# Moving Application Logic from the Firmware to the Cloud: Towards the Thin Server Architecture for the Internet of Things

Matthias Kovatsch, Simon Mayer, Benedikt Ostermaier
*Institute for Pervasive Computing*
*ETH Zürich*
*Zurich, Switzerland*
*Email: {kovatsch, simon.mayer, ostermaier}@inf.ethz.ch*

***Abstract*—Unlike traditional networked embedded systems, the Internet of Things interconnects heterogeneous devices from various manufacturers with diverse functionalities. To foster the emergence of novel applications, this vast infrastructure requires a common application layer. As a single global standard for all device types and application domains is impracticable, we propose an architecture where the infrastructure is agnostic of applications and application development is fully decoupled from the embedded domain. In our design, the application logic of devices is running on application servers, while thin servers embedded into devices export only their elementary functionality using REST resources. In this paper, we present our design goals and preliminary results of this approach, featuring the Californium (Cf) CoAP framework.**

***Keywords*-Next generation networking; software architecture; representational state transfer; wireless sensor networks;**

## I. Introduction

The recent years of research in the field of wireless sensor networks (WSNs) have led to the standardization of Internet technology for constrained embedded devices[1] with *6LoWPAN* as central standard [10]. While consensus was found up to the transport layer to interconnect these devices in an Internet of Things (IoT), an open application layer is yet to emerge. We are working on an architecture that fully complies with the de facto application layer of the Internet, the World Wide Web.

The idea of this Web of Things (WoT) [2, 19] started with smart gateways that run a Web server and provide access to different devices in a RESTful manner [3, 18]. The gateways can shield resource-constrained devices from too many requests, bridge between different communication technologies, and provide additional services for device management and discovery. On the downside, however, changes in the application and device capabilities also affect these gateways, as they carry a piece of the application logic.

To push the WoT idea one step further, i.e., the servers directly onto the devices, a working group of the Internet Engineering Task Force (IETF) is currently standardizing the

[1]In this paper, we are considering the 8-bit microcontroller class with about hundreds of kilobytes of ROM and tens of kilobytes of RAM.

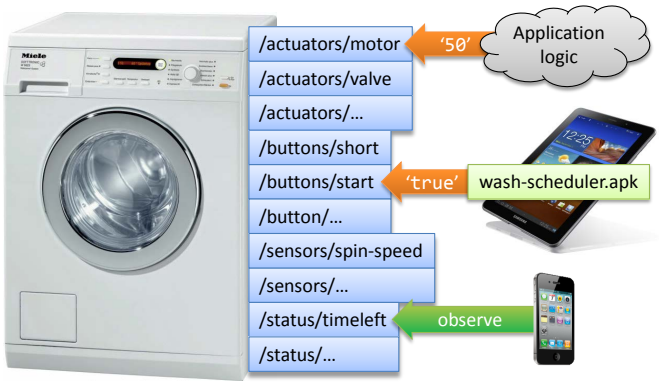



Figure 1. The thin server architecture to the extreme: A washing machine whose every button, actuator, and sensor is modeled as a resource. The application logic runs in the Cloud.

*Constrained Application Protocol* (CoAP) [14]. Being Web-oriented, this protocol also leverages the REST architectural style to interact with resource-constrained devices. In addition, it provides native push notifications [5] and efficient UDP-based group communication. A transparent mapping to HTTP facilitates Web integration through application-independent proxies, which may also reduce the load for embedded devices through the REST caching mechanisms.

To foster interoperability at the application layer, we propose the *thin server architecture* that—analog to the thin client architecture—relieves embedded servers from the burden of application logic, but rather utilizes them as a wrapper for the device's sensors and actuators. Application servers then provide the logic, leveraging not only the RESTful API provided by the device in question, but potentially also other resources of the Web.

For example, think of a washing machine where every button, actuator, and sensor is modeled as a resource (c.f., Figure 1). The machine does not, however, store any washing program locally. Whenever a button is pressed, a request is sent to an application on the Web that hosts the washing machine's decision logic. While this approach seems unconventional, it offers a number of benefits: Washing programs may be optimized continuously, new features may

be introduced, other applications may leverage functionality of the machine (e.g., send notifications when the washing program has finished) and the machine can be monitored continuously to detect possible defects early. In contrast, today's washing machines are offline and delivered with a firmware that can only be changed by a trained service technician.

We argue that scripting languages are a suitable solution for specifying the application logic of devices, as they are relatively easy to understand, widely deployed and already used for similar purposes. To describe this architecture that is based on thin servers and script-based application logic in the cloud, we first introduce our design goals, then describe the proposed architecture in greater detail, and finally present our preliminary results in an effort to create a prototypical infrastructure based on the described concepts.

## II. Design Goals

As a basis, we build upon a strictly layered architecture for the Internet of Things to handle complexity and interoperability. Energy concerns, for instance, can be handled by an application-independent radio duty cycling layer, which also enables multiple applications within the same low-power network. In addition, we have the following motivation for our proposed architecture.

### A. Full Web Integration

The vision of the IoT is to extend the virtual world into the real world and augment physical everyday objects with additional services. This development brings with it the ability to manage real-world processes in real-time and in an automated manner that allows computers to react and adapt to physical phenomena in the real world. The World Wide Web evolved to de facto application layer of the Internet and most large-scale distributed applications are nowadays Web-based. A full and seamless integration of embedded systems with the Web, leveraging the plethora of existing services and tools, is thus a central goal of our architecture.

### B. Intuitive APIs

The big players in the Web sphere mostly define their own application programming interfaces (APIs) for their public services, and still are able to interconnect seamlessly. Also for third parties, the integration of different services is comparatively easy. This is possible because of the REST constraint for uniform interfaces. The APIs are compatible to each other, but their individual interfaces are designed according to the service semantics. Thus, they can provide self-descriptive names and intuitive interaction, which in turn facilitates productive usage, enables better assertions of the correct behavior for complex distributed systems, and eventually lowers the integration costs.

### C. Decoupling of Infrastructure and Applications

The Internet was able to evolve because of a simple constraint: the *end-to-end principle*. Intermediate nodes are kept free from application-specific functions and only serve as transport between the end hosts of the network. A similar principle is required for the IoT infrastructure. The vast number of nodes providing sensing and actuation capabilities cannot be reprogrammed for every change of the application. Application-specific functionality on these nodes would also prevent concurrent applications from leveraging the same infrastructure. Even though less in number, the same applies for application-specific gateways. Thus, devices in the IoT must be agnostic of the application. They must be constrained to simple intermediate nodes to the physical world through their sensors and actuators.

### D. End-User Programming

Although most functionality of today's consumer devices is realized in software, users are limited to use products as designed by the manufacturers. By providing an interface to the elementary functions of a device, users can change and extend its functionality and smoothly integrate it with existing infrastructure without modifying the original device firmware. This adds significant value to the product since users can adopt their devices to their specific needs. Also, by publishing the added functionality, the value of the whole infrastructure increases according to *Metcalfe's law*.

## III. Architecture

### A. Thin Server Model

Based on the idea of *thin clients*, we define a *thin server* as a device in the role of a server that does not host any application logic. This enables an IoT infrastructure that is agnostic of applications. Corresponding to a thin client, which is only equipped with the necessary interfaces to interact with the user such as display and keyboard, thin servers only provide a low-level API to their elementary functionality, such as sensor and actuator access and configuration of device parameters to interact with the physical world. A central implication is that a device can continuously evolve during its lifetime, as new functionality can be added by providing a new app. Not only can an application like an updated washing program enhance the environmental sustainability, but also extend usage of the hardware itself.

The thin server model introduces new challenges that have to be addressed. An open question is where exactly to make the cut in the functionality and place the API. Considering the washing machine example from the beginning, a central constraint is hard real-time for control loops between sensors and actuators. Often, this cannot be met because of the round trip times to the Cloud service. Thus, these control loops must be realized locally and only their parameters can be exported through the API. A second challenge is safety, which must be ensured independent from fluctuating network link

quality. For any API input the washing machine must enforce local safety rules, for instance to avoid escape of water due to improper valve control or hazardous imbalance by an overdriven motor. The security and privacy challenges are independent from the thin server model, as the requirements apply to any architecture. Given a specific hardware platform class, however, the model has the advantage that hardware resources that are freed by removing the application logic can be allocated for the security suite.

### B. Interface Model

Within our architecture, the APIs of thin servers and apps are RESTful and usually implemented with CoAP, but HTTP is also used, for instance for more powerful devices such as low-power Wi-Fi systems-on-a-chip [11]. To ease the job for developers, our design only employs human-readable, self-descriptive APIs supported by concise descriptions like the CoRE Link Format [13] and Microformats[2]. This also enables tech-savvy users to understand in which way their devices publish and consume data, so they can write scripts to modify or extend the functionality. The gained flexibility comes at the price of glue code to integrate a system from different manufacturers. Web development has shown, however, that the integration costs are low and can be done by developers with different backgrounds. Ideally, tools can automatically create components for graphical programming languages like *Yahoo! Pipes* or *ClickScript* only from the descriptions.[3]

This is a compromise between fully specified device profiles and mobile code as the other extreme, where devices completely lack the notion of their provided service. Device standards by industry alliances traditionally strictly specify the possible actions within an application domain and force devices into knowing these profiles. In the IoT, however, devices are not part of a specific application domain and the number of involved manufacturers and device types is expected to be vast. We therefore argue that no global standard can emerge to manage this heterogeneity and rather advocate the implementation of self-descriptive, REST-based interfaces that can be understood and used by humans as well as devices in M2M scenarios.

This does not mean, however, that devices in our architecture must not support standards for their primary application domain. Recently, industry alliances also employ RESTful interfaces as a basis for their profiles such as the ZigBee Smart Energy Profile 2.0[4] or the IPSO Profile[5]. Unfortunately, they do not fully carry out the incorporated Web concepts. The protocol overhead for human-readable, self-descriptive URIs goes to waste by using cryptic one-/two-letter identifiers, which have no advantage over traditional, binary-coded profile protocols. Still, both profiles can be supported by the thin server model through a simple proxy that converts the low-level REST API to the specified resources and representations of the profile. Such proxies can also be realized as an app and thus retrofitted, again extending the lifetime of a device.

[2]http://microformats.org

[3]http://pipes.yahoo.com/pipes, http://clickscript.ch

[4]http://www.zigbee.org/Standards/ZigBeeSmartEnergy

[5]http://www.ipso-alliance.org/technical-information

### C. App Model

The app model enables the reuse of deployed devices for different applications without changing the firmware [6] and leverages a programming model similar to Web 2.0 mashups. For interoperability and multiple concurrent applications that leverage the same infrastructure, it relies on the thin server model. With this setup, apps implement the application logic separated from the firmware of the devices and run in the Cloud. The latter can be a remote server by a service provider or a local host that is always available, such as a router or a digital video recorder. The separation from the firmware also allows developers that are not specialized in the embedded domain to create applications using commonly known languages and tools.

### D. Infrastructure Integration and Discovery

A general requirement for IoT architectures is to support the management and look-up of devices, which requires a robust discovery mechanism to integrate newly arriving devices into the infrastructure and remove them once they become inactive or disconnected. To enable user-friendly and efficient look-up, the functionality offered by devices and apps has to be extracted and stored as service representations. Our approach to enable this is to embed metadata like names, brands, tags, or geographical information into the resource representations. These annotations, however, should be simple enough to allow not only programmers but also tech-savvy Web developers to annotate smart things and thus help foster a community around the generation and publishing of device-related metadata. Though, care must be taken to not overload constrained devices with metadata, which would have severe effects on network performance and device battery consumption. Instead, we propose to provide annotations separately and link them to the associated device using Web Linking techniques specified in the CoRE Link Format [13].

We have been exploring the use of different lightweight markup languages, like *Microformats* and *Microdata*[6], to annotate the services offered by our devices. To this end, the development to using *Microdata* is gaining traction in the research community as well as industry, as many vocabularies have started to emerge. They define extensive collections of concepts like *Person*, *Event*, or *Organization* (e.g., *data-vocabulary.org* or *schema.org*). While many of the definitions in these initiatives are based on earlier

[6]http://www.whatwg.org/specs/web-apps/current-work

and less successful formats like *FOAF*[7], these vocabularies attract a lot of attention in the industrial domain and are supported by big players like Google and Yahoo!. This represents a trend towards creating community metadata that is, rather than being based on rigid and globally standardized models or types, driven by bottom-up annotation of resources according to publicly agreed-upon definitions. To our knowledge, however, there is currently no effort to standardize a description vocabulary for services provided by physical devices, which could be a practicable approach to creating lightweight, easily understandable, and expressive annotations for smart things.

We have created a discovery system for smart things and their resources that is based on the application of multiple semantic identification strategies (e.g., for recognizing *Microdata*) onto the resource representation [9]. This system is designed to enable the creation and updating of strategies at runtime and thus is decoupled from any specific annotation format for resources. When the system is not yet able to understand annotations provided by a smart thing, a new strategy for interpreting the contained resources' annotations can be published to our system, which allows for a future-proof resource discovery service. While this approach is well integrated with the WoT initiative, it is not yet in line with the concepts for constrained RESTful environments. Here, essentially, a transparent mapping to the CoRE Link Format has to be established. In the other direction, WoT device discovery can make use of the *CoRE Resource Directory* [15] to discover new smart things and find the entry point to their REST interface.

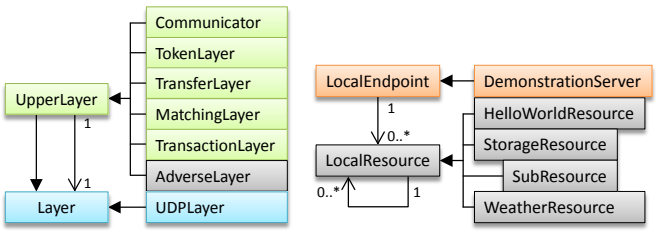
## IV. Preliminary Results

We have implemented several building blocks of a prototypical infrastructure and testbed for our architecture. As a case study for our upcoming evaluation, we chose smart home environments. Private homes provide a plethora of heterogeneous device types such as household appliances, multimedia equipment, sports and simple healthcare products, toys, and home automation sensors and controllers.

### A. Back-end Framework: Californium (Cf)

One main contribution of this paper is *Californium (Cf)*, a CoAP framework in Java. It is designed for back-end clients and servers, as well as CoAP proxies. To fulfill the different roles, *Cf* follows a modular design. The stack is configurable through different *Communicator* classes. The default configuration is shown in Figure 2a. The gray *AdverseLayer* is included in the framework, but not active by default. It allows to test the behavior under different packet loss rates. New layers (e.g., for DTLS) can easily be added, as any of the *UpperLayer* classes can be stacked on top of each other. Only the base layer connecting to the transport system is fixed.

[7]http://www.foaf-project.org




(a) Cf Stack (b) Cf Server

Figure 2. The Californium framework can fulfill three different roles: (a) CoAP server in the back-end, (b) CoAP client for unconstrained environments, and (c) CoAP-HTTP proxy. At the time of writing, caching and the CoAP-to-HTTP mapping are not included.

*Californium* is designed for rapid deployment of CoAP resources in the back-end. A server extends the *LocalEndpoint* class and defines the initial resources in its constructor. Resources can again have multiple sub-resources, which can also be added dynamically. Figure 2b illustrates a subset of our demonstration server[8]. The framework also provides a stub to handle the creation of resources via PUT when resources along the path do not exist at the time the request is received. Depending on the visibility, *Cf* will automatically create the CoRE Link Format for `/.well-known/core`.

Full proxy capabilities with caching and full HTTP mapping have not yet been implemented. Still, *Californium* can already bridge between HTTP and CoAP using the default Java *HttpServer*. Such a setup was shown by another group that already uses the framework [1]. The source code is publicly available on GitHub[9].

At the time of writing, we were in the process of integrating JavaScript support into *Californium*. Scripted apps are instantiated as a CoAP resource of an app server and can easily export user-defined information through sub-resources. For client functionality, we are specifying a `coapRequest` object API similar to the `xmlHttpRequest` object.

### B. Smart Appliances

Our second building block are prototypes of smart appliances based on different platforms. Figure 3 shows the prototypes created to date: several *Tmote Skys* that serve as simple sensors and routers, *smart thermostats* on each radiator in our office area using the Dresden Elektronik deRFmega128[10] wireless module, and *smart power outlets* for electricity metering and switching using modified Ploggs[11] with an AVR Jackdaw. These are based on the *Erbium (Er)* REST Engine [7]. In addition to the device functionality, we export debug information such as the neighbor table and a resource to reconfigure the IEEE 802.15.4 channel during runtime.

[8]coap://vs0.inf.ethz.ch:5683
[9]https://github.com/mkovatsc/Californium
[10]http://www.dresden-elektronik.de/shop/prod148.html?language=en
[11]http://www.plogg.co.uk

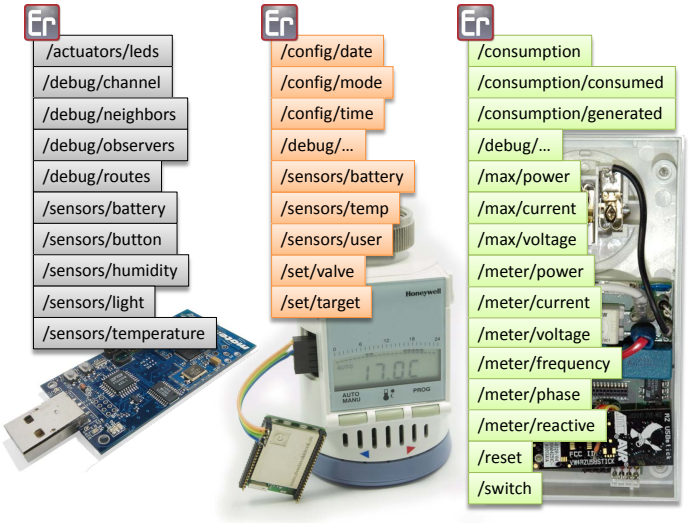



Figure 3. Prototypes of smart appliances that follow the thin server model and export all their functionality through a RESTful API. All 'volatile' values such as temperature or power consumption support CoAP's observe option. The Tmote Skys provide all onboard functionality, the thermostats can be set to a target temperature or controlled manually by setting the valve. The smart power outlets measure current values and cumulative consumption.

Technically speaking, all these appliances are 'dumb,' as they implement the thin server model. From a user perspective, however, they appear smart due to apps running on top of the sensing and actuation capabilities they provide. The appliances already serve a few *Cf*-based applications following the app model. The remote control and logging of the thermostats and power outlets is straight-forward. Another application uses the temperature values from the thermostats to calibrate the temperature sensor of the Tmote Skys: Due to an unfortunate design of the sensor nodes, their power supply heats up the onboard STH11 and the formula from the datasheet cannot be directly applied. The required correction function also varies for battery- and USB-powered motes. The upcoming scripting support for *Californium* will foster additional applications that leverage the thin server model.

### C. Configuration and Management Tool: Copper (Cu)

A central tool in the WoT idea is the Web browser, as it enables the inspection of RESTful Web services. We created *Copper*, a corresponding tool for constrained environments based on CoAP. With it, developers can explore, test, and configure the elementary device APIs and the applications running in the back-end. Users can use it directly as universal remote control for their devices. *Copper* is implemented as a Firefox extension, so CoAP resources integrate fully with the main Web tool, the browser. CoAP links and bookmarks can be used in the normal way. *Cu* provides fine-grained options to customize outgoing requests and renderers for several content-types of responses as shown in Figure 4.

## V. Related Work

Many other concepts and frameworks exist that aim to enable the interconnection of real-world devices and their sensors and actuators within ecosystems of smart things. The "Web of Things plug and play experience" [17] requires all participating parties to have a common understanding about how resources should be addressed and controlled and about the handling of event notifications. The authors discuss a support infrastructure for interconnected Web-enabled devices that represents devices as resources and manages discovery using a centralized resource repository that also holds data about the context of available sensors and actuators. For interaction with the resources, a REST API is used and notifications are channelled over an XML-RPC-based push mechanism (*blog ping*).

The *WebPlug* framework [12] extends this idea of a plug and play experience to provide a generic approach for the interaction with Web-enabled objects and the construction of mashups that incorporate functionality provided by their sensors and actuators. *WebPlug* describes a number of components like typed resources, collections, or pollers (which emulate push-functionality for resources outside the framework) and connects them using the Observer pattern via URL callbacks to propagate knowledge about events. The framework follows a uniform URL-based approach that does not depend on HTTP extensions nor on a single serialization format like Atom to implement versioning and the manipulation of collections.

The *Simple Measurement and Actuation Profile* (sMAP) [16] has been designed to relay real-world information from sensors and actuators over HTTP using defined JSON schemas. Resources are modeled with URIs in the form of `point/type/channel`, where `point` represents a

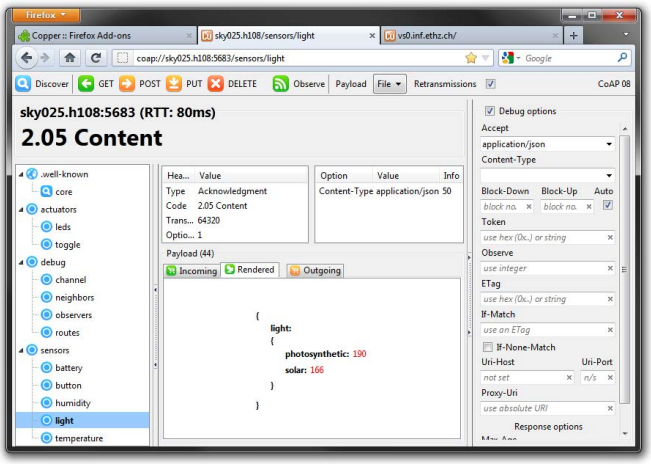



Figure 4. The Copper (Cu) CoAP user-agent allows for direct interaction with networked embedded devices. It is a tool for configuration and assessment of correct system behavior like the Web browser for classic RESTful Web services.

physical point of information, `type` specifies either a meter, sensor, or actuator, and `channel` represents a particular stream of readings (e.g., temperature). The readings themselves are sent as JSON objects with strict formats and data semantics that are defined in a set of JSON schemas. The architecture supports resource-constrained devices through proxies that translate between JSON and binary JSON. The ideas presented in *sMAP* are somewhat reminiscent of earlier attempts to establish Web Service specifications for devices. The *Devices Profile for Web Services* (DPWS) initiative defined implementation constraints for SOAP Web services so they can be used in machine-to-machine communication that is more constrained than server systems. Current research in this field targets the adaptation of DPWS for highly resource-constrained devices like the Tmote Skys used in our architecture [8].

A very different approach to enabling ecosystems of sensor nodes is *dinam-mite* [4]. It is a fully self-contained wireless sensor network development environment that eliminates the setup overhead for the development infrastructure. The entire tool-chain is served by each sensor node, including the IDE, libraries, code, data, and visualization. While this setup is very appropriate regarding the simplified rapid prototyping of applications on powerful motes (e.g., 80 MHz PIC32 microcontroller with 128 kB of program memory, 32 kB of RAM and additional storage memory [4]), it cannot be run on the inexpensive, resource-constrained nodes envisioned in our proposed architecture.

## VI. Conclusion and Future Work

The proposed *thin server architecture* promotes a Web-like application layer with a common programming model for constrained networked embedded devices. The key idea is to separate the application logic from the firmware to enable application developers with different backgrounds to provide applications for the heterogeneous device types from many different vendors found in an Internet of Things environment. With our prototypical infrastructure, we showed how the presented design goals can be realized and that the architecture is feasible. Devices running the *Erbium (Er)* REST Engine export their functionality without hosting any application logic. Thus, they can serve multiple applications running on the back-end without reprogramming. Also multiple concurrent applications are supported, for safe operations. The applications could be developed without any knowledge of the embedded domain and details of the system running on the devices. The implementation of the application logic itself is fostered by the presented *Californium (Cf)* CoAP framework, which provides a large code base for clients, servers, and proxies. While we acknowledge that security is a crucial aspect that has to be addressed in order to enable a wide real-world deployment of an IoT architecture, we do not address this issue at this time.

As a next step we will fully integrate scripting support for the application logic. With that, we will be able to develop various application classes for the evaluation of the architecture in our testbed. A comparison with a native implementation of selected application kernels will show the feasibility and limitations of our architecture, depending on the application class.

## References


[1] W. Colitti, N. De Caro, J. Tiete, H. Phung, K. Steenhaut, and A. Touhafi. Demo Abstract: Enabling Transparent WSN Resource Access via RESTful Web Services. In *Proc. EWSN*, Trento, Italy, 2012.

[2] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin. pREST: A REST-based Protocol for Pervasive Systems. In *Proc. MASS*, Fort Lauderdale, FL, USA, 2004.

[3] R. T. Fielding and R. N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, 2002.

[4] D. Gordon, M. A. Neumann, and M. Beigl. Demo Abstract: Program Your Reality with dinam-mite. In *Proc. Pervasive*, San Francisco, CA, 2011.

[5] K. Hartke. Observing Resources in CoAP. draft-ietf-core-observe-05, 2012.

[6] M. Kovatsch. Firm Firmware and Apps for the Internet of Things. In *Proc. SESENA*, Honolulu, HI, USA, 2011.

[7] M. Kovatsch, S. Duquennoy, and A. Dunkels. A Low-Power CoAP for Contiki. In *Proc. MASS*, Valencia, Spain, 2011.

[8] C. Lerche, N. Laum, G. Moritz, E. Zeeb, F. Golatowski, and D. Timmermann. Implementing Powerful Web Services for Highly Resource-Constrained Devices. In *PERCOM Workshops*, Seattle, WA, USA, 2011.

[9] S. Mayer and D. Guinard. An Extensible Discovery Service for Smart Things. In *Proc. WoT*, San Francisco, USA, 2011.

[10] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC4944, 2007.

[11] B. Ostermaier, M. Kovatsch, and S. Santini. Connecting Things to the Web using Programmable Low-power WiFi Modules. In *Proc. WoT 2011*, San Francisco, CA, USA, 2011.

[12] B. Ostermaier, F. Schlup, and K. Römer. WebPlug: A Framework for the Web of Things. In *Proc. WoT*, Mannheim, Germany, 2010.

[13] Z. Shelby. CoRE Link Format. draft-ietf-core-link-format-11, 2012.

[14] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Constrained Application Protocol (CoAP). draft-ietf-core-coap-08, 2011.

[15] Z. Shelby and S. Krco. CoRE Resource Directory. draft-shelby-core-resource-directory-02, 2011.

[16] D. C. Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz. sMAP: A Simple Measurement and Actuation Profile for Physical Information. In *Proc. SenSys*, Zurich, Switzerland, 2010.

[17] V. Stirbu. Towards a RESTful Plug and Play Experience in the Web of Things. In *Proc. ICSC*, Los Alamitos, CA, USA, 2008.

[18] V. Trifa, S. Wieland, D. Guinard, and T. M. Bohnert. Design and Implementation of a Gateway for Web-based Interaction and Management of Embedded Devices. In *Proc. IWSNE*, Marina del Rey, CA, USA, 2009.

[19] E. Wilde. Putting Things to REST. Technical Report 2007-015, School of Information, UC Berkeley, Berkeley, CA, USA, 2007.