User Guide*, 1996.

[40] Friedemann Mattern. Die technische Basis für M2M und das Internet der Dinge. In Jörg Eberspächer and Uwe Kubach, editors, *M2M und das Internet der Dinge. Münchner Kreis*, pages 46–69, May 2013. Available from: http://www.vs.inf.ethz.ch/publ/papers/mattern-m2m_iot-2013.pdf.

[41] Bundesministerium für Wirtschaft und Technologie. Machine-to-Machine Kommunikation – eine Chance für die deutsche Industrie. Berlin, Germany, November 2011. Available from: http://www.m2m-alliance.com/fileadmin/user_upload/pdf/IT_Gipfel_AG2_M2M_2011.pdf.

[42] Type-approval requirements for the deployment of the eCall in-vehicle system based on the 112 service. European Parliament ordinary legislative procedure 2013/0165(COD), April 2015. Available from: http://www.europarl.europa.eu/oeil/popups/ficheprocedure.do?lang=en&reference=2013/0165%28COD%29.

[43] acatech Industrie 4.0 Working Group. Recommendations for implementing the strategic initiative INDUSTRIE 4.0: Final report of the Industrie 4.0 Working Group, April 2013. Available from: http://www.acatech.de/fileadmin/user_upload/Baumstruktur_nach_Website/Acatech/root/de/Material_fuer_Sonderseiten/Industrie_4.0/Final_report__Industrie_4.0_accessible.pdf.

[44] Harald Weiss. Industrie 4.0 – ein deutscher Begriff. VDI Nachrichten, January 2014. Available from: http://www.imscenter.net/industry-4-0-activities/industrie-4-0-jay-lee.pdf.

[45] Bundesministerium für Bildung und Forschung. Zukunftsbild „Industrie 4.0“. Bonn, Germany, November 2013. Available from: http://www.bmbf.de/pubRD/Zukunftsbild_Industrie_40.pdf.

[46] Friedemann Mattern and Christian Floerkemeier. From the Internet of Computers to the Internet of Things. In Kai Sachs, Ilia Petrov, and Pablo Guerrero, editors, *From Active Data Management to Event-Based Systems and More*, volume 6462 of *LNCS*, pages 242–259. Springer, 2010.

[47] Kevin Ashton. That ‘Internet of Things’ Thing. RFID Journal, June 2009. Available from: http://www.rfidjournal.com/articles/view?4986.

[48] Christian Floerkemeier, Christof Roduner, and Matthias Lampe. RFID Application Development with the Accada Middleware

Platform. *IEEE Systems Journal, Special Issue on RFID Technology*, 1(2):82–94, December 2007.

[49] Dominique Guinard, Mathias Mueller, and Jacques Pasquier. Giving RFID a REST: Building a Web-Enabled EPCIS. In *Proceedings of Internet of Things 2010 International Conference (IoT 2010)*, Tokyo, Japan, November 2010.

[50] Eamonn O'Neill, Peter Thompson, Stavros Garzonis, and Andrew Warr. Reach Out and Touch: Using NFC and 2D Barcodes for Service Discovery and Interaction with Mobile Devices. In *Proceedings of the 5th International Conference on Pervasive Computing*, volume 4480 of *LNCS*, pages 19–36. Springer, May 2007.

[51] Robert Adelmann. *An efficient bar code recognition engine for enabling mobile services.* PhD thesis, ETH Zurich, Zurich, Switzerland, 2011.

[52] CMU SCS Coke Machine. Web page. Available from: http://www.cs.cmu.edu/~coke/.

[53] Ken Harrenstien. NAME/FINGER Protocol. Internet Requests for Comments: 742, December 1977. Available from: http://tools.ietf.org/html/rfc742.

[54] Adam Dunkels. Design and Implementation of the lwIP TCP/IP Stack. Master's thesis, Swedish Institute of Computer Science, February 2001.

[55] Adam Dunkels. Full TCP/IP for 8-bit Architectures. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys '03)*, pages 85–98. ACM, 2003.

[56] Adam Dunkels, Juan Alonso, Thiemo Voigt, Hartmut Ritter, and Jochen Schiller. Connecting Wireless Sensornets with TCP/IP Networks. In *Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC 2004)*, volume 2957 of *LNCS*, pages 143–152, Frankfurt, Germany, February 2004. Springer.

[57] Internet Protocol. Internet Requests for Comments: 791, September 1981. Available from: http://tools.ietf.org/html/rfc791.

[58] RIPE Network Coordination Centre. Reaching the Last /8. Web page, April 2015. Available from: https://www.ripe.net/publications/ipv6-info-centre/about-ipv6/ipv4-exhaustion/reaching-the-last-8.

[59] Stephen E. Deering and Robert M. Hinden. Internet Protocol, Version 6 (IPv6) Specification. Internet Requests for Comments: 2460, December 1998. Available from: http://tools.ietf.org/html/rfc2460.

[60] Nandakishore Kushalnagar, Gabriel Montenegro, and Christian Peter Pii Schumacher. IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals. Internet Requests for Comments: 4919, August 2007. Available from: http://tools.ietf.org/html/rfc4919.

[61] Jonathan W. Hui and David E. Culler. IP is Dead, Long Live IP for Wireless Sensor Networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys '08)*, pages 15–28, Raleigh, NC, USA, November 2008. ACM.

[62] Adam Dunkels and JP Vasseur. IP for Smart Objects. White paper, IPSO Alliance, July 2010. Available from: http://www.ipso-alliance.org/wp-content/media/why_ip.pdf.

[63] Martin Gudgin (Ed.), Marc Hadley (Ed.), Noah Mendelsohn (Ed.), Jean-Jacques Moreau (Ed.), Henrik Frystyk Nielsen (Ed.), Anish Karmarkar (Ed.), and Yves Lafon (Ed.). SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C Recommendation, April 2007. Available from: http://www.w3.org/TR/soap12-part1/.

[64] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Internet Requests for Comments: 2616, June 1999. Available from: http://tools.ietf.org/html/rfc2616.

[65] Roy T. Fielding and Richard N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology (TOIT)*, 2:115–150, May 2002.

[66] 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, 2011.

[67] Vlad Trifa. *Building Blocks for a Participatory Web of Things: Devices, Infrastructures, and Programming Frameworks*. PhD thesis, ETH Zurich, Zurich, Switzerland, August 2011.

[68] Dominique Guinard. *A Web of Things Application Architecture – Integrating the Real-World into the Web*. PhD thesis, ETH Zurich, Zurich, Switzerland, August 2011.

[69] Zach Shelby, Klaus Hartke, and Carsten Bormann. The Constrained Application Protocol (CoAP). Internet Requests for Comments: 7252, June 2014. Available from: http://tools.ietf.org/html/rfc7252.

[70] Ed. Akbar Rahman and Ed. Esko Dijk. Group Communication for the Constrained Application Protocol (CoAP). Internet Requests for Comments: 7390, October 2014. Available from: http://tools.ietf.org/html/rfc7390.

[71] Klaus Hartke. Observing Resources in CoAP. Internet-Draft, December 2014. Available from: http://tools.ietf.org/html/draft-ietf-core-observe-16.

[72] Matthias Kovatsch, Simon Duquennoy, and Adam Dunkels. A Low-Power CoAP for Contiki. In *Proceedings of the 8th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS 2011)*, pages 855–860, Valencia, Spain, October 2011. IEEE.

[73] Angelo P. Castellani, Mattia Gheda, Nicola Bui, Michele Rossi, and Michele Zorzi. Web Services for the Internet of Things through CoAP and EXI. In *Proceedings of the 2011 IEEE International Conference on Communications Workshops (ICC)*, pages 1–6, Kyoto, Japan, June 2011.

[74] Thomas Pötsch, Koojana Kuladinithi, Markus Becker, Peter Trenkamp, and Carmelita Goerg. Performance Evaluation of CoAP Using RPL and LPL in TinyOS. In *Proceedings of 5th International Conference on New Technologies, Mobility and Security (NTMS 2012)*, pages 1–5, Istanbul, Turkey, May 2012.

[75] International Business Machines Corporation (IBM) and Eurotech. MQ Telemetry Transport V3.1 (Protocol Specification), August 2010. Available from: http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/MQTT_V3.1_Protocol_Specific.pdf.

[76] AMQP Specification v1.0, October 2011. Available from: http://www.amqp.org/sites/amqp.org/files/amqp.pdf.

[77] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. Internet Requests for Comments: 6120, March 2011. Available from: http://tools.ietf.org/html/rfc6120.

[78] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. MQTT-S - A Publish/Subscribe Protocol For Wireless Sensor Networks. In *Proceedings of the 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE 2008)*, pages 791–798, Bangalore, India, January 2008. IEEE.

[79] Sven Bendel, Thomas Springer, Daniel Schuster, Alexander Schill, Ralf Ackermann, and Michael Ameling. A Service Infrastructure for the Internet of Things based on XMPP. In *Proceedings of the 11th IEEE International Conference on Pervasive Computing and Communications (PerCom 2013)*, pages 385–388, March 2013.

[80] IEEE Standards Glossary. Web page. Available from: http://www.ieee.org/education_careers/education/standards/standards_glossary.html.

[81] Gaetano Borriello and Roy Want. Embedded computation meets the World Wide Web. *Communications of the ACM*, 43(5):59–66, May 2000.

[82] John Barton and Tim Kindberg. The Challenges and Opportunities of Integrating the Physical World and Networked Systems. Technical Report HPL-2001-18, Hewlett Packard, January 2001.

[83] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, Bill Serra, and Mirjana Spasojevic. People, Places, Things: Web Presence for the Real World. *Mobile Networks and Applications*, 7(5):365–376, October 2002.

[84] Tim Kindberg and John Barton. A Web-based nomadic computing system. *Computer Networks*, 35(4):443–456, March 2001.

[85] Erik Wilde. Putting Things to REST. Technical Report 2007-015, UC Berkeley School of Information, November 2007.

[86] Dominique Guinard and Vlad Trifa. Towards the Web of Things: Web Mashups for Embedded Devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in Proceedings of WWW (International World Wide Web Conferences)*, Madrid, Spain, April 2009.

[87] Simon Mayer, Andreas Tschofen, Anind K. Dey, and Friedemann Mattern. User Interfaces for Smart Things - A Generative Approach with Semantic Interaction Descriptions. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 21(2, Article 12), February 2014.

[88] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web: a new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, 284(5):34–43, May 2001.

[89] Dennis Pfisterer, Kay Römer, Daniel Bimschas, Oliver Kleine, Richard Mietz, Cuong Truong, Henning Hasemann, Alexander Kröller, Max Pagel, Manfred Hauswirth, Marcel Karnstedt, Myriam Leggieri, Alexandre Passant, and Ray Richardson. SPITFIRE: Toward a Semantic Web of Things. *IEEE Communications Magazine*, 49(11):40–48, November 2011.

[90] Michael Compton, Payam Barnaghi, Luis Bermudez, Raúl García-Castro, Oscar Corcho, Simon Cox, John Graybeal, Man-

fred Hauswirth, Cory Henson, Arthur Herzog, Vincent Huang, Krzysztof Janowicz, W. David Kelsey, Danh Le Phuoc, Laurent Lefort, Myriam Leggieri, Holger Neuhaus, Andriy Nikolov, Kevin Page, Alexandre Passant, Amit Sheth, and Kerry Taylor. The SSN ontology of the W3C semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25–32, December 2012.

[91] Tim Berners-Lee. Personal Homepage. Available from: http://www.w3.org/People/Berners-Lee/.

[92] Dave Wiener. XML-RPC Specification. Web page, June 1999. Available from: http://xmlrpc.scripting.com/spec.html.

[93] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. Internet Requests for Comments: 3986, January 2005. Available from: http://tools.ietf.org/html/rfc3986.

[94] Dave Raggett (Ed.), Arnaud Le Hors (Ed.), and Ian Jacobs (Ed.). HTML 4.01 Specification. W3C Recommendation, December 1999. Available from: http://www.w3.org/TR/1999/REC-html401-19991224/.

[95] Ian Hickson (Ed.), Robin Berjon (Ed.), Steve Faulkner (Ed.), Travis Leithead (Ed.), Erika Doyle Navara (Ed.), Edward O'Connor (Ed.), and Silvia Pfeiffer (Ed.). HTML5 – A vocabulary and associated APIs for HTML and XHTML. W3C Recommendation, October 2014. Available from: http://www.w3.org/TR/2014/REC-html5-20141028/.

[96] Tim Bray (Ed.), Jean Paoli (Ed.), C. M. Sperberg-McQueen (Ed.), Eve Maler (Ed.), François Yergeau (Ed.), and John Cowa (Ed.). Extensible Markup Language (XML) 1.1 (Second Edition). W3C Recommendation, August 2006. Available from: http://www.w3.org/TR/xml11/.

[97] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). Internet Requests for Comments: 4627, July 2006. Available from: http://tools.ietf.org/html/rfc4627.

[98] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison Wesley, fourth edition, 2005.

[99] Benedikt Ostermaier, B. Maryam Elahi, Kay Römer, Michael Fahrmair, and Wolfgang Kellerer. Poster Abstract: Dyser – Towards a Real-Time Search Engine for the Web of Things. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys '08)*, pages 429–430, Raleigh, NC, USA, November 2008. ACM.

[100] B. Maryam Elahi, Kay Römer, Benedikt Ostermaier, Michael Fahrmair, and Wolfgang Kellerer. Sensor Ranking: A Primitive for Efficient Content-based Sensor Search. In *Proceedings of the 8th International Conference on Information Processing in Sensor Networks (IPSN 2009)*, pages 217–228, San Francisco, CA, USA, April 2009. IEEE Computer Society.

[101] 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.

[102] Kay Römer, Benedikt Ostermaier, Friedemann Mattern, Michael Fahrmair, and Wolfgang Kellerer. Real-Time Search for Real-World Entities: A Survey. *Proceedings of the IEEE*, 98(11):1887–1902, November 2010.

[103] Michael Fahrmair, Wolfgang Kellerer, Kay Römer, Benedikt Ostermaier, and Friedemann Mattern. Method and apparatus for searching a plurality of realtime sensors. Patent application EP2131292, filed June 2008, published September 2009. Available from: https://register.epo.org/espacenet/application?lng=de&number=EP08157818&tab=main.

[104] Google. Inside Search: Algorithms. Web page, October 2013. Available from: http://www.google.com/insidesearch/howsearchworks/algorithms.html.

[105] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, April 1998.

[106] Simon Tabor. Google's 2013 downtime caused a 40% drop in global traffic. GoSquared Engineering Blog, August 2013. Available from: https://engineering.gosquared.com/googles-downtime-40-drop-in-traffic.

[107] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. Web page, 1998. Full version with appendix. Available from: http://infolab.stanford.edu/~backrub/google.html.

[108] Knut Magne Risvik and Rolf Michelsen. Search engines and Web dynamics. *Computer Networks*, 39(3):289–302, June 2002.

[109] Abhishek Das and Ankit Jain. *Next Generation Search Engines: Advanced Models for Information Retrieval*, chapter Indexing the World Wide Web: The Journey So Far, pages 1–28. IGI Global, 2012.

[110] ISO/IEC. Information technology – Database languages – SQL. ISO/IEC 9075(1-4,9-11,13,14):2011, December 2011. Available from: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=53681.

[111] Nathan Eagle and Alex (Sandy) Pentland. Reality Mining: Sensing Complex Social Systems. *Personal and Ubiquitous Computing*, 10(4):255–268, May 2006.

[112] Dave Evans. The Internet of Things - How the Next Evolution of the Internet Is Changing Everything. White Paper, Cisco, April 2011. Available from: http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf.

[113] Augustine Chidi Ikeji and Farshad Fotouhi. An adaptive real-time Web search engine. In *Proceedings of the 2nd International Workshop on Web information and Data Management (WIDM '99)*, pages 12–16, Kansas City, MO, USA, November 1999. ACM.

[114] Burr S. Watters. Development and Performance Evaluation of a Real Time Web Search Engine. Master's thesis, University of North Florida, December 2004.

[115] Nathan Eagle and Alex Sandy Pentland. Eigenbehaviors: identifying structure in routine. *Behavioral Ecology and Sociobiology*, 63(7):1057–1066, May 2009.

[116] Chaoming Song, Zehui Qu, Nicholas Blumm, and Albert-László Barabási. Limits of Predictability in Human Mobility. *Science*, 327(5968):1018–1021, February 2010.

[117] Adi Raveh and Charles S. Tapiero. Periodicity, Constancy, Heterogeneity and the Categories of Qualitative Time Series. *Ecology*, 61(3):715–719, June 1980.

[118] Mohamed Elfeky, Walid Aref, and Ahmed Elmagarmid. Using Convolution to Mine Obscure Periodic Patterns in One Pass. In Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, editors, *Advances in Database Technology - EDBT 2004*, volume 2992 of *LNCS*, pages 543–544. Springer, March 2004.

[119] Homepage of Bicing. Web page. Available from: http://www.bicing.cat/.

[120] Homepage of microformats. Web Page. Available from: http://microformats.org.

[121] WHATWG. Microdata. WHATWG Draft Standard, October 2013. Available from: http://www.whatwg.org/specs/web-apps/current-work/multipage/microdata.html.

[122] W3C. RDFa Core 1.1 - Third Edition. W3C Recommendation, March 2015. Available from: http://www.w3.org/TR/2015/REC-rdfa-core-20150317/.

[123] Eric Prud'hommeaux (Ed.) and Andy Seaborne (Ed.). SPARQL Query Language for RDF. W3C Recommendation, January 2008. Available from: http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[124] Karl Aberer, Manfred Hauswirth, and Ali Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *Proceedings of the 2007 International Conference on Mobile Data Management*, pages 198–205, Mannheim, Germany, May 2007. IEEE.

[125] Kok-Kiong Yap, Vikram Srinivasan, and Mehul Motani. MAX: Human-Centric Search of the Physical World. In *Proceedings*

of the 3rd Conference on Embedded Networked Sensor Systems (SenSys '05), pages 166–179, San Diego, CA, USA, November 2005. ACM.

[126] Kok-Kiong Yap, Vikram Srinivasan, and Mehul Motani. MAX: Wide Area Human-Centric Search of the Physical World. *ACM Transactions on Sensor Networks (TOSN)*, 4(4):26:1–26:34, August 2008.

[127] Chiu C. Tan, Bo Sheng, Haodong Wang, and Qun Li. Microsearch: A Search Engine for Embedded Devices Used in Pervasive Computing. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(4):43:1–43:29, March 2010.

[128] Haodong Wang, Chiu 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.

[129] Christian Frank, Philipp Bolliger, Christof Roduner, and Wolfgang Kellerer. Objects Calling Home: Locating Objects Using Mobile Phones. In *Proceedings of the 5th International Conference on Pervasive Computing (Pervasive 2007)*, volume 4480 of *LNCS*. Springer, Toronto, Canada, May 2007.

[130] Christian Frank, Christof Roduner, Chie Noda, and Wolfgang Kellerer. Query Scoping for the Sensor Internet. In *Proceedings of the IEEE International Conference on Pervasive Services (ICPS 2006)*, Lyon, France, June 2006.

[131] Jianwei Liu, Haiying Shen, Ze Li, Shoshana Loeb, and Stanley Moyer. SCPS: A Social-Aware Distributed Cyber-Physical Human-Centric Search Engine. In *Proceedings of the 2011 IEEE Global Telecommunications Conference (GLOBECOM 2011)*, pages 1–5, Houston, TX, USA, December 2011.

[132] Tingxin Yan, Deepak Ganesan, and R. Manmatha. Distributed Image Search in Camera Sensor Networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys '08)*, pages 155–168, Raleigh, NC, USA, November 2008. ACM.

[133] Misbah Uddin, Rolf Stadler, and Alexander Clemm. Management by Network Search. In *Proceedings of the 2012 IEEE Network*

Operations and Management Symposium (NOMS 2012), pages 146–154, Maui, HI, USA, April 2012. IEEE.

[134] Misbah Uddin, Rolf Stadler, and Alexander Clemm. Scalable Matching and Ranking for Network Search. In *Proceedings of the 9th International Conference on Network and Service Management (CNSM '13)*, pages 251–259, Zurich, Switzerland, October 2013. IEEE.

[135] Richard Mietz and Kay Römer. Exploiting Correlations for Efficient Content-based Sensor Search. In *Proceedings of the IEEE Sensors 2011 Conference*, pages 187–190, Limerick, Ireland, October 2011. IEEE.

[136] Cuong Truong and Kay Römer. Content-Based Sensor Search for the Web of Things. In *Proceedings of the 2013 IEEE Global Telecommunications Conference (GLOBECOM 2013)*, pages 2654–2660, Atlanta, GA, USA, December 2013. IEEE.

[137] Cuong Truong, Kay Römer, and Kai Chen. Fuzzy-based sensor search in the Web of Things. In *Proceedings of the 3rd International Conference on the Internet of Things (IoT 2012)*, pages 127–134, Wuxi, China, October 2012.

[138] Richard Mietz, Sven Groppe, Kay Römer, and Dennis Pfisterer. Semantic Models for Scalable Search in the Internet of Things. *Journal of Sensor and Actuator Networks*, 2(2):172–195, March 2013.

[139] Simon Mayer, Dominique Guinard, and Vlad Trifa. Searching in a Web-based Infrastructure for Smart Things. In *Proceedings of the 3rd International Conference on the Internet of Things (IoT 2012)*, pages 119–126, Wuxi, China, October 2012.

[140] Jonas Michel, Christine Julien, Jamie Payton, and Gruia-Catalin Roman. Gander: Personalizing Search of the Here and Now. In Alessandro Puiatti and Tao Gu, editors, *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, volume 104 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 88–100. Springer, 2012. Revised Selected Papers of the 8th International ICST Conference, MobiQuitous 2011.

[141] Zhiming Ding, Zhikui Chen, and Qi Yang. IoT-SVKSearch: a real-time multimodal search engine mechanism for the internet of things. *International Journal of Communication Systems*, 27(6):871–897, June 2014.

[142] Benoit Christophe, Vincent Verdot, and Vincent Toubiana. Searching the 'Web of Things'. In *Proceedings of the Fifth IEEE International Conference on Semantic Computing (ICSC 2011)*, pages 308–315, Palo Alto, CA, USA, September 2011.

[143] Biz Stone. Twitter and XMPP: Drinking from The Fire Hose. Twitter Blog, July 2008. Available from: http://blog.twitter.com/2008/07/twitter-and-xmpp-drinking-from-fire.html.

[144] Danny Sullivan. Bing, Now With Extra Facebook: See What Your Friends Like & People Search Results. Search Engine Land, October 2010. Available from: http://searchengineland.com/bing-now-with-extra-facebook-see-what-your-friends-like-52848.

[145] Twitter. Posting a Tweet. Web page, September 2015. Available from: https://support.twitter.com/articles/15367.

[146] Biz Stone. Finding A Perfect Match. Twitter Blog, July 2008. Available from: http://blog.twitter.com/2008/07/finding-perfect-match.html.

[147] Twitter Engineering. The Engineering Behind Twitter's New Search Experience. Twitter Blog, May 2011. Available from: https://blog.twitter.com/2011/engineering-behind-twitter%E2%80%99s-new-search-experience.

[148] The top 500 sites on the web. Homepage of Alexa.com, February 2012. Available from: http://www.alexa.com/topsites.

[149] Stefanie Olsen. Google search gets newsier. CNET News, September 2002. Available from: http://news.cnet.com/2100-1023-958927.html.

[150] Amit Singhal. Relevance meets the real-time web. The official Google Blog, July 2009. Available from: http://googleblog.blogspot.com/2009/12/relevance-meets-real-time-web.html.

[151] Dylan Casey. Replay it: Google search across the Twitter archive. The official Google blog, April 2010. Available from: http://googleblog.blogspot.com/2010/04/replay-it-google-search-across-twitter.html.

[152] Danny Sullivan. As Deal With Twitter Expires, Google Realtime Search Goes Offline. Search Engine Land, July 2011. Available from: http://searchengineland.com/as-deal-with-twitter-expires-google-realtime-search-goes-offline-84175.

[153] Ben Parr. Google To Revive Realtime Search, Thanks to Google+. Mashable.com, August 2011. Available from: http://mashable.com/2011/08/04/google-realtime-search-revive/.

[154] Carrie Grimes. Our new search index: Caffeine. The official Google blog, June 2010. Available from: http://googleblog.blogspot.ch/2010/06/our-new-search-index-caffeine.html.

[155] Vic Gundotra. Introducing the Google+ project: Real-life sharing, rethought for the web. The official Google Blog, June 2011. Available from: http://googleblog.blogspot.com/2011/06/introducing-google-project-real-life.html.

[156] Betsy Aoki. Bing Feature Update: Bing News with Real-Time Twitter feed and Enhanced Entertainment Sharing. Bing Blogs, March 2011. Available from: http://blogs.bing.com/search/2011/03/25/bing-feature-update-bing-news-with-real-time-twitter-feed-and-enhanced-entertainment-sharing/.

[157] Technorati. About Technorati. Web page, July 2010. Archived by web.archive.org. Available from: http://web.archive.org/web/20100722150939/http://technorati.com/about-technorati/.

[158] Technorati. Welcome to the former ping page. Web page, March 2011. Archived by web.archive.org. Available from: https://web.archive.org/web/20110322224938/http://technorati.com/ping/.

[159] Chris Sherman. Me.dium Launches "Real Time" Social Search. Search Engine Land, July 2008. Available from: http://searchengineland.com/medium-launches-real-time-social-search-14348.

[160] OneRiot. FAQ. Web page, June 2009. Archived by web.archive.org. Available from: http://web.archive.org/web/20090601170157/http://www.oneriot.com/company/help.

[161] Tobias Peggs. The Inner Workings of a Realtime Search Engine: Thoughts on realtime search, by the team at OneRiot. White Paper from OneRiot, Online (docstoc), November 2009. Available from: http://www.docstoc.com/docs/16947406/OneRiot-Inner-Workings-of-a-Realtime-Search-Engine.

[162] Anand Rajaraman. @WalmartLabs += OneRiot; // Welcome aboard, Team OneRiot! The official @WallmartLabs Blog, September 2011. Available from: http://walmartlabs.blogspot.com/2011/09/walmartlabs-oneriot-welcome-aboard-team.html.

[163] Jolie O'Dell. Startup Collecta Shuts Down Its Product, Starts Working on a New One. Mashable.com, January 2011. Available from: http://mashable.com/2011/01/19/startup-collecta-shuts-down-search-engine/.

[164] Junghoo Cho and Hector Garcia-Molina. The Evolution of the Web and Implications for an Incremental Crawler. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*, pages 200–209. Morgan Kaufmann Publishers, September 2000.

[165] Jenny Edwards, Kevin McCurley, and John Tomlin. An Adaptive Model for Optimizing Performance of an Incremental Web Crawler. In *Proceedings of the 10th International World Wide Web Conference (WWW '01)*, pages 106–113, Hong Kong, PRC, May 2001.

[166] sitemaps.org. Sitemaps XML format. Web page, February 2008. Available from: http://www.sitemaps.org/protocol.html.

[167] M. Nottingham (Ed.) and R. Sayre (Ed.). The Atom Syndication Format. Internet Requests for Comments: 4287, December 2005. Available from: http://tools.ietf.org/html/rfc4287.

[168] RSS 2.0 Specification. Web page, March 2009. Available from: http://www.rssboard.org/rss-2-0-11.

[169] Karl Aberer, Manfred Hauswirth, and Ali Salehi. The Global Sensor Networks middleware for efficient and flexible deployment and interconnection of sensor networks. Technical Report LSIR-REPORT-2006-006, Ecole Polytechnique Fédérale de Lausanne (EPFL), 2006. Available from: http://lsirpeople.epfl.ch/salehi/papers/LSIR-REPORT-2006-006.pdf.

[170] Karl Aberer, Manfred Hauswirth, and Ali Salehi. Middleware Support for the "Internet of Things". In *Fachgespräch Sensornetze*, Stuttgart, Germany, 2006.

[171] Aman Kansal, Suman Nath, Jie Liu, and Feng Zhao. SenseWeb: An Infrastructure for Shared Sensing. *IEEE MultiMedia*, 14(4):8–13, October-December 2007.

[172] Microsoft Research. SenseWeb. Web page. Available from: http://research.microsoft.com/en-us/projects/senseweb/.

[173] Microsoft Research. SenseWeb Tutorial. Online, January 2009. Available from: http://research.microsoft.com/en-us/projects/senseweb/SenseWebTutorial.pdf.

[174] Pachube. About us. Web page, June 2011. Archived by web.archive.org. Available from: https://web.archive.org/web/20110623144213/http://pachube.com/about_us.

[175] Pachube. Find Feeds. Web page, June 2011. Archived by web.archive.org. Available from: https://web.archive.org/web/20110625074320/http://pachube.com/feeds.

[176] Hompage of Traderbot.com. Web page, June 2007. Archived by web.archive.org. Available from: http://web.archive.org/web/20070601214544/http://www.traderbot.com/.

[177] Benedikt Ostermaier, Fabian Schlup, and Kay Römer. WebPlug: A Framework for the Web of Things. In *Proceedings of the*

1st International Workshop on the Web of Things (WoT 2010), Mannheim, Germany, March 2010.

[178] Benedikt Ostermaier, Fabian Schlup, and Matthias Kovatsch. Leveraging the Web of Things for Rapid Prototyping of UbiComp Applications. In *Adjunct Proceedings of the 12th ACM International Conference on Ubiquitous Computing (UbiComp 2010)*, pages 375–376, Copenhagen, Denmark, September 2010. ACM.

[179] Fabian Schlup. Design and Implementation of a Framework for the Web of Things. Master's thesis, ETH Zurich, Zurich, Switzerland, September 2009.

[180] Brian Sletten. Resource-Oriented Architecture: The Rest of REST. Web page, December 2009. Available from: http://www.infoq.com/articles/roa-rest-of-rest.

[181] Lisa Dusseault (Ed.). HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). Internet Requests for Comments: 4918, June 2007. Available from: http://tools.ietf.org/html/rfc4918.

[182] Sandro Hawke. Re: siteData-36: strawman + httpRange-14 ["Resource-Type:"]. Archived message of W3C mailing list www-tag@w3.org, February 2003. Available from: http://lists.w3.org/Archives/Public/www-tag/2003Feb/0299.

[183] Bernard Desruisseaux (Ed.). Internet Calendaring and Scheduling Core Object Specification (iCalendar). Internet Requests for Comments: 5545, September 2009. Available from: http://tools.ietf.org/html/rfc5545.

[184] Jeff Lindsay. Webhooks. Web page. Available from: http://www.webhooks.org/.

[185] Brad Fitzpatrick, Brett Slatkin, Martin Atkins, and Julien Genestoux. PubSubHubbub Core 0.4 – Working Draft, June 2013. Available from: http://pubsubhubbub.github.io/PubSubHubbub/pubsubhubbub-core-0.4.html.

[186] Joe Gregorio, Roy T. Fielding, Marc Hadley, Mark Nottingham, and David Orchard. URI Template. Internet Requests for Comments: 6570, March 2012. Available from: http://tools.ietf.org/html/rfc6570.

[187] Daiki Ueno, Tatsuo Nakajima, Ichiro Satoh, and Kouta Soejima. Web-Based Middleware for Home Entertainment. In Alain Jean-Marie, editor, *Advances in Computing Science — ASIAN 2002*, volume 2550 of *LNCS*, pages 206–219. Springer, November 2002.

[188] Witold Drytkiewicz, Ilja Radusch, Stefan Arbanowski, and Radu Popescu-Zeletin. pREST: a REST-based Protocol for Pervasive Systems. In *Proceedings of the 1st IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS 2004)*, pages 340–348, Fort Lauderdale, FL, USA, October 2004. IEEE.

[189] Justin R. Erenkrantz, Michael M. Gorlick, and Richard N. Taylor. CREST: A new model for Decentralized, Internet-Scale Applications. Technical Report UCI-ISR-09-4, University of California, Irvine, September 2009. Available from: http://isr.uci.edu/tech_reports/UCI-ISR-09-4.pdf.

[190] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz, and David Culler. sMAP - a Simple Measurement and Actuation Profile for Physical Information. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys '10)*, pages 197–210, Zurich, Switzerland, November 2010. ACM.

[191] Michael Blackstock and Rodger Lea. IoT mashups with the WoTKit. In *3rd International Conference on the Internet of Things (IoT 2012)*, pages 159–166. IEEE, October 2012.

[192] Dominique Guinard, Vlad Trifa, and Erik Wilde. A Resource Oriented Architecture for the Web of Things. In *Proceedings of Internet of Things 2010 International Conference (IoT 2010)*, Tokyo, Japan, November 2010.

[193] UPnP Forum. UPnP Device Architecture 1.1, October 2008. Available from: http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf.

[194] Joe Gregorio (Ed.) and Bill de hOra (Ed.). The Atom Publishing Protocol. Internet Requests for Comments: 5023, October 2007. Available from: http://tools.ietf.org/html/rfc5023.

[195] Netscape. An Exploration of Dynamic Documents in Netscape 1.1. Web page, April 1999. Archived by web.archive.org. Avail-

able from: https://web.archive.org/web/19990423194400/http://www1.netscape.com/home/demo/1.1b1/pushpull.html.

[196] Josh Cohen. GENA Framework. Presentation slides at WISEN Workshop on Internet Scale Event Notification, Irvine, CA, USA, July 1998. Available from: http://isr.uci.edu/events/twist/wisen98/presentations/Cohen/Cohen.PPT.

[197] Comet (programming). Wikipedia.org, April 2015. Available from: http://en.wikipedia.org/w/index.php?title=Comet_(programming)&oldid=659876056.

[198] Ian Hickson (Ed.). Server-Sent Events. W3C Recommendation, February 2015. Available from: http://www.w3.org/TR/2015/REC-eventsource-20150203/.

[199] Ian Fette and Alexey Melnikov. The WebSocket Protocol. Internet Requests for Comments: 6455, December 2011. Available from: http://tools.ietf.org/html/rfc6455.

[200] Technorati. Developers: Ping Configurations. Web page, February 2005. Archived by web.archive.org. Available from: https://web.archive.org/web/20050221002717/http://www.technorati.com/developers/pingconfig.html.

[201] Vipul Gupta, Poornaprajna Udupi, and Arshan Poursohi. Early Lessons from Building Sensor.Network: an Open Data Exchange for the Web of Things. In *Proceedings of the 1st International Workshop on the Web of Things (WoT 2010)*, Mannheim, Germany, April 2010. IEEE.

[202] Vlad Trifa, Dominique Guinard, Vlatko Davidovski, Andreas Kamilaris, and Ivan Delchev. Web Messaging for Open and Scalable Distributed Sensing Applications. In *Proceedings of ICWE 2010 (International Conference on Web Engineering)*, Vienna, Austria, July 2010. Springer.

[203] Andreas Kamilaris, Andreas Pitsillides, and Vlad Trifa. The Smart Home meets the Web of Things. *International Journal of Ad Hoc and Ubiquitous Computing*, 7(3):145–154, May 2011.

[204] Kevin Chang, Nathan Yau, Mark Hansen, and Deborah Estrin. SensorBase.org - A Centralized Repository to Slog Sensor Network Data. In *Proceedings of DCOSS '06 Workshop Euro-American Workshop on Middleware for Sensor Networks (EAWMS)*, San Francisco, CA, USA, June 2006.

[205] Gong Chen, Nathan Yau, Mark Hansen, and Deborah Estrin. Sharing Sensor Network Data. Technical Report 71, CENS, March 2007.

[206] Robert F. Dickerson, Konrad Jiakang Lu, Jian Lu, and Kamin Whitehouse. Stream Feeds - An Abstraction for the World Wide Sensor Web. In *Proceedings of the First Internet of Things International Conference (IoT 2008)*, volume 4952 of *LNCS*, pages 360–375, Zurich, Switzerland, 2008. Springer.

[207] Mark Nottingham. FIQL: The Feed Item Query Language. Internet-Draft, December 2007. Available from: http://tools.ietf.org/html/draft-nottingham-atompub-fiql-00.

[208] Geoffrey Clemm, Jim Amsden, Tim Ellison, Christopher Kaler, and Jim Whitehead. Versioning Extensions to WebDAV. Internet Requests for Comments: 3253, March 2002. Available from: http://tools.ietf.org/html/rfc3253.

[209] Florian Müller (Ed.), Ryan McVeigh (Ed.), and Jens Hübel (Ed.). Content Management Interoperability Services (CMIS) Version 1.1. OASIS Committee Specification, November 2012. Available from: http://docs.oasis-open.org/cmis/CMIS/v1.1/cs01/CMIS-v1.1-cs01.pdf.

[210] Al Brown, Geoffrey Clemm, and Julian F. Reschke (Ed.). Link Relation Types for Simple Version Navigation between Web Resources. Internet Requests for Comments: 5829, April 2010. Available from: http://tools.ietf.org/html/rfc5829.

[211] Benedikt Ostermaier, Matthias Kovatsch, and Silvia Santini. Connecting Things to the Web using Programmable Low-power WiFi Modules. In *Proceedings of the 2nd International Workshop on the Web of Things (WoT 2011)*, San Franciso, CA, USA, June 2011.

[212] Michael Beigl, Tobias Zimmer, Albert Krohn, Christian Decker, and Philip Robinson. Smart-Its - Communication and Sensing Technology for UbiComp Environments. Technical Report ISSN 1432-7864 2003/2, Universität Karlsruhe, Fakultät für Informatik, February 2003.

[213] Open Security Architecture. Icon Library. Web page, September 2015. Available from: http://www.opensecurityarchitecture.org/cms/library/icon-library.

[214] The Active Badge System. Web page. Available from: http://www.cl.cam.ac.uk/research/dtg/attarchive/ab.html.

[215] Fitbit Flex Product Specifications. Web page, May 2015. Available from: https://www.fitbit.com/flex#specs.

[216] Markus Kreitmair. Raspberry Pi talks EnOcean. White Paper, October 2013. Available from: https://www.enocean.com/fileadmin/redaktion/pdf/white_paper/wp_Raspberry_talks_EnOcean.pdf.

[217] Angelo P. Castellani, Salvatore Loreto, Akbar Rahman, Thomas Fossati, and Esko Dijk. Guidelines for HTTP-CoAP Mapping Implementations. Internet-Draft, March 2015. Available from: https://tools.ietf.org/html/draft-ietf-core-http-mapping-06.

[218] John Schneider and Takuki Kamiya. Efficient XML Interchange (EXI) Format 1.0. W3C Recommendation, February 2014. Available from: http://www.w3.org/TR/2014/REC-exi-20140211/.

[219] Carsten Bormann and Paul Hoffman. Concise Binary Object Representation (CBOR). Internet Requests for Comments: 7049, October 2013. Available from: http://tools.ietf.org/html/rfc7049.

[220] Transmission Control Protocol. Internet Requests for Comments: 793, September 1981. Available from: http://tools.ietf.org/html/rfc793.

[221] Tim Dierks (Ed.) and Eric Rescorla (Ed.). The Transport Layer Security (TLS) Protocol Version 1.2. Internet Requests for Com-

ments: 5246, August 2008. Available from: http://tools.ietf.org/html/rfc5246.

[222] P.V. Mockapetris. Domain names - concepts and facilities. Internet Requests for Comments: 1034 (Standard), November 1987. Available from: http://tools.ietf.org/html/rfc1034.

[223] P.V. Mockapetris. Domain names - implementation and specification. Internet Requests for Comments: 1035 (Standard), November 1987. Available from: http://tools.ietf.org/html/rfc1035.

[224] J. Postel. User Datagram Protocol. Internet Requests for Comments: 768 (Standard), August 1980. Available from: http://tools.ietf.org/html/rfc768.

[225] Ramon Caceres and Liviu Iftode. Improving the Performance of Reliable Transport Protocols in Mobile Computing Environments. *IEEE Journal on Selected Areas in Communications*, 13(5):850–857, June 1995.

[226] Roving Networks. RN-131. Data sheet, February 2010.

[227] GainSpan. GS1011MxxS. Data sheet, April 2013. Available from: https://s3.amazonaws.com/site_support/uploads/document_upload/GS1011MxxS_Module_Datasheet_rev1_01.pdf.

[228] Skyhook Wireless Location SDK Overview. Web page, March 2013. Archived by web.archive.org. Available from: http://web.archive.org/web/20130316012528/http://www.skyhookwireless.com/location-technology/index.php.

[229] Google Maps Geolocation API. Web page. Available from: https://developers.google.com/maps/documentation/geolocation/intro.

[230] Philipp Bolliger. Redpin – Adaptive, Zero-Configuration Indoor Localization through User Collaboration. In *Proceedings of the First ACM International Workshop on Mobile Entity Localization and Tracking in GPS-less Environment Computing and Communication Systems*, San Francisco, USA, September 2008.

[231] G2 Microsystems. Epsilon Module Family. Product Brief, July 2009.

[232] G2 Microsystems. G2C547 Wi-Fi SoC. Product Brief, April 2009.

[233] Hans van Leeuwen. Wi-Fi Enabled Sensors. In *10th Leibniz Conference of Advanced Science (Sensorsysteme 2010)*, Lichtenwalde, Germany, October 2010. Presentation slides. Available from: http://www.leibniz-institut.de/ss2010/van_leuwen_sensors_lower_power_wifi.pdf.

[234] Silviu Folea and Marius Ghercioiu. Ultra-Low Power Wi-Fi Tag for Wireless Sensing. In *Proceedings of the 2008 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR 2008)*, pages 247–252, Cluj-Napoca, Romania, May 2008. IEEE.

[235] Serbulent Tozlu. Feasibility of Wi-Fi Enabled Sensors for Internet of Things. In *Proceedings of the 7th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 291–296, Istanbul, Turkey, July 2011.

[236] GS1 and EPCglobal. EPC Radio-Frequency Identity Protocols Class-1 Generation-2 UHF RFID Protocol for Communications at 860 MHz — 960 MHz Version 1.2.0. Specification for RFID Air Interface, October 2008. Available from: http://www.gs1.org/gsmp/kc/epcglobal/uhfc1g2/uhfc1g2_1_2_0-standard-20080511.pdf.

[237] ISO/IEC. Information technology – Real-time locating systems (RTLS) – Part 2: 2,4 GHz air interface protocol. ISO/IEC 24730-2:2006, December 2006. Available from: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=40508.

[238] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: Enabling Ultra-Low Power Wireless Research. In *Proceedings of the 4th International Conference on Information Processing in Sensor Networks: Special Track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS 2005)*, pages 364–369, Los Angeles, CA, USA, April 2005.

[239] Roving Networks. RN-134. Data sheet, December 2009.

[240] Michael Beigl and Hans Gellersen. Smart-Its: An Embedded Platform for Smart Objects. In *Proceedings of Smart Objects Conference (sOc)*, pages 15–17, Grenoble, France, May 2003.

[241] Ralph Droms. Dynamic Host Configuration Protocol. Internet Requests for Comments: 2131, March 1997. Available from: http://tools.ietf.org/html/rfc2131.

[242] Wi-Fi CERTIFIED Wi-Fi Protected Setup™: Easing the User Experience for Home and Small Office Wi-Fi® Networks. White Paper, March 2014. Available from: https://www.wi-fi.org/file/wi-fi-certified-wi-fi-protected-setup-easing-the-user-experience-for-home-and-small-office-wi.

[243] Roving Networks. WPS App note. Application Note, September 2011.

[244] BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks. Web page. Available from: http://btnode.ethz.ch/.

[245] Matthias Ringwald, Kay Römer, and Andrea Vitaletti. Passive Inspection of Sensor Networks. In *Proceedings of the 3rd IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS 2007)*, pages 205–222, Santa Fe, NM, USA, June 2007.

[246] Jan Beutel, Oliver Kasten, Friedemann Mattern, Kay Römer, Frank Siegemund, and Lothar Thiele. Prototyping Wireless Sensor Network Applications with BTnodes. In *1st European Workshop on Wireless Sensor Networks (EWSN)*, volume 2920 of *LNCS*, pages 323–338, Berlin, Germany, January 2004. Springer.

[247] Jari T. Savolainen, Harri Hirvola, and Sassan Iraji. EPC UHF RFID Reader: Mobile Phone Integration and Services. In *Proceedings of the 6th IEEE Consumer Communications and Networking Conference (CCNC 2009)*, pages 1–5, January 2009.

[248] David L. Mills, Jim Martin (Ed.), Jack Burbank, and William Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. Internet Requests for Comments: 5905, June 2010. Available from: http://tools.ietf.org/html/rfc5905.

[249] John Franks, Phillip M. Hallam-Baker, Jeffery L. Hostetler, Scott D. Lawrence, Paul J. Leach, Ari Luotonen, and Lawrence C. Stewart. HTTP Authentication: Basic and Digest Access Authentication. Internet Requests for Comments: 2617, June 1999. Available from: http://tools.ietf.org/html/rfc2617.

[250] EPCglobal. EPCglobal Tag Data Standards Version 1.3. Ratified Specification, March 2008. Available from: http://www.gs1.at/images/stories/Leistungen_und_Standards/EPCglobal/Standards/TDS/GS1_EPC_tds_1_3-standard-20060308.pdf.

[251] EPC Global. Low Level Reader Protocol (LLRP), Version 1.0.1. Ratified Standard with Approved Fixed Errata, August 2007. Available from: http://www.gs1.org/gsmp/kc/epcglobal/llrp/llrp_1_0_1-standard-20070813.pdf.

[252] Impinj. Speedway Reader Brochure. Data sheet, November 2008. Available from: http://www.cisper.nl/rfid/downloads/Impinj_Speedway_Reader_Brochure_11_08.pdf.

[253] Impinj. CS-777 Brickyard™ Near-Field Antenna. Data sheet, May 2007. Available from: https://support.impinj.com/hc/en-us/article_attachments/200774758/IPJ_Brickyard_CSL_Datasheet_20081212.pdf.

[254] Meshed Systems GmbH. UHF Mini Antenne für Etikettendrucker. Web page, September 2015. Available from: http://www.meshedsystems.com/printable/rfid-komponenten/uhf-rfid-antennen/uhf-mini-rfid-antenne-fuer-etikettendrucker/index.htm.

[255] Alien Technology. ALN-9662 Short Inlay. Data sheet, June 2013. Available from: http://www.alientechnology.com/wp-content/uploads/Alien-Technology-Higgs-3-ALN-9662-Short.pdf.

[256] Impinj. Speedway® xPortal™. Data sheet, June 2011. Available from: https://support.impinj.com/hc/en-us/article_attachments/200774348/IPJ_Speedway_xPortal_Brochure_20110514.pdf.

[257] Brian Frank. Chopan - Compressed HTTP Over PANs. Internet-Draft, September 2009. Available from: http://tools.ietf.org/html/draft-frank-6lowapp-chopan-00.

[258] Gilman Tolle. Embedded Binary HTTP (EBHTTP). Internet-Draft, March 2010. Available from: https://tools.ietf.org/html/draft-tolle-core-ebhttp-00.

[259] Bluetooth Special Interest Group. GATT REST API. White Paper, April 2014. Available from: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=285910.

[260] Jim Rees and Peter Honeymanx. Webcard: a Java Card web server. Technical Report 99-3, Center for Information Technology Integration, University of Michigan, September 1999. Available from: https://www.citi.umich.edu/techreports/reports/citi-tr-99-3.pdf.

[261] Dogan Yazar and Adam Dunkels. Efficient Aplication Integration in IP-Based Sensor Networks. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys '09)*, pages 43–48, Berkeley, CA, USA, November 2009. ACM.

[262] Serbulent Tozlu. Experimental study of security impact on battery lifetime for low-power Wi-Fi systems. Presentation slides at Wireless Congress 2010, Munich, Germany, November 2010.

[263] Serbulent Tozlu and Murat Senel. Battery Lifetime Performance of Wi-Fi Enabled Sensors. In *Proceedings of the 2012 IEEE Consumer Communications and Networking Conference (CCNC 2012)*, pages 429–433, Las Vegas, NV, USA, January 2012.

[264] Serbulent Tozlu, Murat Senel, Wei Mao, and Abtin Keshavarzian. Wi-Fi Enabled Sensors for Internet of Things: A Practical Approach. *IEEE Communications Magazine*, 50(6):134 –143, June 2012.

[265] Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs). *IEEE Std 802.15.4-2006*, September 2006.

[266] Texas Instruments. MSP Low-Power Microcontrollers. Product Brochure, May 2015. Available from: http://www.ti.com/lit/sg/slab034w/slab034w.pdf.

[267] Texas Instruments. CC2420 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver. Data sheet, March 2013. Available from: http://www.ti.com/lit/ds/symlink/cc2420.pdf.

[268] Alexander Klapproth and Thomas Bürkli. TCP/IP über IEEE 802.15.4. ZigBee Entwicklerforum, Design & Elektronik, Munich, Germany, April 2006. Available from: http://www.ihomelab.ch/fileadmin/Dateien/PDF/FHLuzern_TCPIPoverIEEE8022015204.pdf.

[269] Gabriel Montenegro, Nandakishore Kushalnagar, Jonathan W. Hui, and David E. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. Internet Requests for Comments: 4944, September 2007. Available from: http://tools.ietf.org/html/rfc4944.

[270] Christin Groba and Siobhan Clarke. Web services on embedded systems - A performance study. In *Proceedings of the 1st International Workshop on the Web of Things (WoT 2010)*, Mannheim, Germany, March 2010.

[271] Adam Dunkels. The ContikiMAC Radio Duty Cycling Protocol. Technical Report T2011:05, Swedish Institute of Computer Science (SICS), December 2011. Available from: http://soda.swedish-ict.se/5128/1/contikimac-report.pdf.

[272] Jonathan W. Hui (Ed.) and Pascal Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. Internet Requests for Comments: 6282, September 2011. Available from: http://tools.ietf.org/html/rfc6282.

[273] Carsten Bormann. Getting Started with IPv6 in Low-Power Wireless "Personal Area" Networks (6LoWPAN), March 2011. Available from: http://6lowpan.net/wp-content/uploads/2011/03/6lowpan-tutorial-ietf80-7-sanitized.pdf.

[274] Matthias Kovatsch. CoAP for the Web of Things: From Tiny Resource-constrained Devices to the Web Browser. In *Proceedings*

of the 4th International Workshop on the Web of Things (WoT 2013), Zurich, Switzerland, September 2013.

[275] Zach Shelby. Embedded Web Services. *IEEE Wireless Communications*, 17(6):52–57, December 2010.

[276] Simon Duquennoy, Niklas Wirström, Nicolas Tsiftes, and Adam Dunkels. Leveraging IP for Sensor Network Deployment. In *Proceedings of the Workshop on Extending the Internet to Low power and Lossy Networks (IP+SN 2011)*, Chicago, IL, USA, April 2011.

[277] Walter Colitti, Kris Steenhaut, and Niccolo De Caro. Integrating Wireless Sensor Networks with the Web. In *Proceedings of the Workshop on Extending the Internet to Low power and Lossy Networks (IP+SN 2011)*, Chicago, IL, USA, April 2011.

[278] Walter Colitti, Kris Steenhaut, Niccolò De Caro, Bogdan Buta, and Virgil Dobrota. Evaluation of Constrained Application Protocol for Wireless Sensor Networks. In *Proceedings of the 18th IEEE Workshop on Local & Metropolitan Area Networks (LANMAN)*, pages 1–6, Chapel Hill, NC, USA, October 2011.

[279] ISO/IEC. Information technology – Home Electronic Systems (HES) – Part 3-10: Wireless Short-Packet (WSP) protocol optimized for energy harvesting – Architecture and lower layer protocols. ISO/IEC 14543-3-10:2012, March 2012. Available from: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=59865.

[280] EnOcean Alliance – Technical Task Group Interoperability. EnOcean Equipment Profiles (EPP) Version 2.6.3, June 2015. Available from: http://www.enocean-alliance.org/eep/.

[281] ZigBee Alliance. New ZigBee PRO Feature: Green Power. White Paper, December 2012. Available from: https://docs.zigbee.org/zigbee-docs/dcn/12/docs-12-0646-01-0mwg-new-zigbee-pro-feature-green-power.pdf.

[282] Juan Jose Echevarria, Jonathan Ruiz-de Garibay, Jon Legarda, Maite Álvarez, Ana Ayerbe, and Juan Ignacio Vazquez. WebTag: Web Browsing into Sensor Tags over NFC. *Sensors*, 12(7):8675–8690, June 2012.

[283] Yaron Y. Goland. Multicast and Unicast UDP HTTP Messages. Internet Draft, October 2000. Available from: http://tools.ietf.org/html/draft-goland-http-udp-01.

[284] Tathagata Das, Prashanth Mohan, Venkata N. Padmanabhan, Ramachandran Ramjee, and Asankhaya Sharma. PRISM: Platform for Remote Sensing Using Smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 63–76, San Francisco, CA, USA, 2010. ACM.

[285] Zach Shelby and Cedric Chauvenet. The IPSO Application Framework. IPSO Alliance, August 2012. Available from: http://www.ipso-alliance.org/wp-content/media/draft-ipso-app-framework-04.pdf.

[286] Open Mobile Alliance. Lightweight Machine to Machine Technical Specification – Candidate Version 1.0, December 2013. Available from: http://technical.openmobilealliance.org/Technical/technical-information/release-program/current-releases/oma-lightweightm2m-v1-0.

[287] ITU. Internet of Things Global Standards Initiative. Web page. Available from: http://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx.

[288] IEEE-SA. Internet of Things. Web page, August 2014. Available from: http://standards.ieee.org/innovate/iot/.

[289] Mike Belshe, Roberto Peon, and Martin Thomson (Ed.). Hypertext Transfer Protocol Version 2 (HTTP/2). Internet Requests for Comments: 7540, May 2015. Available from: http://tools.ietf.org/html/rfc7540.

[290] Philipp Bolliger and Benedikt Ostermaier. Koubachi: A Mobile Phone Widget to enable Affective Communication with Indoor Plants. In *Adjunct Proceedings of MobileHCI, Mobile Interaction with the Real World Workshop (MIRW 2007)*, Singapore, September 2007.

[291] Silvia Santini, Benedikt Ostermaier, and Andrea Vitaletti. First Experiences Using Wireless Sensor Networks for Noise Pollution Monitoring. In *Proceedings of the 3rd ACM Workshop on*

Real-World Wireless Sensor Networks (REALWSN'08), Glasgow, United Kingdom, April 2008.

[292] Benedikt Ostermaier and Philipp Bolliger. Creating Location-based Services by utilising a Web of Places. In *2nd International Workshop on SensorWebs, Databases and Mining in Networked Sensing Systems (SWDMNSS) at SAINT 2008*, Turku, Finland, June 2008.

[293] Silvia Santini, Benedikt Ostermaier, and Robert Adelmann. On the Use of Sensor Nodes and Mobile Phones for the Assessment of Noise Pollution Levels in Urban Environments. In *Proceedings of the Sixth International Conference on Networked Sensing Systems (INSS 2009), Pittsburgh, PA, USA*, June 2009.

[294] Matthias Kovatsch, Simon Mayer, and Benedikt Ostermaier. Moving Application Logic from the Firmware to the Cloud: Towards the Thin Server Architecture for the Internet of Things. In *Proceedings of the 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS 2012), Palermo, Italy*, July 2012.