Architecting a mashable open world wide web of things

Architecting a Mashable Open World Wide Web of Things

Dominique Guinard, Vlad Trifa
Institute for Pervasive Computing, ETH Zurich
and SAP Research CEC Zurich
8092 Zurich, Switzerland
Email: {dguinard,trifam}@ethz.ch

Erik Wilde
School of Information, UC Berkeley
Berkeley CA 94720, USA
Email: dret@berkeley.edu

***Abstract*—Many efforts are currently going towards networking smart things from the physical world (e.g. RFID, wireless sensor and actuator networks, embedded devices) on a larger scale. Rather than exposing real-world data and functionality through proprietary and tightly-coupled systems we propose to make them an integral part of the Web. As a result, smart things become easier to build upon. Popular Web languages (e.g. HTML, URI, JavaScript, PHP) can be used to build applications involving smart things and users can leverage well-known Web mechanisms (e.g. browsing, searching, bookmarking, caching, linking) to interact and share things. In this paper, we begin by describing a Web of Things architecture and best-practices rooted on the RESTful principles that contributed to the popular success, scalability, and evolvability of the traditional Web. We then discuss several prototypes implemented using these principles to connect environmental sensor nodes, energy monitoring systems and RFID tagged objects to the World Wide Web. We finally show how Web-enabled things can be used in lightweight ad-hoc applications called "physical mashups".**

I. Introduction

A central concern in the area of pervasive computing has been the integration of digital artifacts with the physical world. In particular, the "Internet of Things" has essentially explored the development of applications built upon various networked physical objects [1]. Inhabitants of the physical world such as sensor and actuator networks, embedded devices, appliances and everyday digitally enhanced objects (subsequently called *smart things*) are still mostly disconnected from the Web and form a myriad of small incompatible islands. Increasingly, embedded devices and consumer electronics as for example Chumby, Gumstix, or Nabaztag get Internet (and sometimes Web) connectivity but cannot be controlled and monitored without using dedicated software and proprietary interfaces. As a consequence, smart things are hard to integrate into composite applications, which severely hinders the realization of a flexible ecosystem of devices that can be reused serendipitously.

The Internet of Things has mainly focused on establishing connectivity in a variety of challenging and constrained networking environments, and the next logical objective is to build on top of network connectivity by focusing on the application layer. In the Web of Things (WoT), we would like to consider smart things as being first-class citizens of the Web. This way Web tools and techniques (e.g. browsers, search engines, caching systems), languages (e.g, HTML, JavaScript, mashups) and interaction techniques (e.g. browsing, linking, bookmarking) can be directly applied to the real world. We position the Web of Things as a refinement of the Internet of Things by integrating smart things not only to the Internet (i.e. to the network), but also to the Web (i.e. to the application layer).

To achieve this goal, we propose to reuse and adapt patterns commonly used for the Web, and introduce in the paper an architecture for the Web of Things. First, by embedding Web servers [2], [3], [4] on smart things and applying the REST architectural style [5], [6] to the physical world (see Section III-A). The essence of REST is to focus on creating loosely coupled services on the Web so that they can be easily reused [7]. REST is actually core to the Web and uses URIs for encapsulating and identifying services on the Web. In its Web implementation it also uses HTTP as a true application protocol. It finally decouples services from their presentation and provides mechanisms for clients to select the best possible formats. This makes REST an ideal candidate to build an "universal" API (Application Programming Interface) for smart things.

As the "client-pull" interaction model of HTTP does not fully match the needs of event-driven applications, we further suggest the use of syndication techniques such as Atom to enable sensor push interactions (see Section III-B).

As a consequence of the proposed architecture, smart things and their functionality get transportable URIs that one can exchange, reference on Web sites and bookmark. Things are then also linked together enabling discovery simply by browsing. The interaction with smart things can also almost entirely happen from the browser, a tool that is ubiquitously available and that most of the people understand well [8]. Furthermore, smart things can benefit from the mechanisms that made the Web scalable and successful such as caching, load-balancing, indexing and searching.

Since some devices cannot connect to the Internet or fully respect the REST architectural style, we finally propose the use of Smart Gateways [9] which are embedded Web servers that abstract communication and services of non Web-enabled devices behind a RESTful API (see Section IV). In Section V we illustrate the WoT architecture by means of several prototypes.

We begin by applying these to sensors, actuators and tagged objects and show how simple Web applications (e.g. AJAX, JavaScript, HTML) can be built upon these smart things. We then show that a Web of Things makes it possible for tech-savvy end-users to create *physical mashups* involving smart objects just as they would create Web mashups.

II. Related Work

Linking the Web and physical objects is not a new idea. Early approaches started by attaching physical tokens (such as barcodes) to objects to direct the user to pages on the Web containing information about the objects [10], [11]. These pages were first served by static Web Servers on mainframes, then by early gateway system that enabled low-power devices to be part of wider networks [12]. The key idea of these work was to provide a virtual counterpart of the physical objects on the Web. URIs to Web pages were scanned by users e.g. using mobile devices and directed them to online representation of real things (e.g. containing status of appliances on HTML pages or user manuals). With advances in computing technology, tiny Web servers could be embedded in most devices [3], [2]. The Cooltown project pioneered this area of the physical Web by associating pages and URIs to people, places and things [8] and implementing scenarios where this information could by physically discovered by scanning infrared tags in the environment. We would like to go a step further and to propose an architecture to truly make smart things part of the Web so that they *proactively serve their functionality as re-usable Web services.*

A number of projects proposed solutions to expose the functionality of smart things in order to build applications upon. Among them, JINI, UPnP, DNLA, etc. JXTA [13] is a set of open protocols for allowing devices to collaborate in a peer-to-peer fashion, and was eventually the first attempt to bridge the physical objects world over the Internet. The advent of WS-* Web Services (SOAP, WSDL, etc.) led to a number of work towards deploying them on embedded devices and sensor networks [14], [15]. While helping towards the integration to enterprise applications, these solutions are often too heavy for devices with limited capabilities [4], do not directly expose the smart things' functionality on the Web as RESTful architectures do and are not truly loosely-coupled [7]. Several systems for integration of sensor systems with the Internet have been proposed — for example SenseWeb [16] and Pachube — which offer a platform for people to share their sensory readings using Web services to transmit data onto a central server. Unlike the Web of Things, these approaches are based on a centralized repository and devices are considered as passive actors only able to push data.

One of the first mentions of a Web of Things composed of RESTful smart things comes from [17]. However it focuses mainly on the discovery of devices and not on how to provide their functionality on the Web. Closer to our work, [18] and in particular [19] consider the use of REST-like architectures for sensor networks. We build upon these approaches and propose a systematic implementation of the RESTful constraints (see Section III-A) and extend the model with the use of standard Web syndication such as using Atom. Furthermore we do not focus on the lower sensors level but explore the applications from a Web view-point. We propose a unified view of the Web of today and tomorrow's Web of Things in applications called "physical mashups".

III. Web of Things Architecture

Realization of the Web of Things requires to extend the existing Web so that real-world objects and embedded devices can blend seamlessly into it. Instead of using the Web protocols solely as a transport protocol — as done when using WS-* Web services for instance — we would like to make devices an integral part of the Web by using HTTP as an application layer protocol.

The main contribution of the "Web of Things" approach is to take the next logical step beyond the network connectivity established by activities often summarized under the "Internet of Things" label. Many activities in the "Internet of Things" area put their emphasis on establishing Internet-level connectivity (often in terms of TCP and/or UDP), and then propose interaction protocols layered on top of this basic connectivity. Often, these protocols follow RPC-style designs, introducing their own functions and thus requiring any users of these protocols to specifically support the functions provided by these platforms. We propose to follow a different architectural style and to use REST's idea of a uniform interface, so that the interactions with smart things can be built around universally supported methods [7].

We do not make the assumption that devices must offer RESTful interfaces directly provided by each individual thing. In a number of cases it makes a lot of sense to shield the implementation of a specific platform in terms of implementation specifics, and to expose the resources made available by that platform through a RESTful API. The interactions behind that RESTful interface are invisible and often will include highly specialized protocols for the specific implementation scenario. Because REST has the concept of *intermediaries* as a core part of the architectural style, such a design can easily be achieved by modeling the RESTful service using intermediaries. By using either *proxies* or *reverse proxies* (see Section IV) it is furthermore possible to establish such an intermediary from the client or from the server side, effectively introducing a robust pattern of how non-RESTful services can be wrapped in RESTful abstractions.

Before we move on to discuss the Web of Things in greater detail, we will first briefly review REST's main principles, and how they apply to the vision presented in this paper.

A. A Resource Oriented Architecture for Things

REST is an architectural style, which means that it is not a specific set of technologies. For this paper, we focus on the specific technologies that implement the Web as a RESTful system, and we propose how these can be applied to the area of pervasive computing. The central idea of REST revolves around the notion of resource as *any component of an*

application that needs to be used or addressed. Resources can include physical objects (e.g. a temperature sensors) abstract concepts such as collections of objects, but also dynamic and transient concepts such as server-side state or transactions. REST can be described in five constraints:

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

C2 *Uniform Interface:* Resources should be available through a uniform interface with well-defined interaction semantics, as is *Hypertext Transfer Protocol (HTTP)*. HTTP has a very small set of methods with different semantics (*safe*, *idempotent*, and others), which allows interactions to be effectively optimized. The vast majority of Web-facing applications offer RESTful interfaces, while the backends are implemented using different interaction models (such as database systems), and the same approach can be employed for the Web of Things.

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

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

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

In HTTP, the uniform interface constraint (C2) has four main operations, `GET`, `PUT`, `POST`, and `DELETE`. In a Web of Things these map rather naturally: `GET` is used to retrieve the representation of a resource, e.g. the current consumption of an electricity sensor. `PUT` is used to update the state of an existing resource or to create a resource by providing its identifier. For example it can be used to turn a led on or off. `DELETE` is used to remove a resource. It can for example be used to delete a threshold on a sensor or to shutdown a device. Finally, `POST` creates a new resource, e.g. creates a new feed used to trace the location of a tagged object.

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

REST is still an active research topic and more complex interactions patterns (such as RESTful transactions) are not yet established as specifications or design patterns. It is important to keep in mind, though, that the design goals for the Web as a RESTful system and REST's advantages for a decentralized and massive-scale service system align very well with the field of pervasive computing: millions to billions of available resources and loosely coupled clients, with potentially millions of concurrent interactions with one service provider, and a large share of long-lasting interactions. Based on these observations, we conclude that RESTful architectures for the Web of Things are the most effective solution, because they scale much better than RPC-based architectures.

The most important step in any RESTful design is the first step of identifying all resources that should be made available, and for a Web of Things, the smart things themselves would naturally be prime candidates for this. However, sometimes these things do not exist by themselves, but within certain scenarios or groups through which they are accessed and managed, which maps very well onto the syndication concept described in the following section.

B. Syndicating Things

One common theme among many scenarios with smart things is that there are collections of things, based on certain properties or just on the application scenario. With Atom, the Web has a standardized and RESTful model for interacting with collections, and the *Atom Publishing Protocol (AtomPub)* extends Atom's read-only interactions with methods for write access to collections. Because Atom is RESTful, interactions with Atom feeds can be based on simple `GET` operations which can then be cached. More advanced scenarios can be based on feeds supporting query features, but this is an active area of research and there are not yet any standards [20]. Atom also makes it possible to support asynchronous scenarios in a WoT

where clients can monitor smart things by subscribing to feeds and pulling a feed server instead of by directly pulling data from each smart thing.

However, many pervasive scenarios must deal with real-time information. This is particularly useful when one needs to combine stored or streaming data from various sources to detect spatial or temporal patterns, as is the case in many environment monitoring applications. HTTP was designed as a client-server architecture, where clients can explicitly request (pull) data and receive it as a response. This makes REST well suited for controlling smart things over HTTP, but this client-initiated interaction models seems unsuited for event-based and streaming systems, where data must be sent asynchronously to the clients as soon as it is produced.

For scenarios requiring direct push, various research efforts are currently active. One is a model called *Comet* (also called *HTTP streaming* or *server push*) [21] which is mostly based on long-lasting HTTP interactions and tries to solve the problem purely based on existing infrastructure. An alternative approach is *PubSubHubbub*, which starts with feeds, and then adds a layered infrastructure of nodes which are forwarding notifications and accept subscribers. On a different level are HTML5's *Server-Sent Events* [22], which provide hooks to receive push notifications in regular browser-based applications.

While it is clear that a Web of Things needs more developments and standards in the areas we have described, the developments of recent years and the foreseeable future of HTML5 and richer interaction models on the Web make us optimistic that most of the problems of missing building blocks will be overcome in a relatively short period of time.

IV. Connecting Things to the Internet

To make smart things part of the Web, two solutions are possible: direct Web connectivity on the devices or through a proxy. Previous work has shown that embedded Web servers on resource constrained devices is feasible [3], [4], and it is likely that in the near future most embedded platforms will have native support for TCP/IP connectivity (in particular with 6LowPAN [2]), therefore a Web server on each device is a reasonable assumption. This approach is sometimes desirable because there is no need to translate HTTP requests from Web clients into the appropriate protocol for the different devices, thus devices can be directly integrated and make their RESTful APIs directly accessible on the Web, as shown on the right part of Figure 1.

However, when an on-board HTTP server is not possible or not desirable, Web integration takes place using a reverse proxy that bridges devices that do not talk IP with the Web. We call such as proxy a *Smart Gateway* to encapsulate the fact that it is a network component that does more than only data forwarding. A Smart Gateway is actually a Web server that abstracts behind a RESTful API the actual communication between devices and the gateway (e.g.. Bluetooth or Zigbee) through the use of dedicated drivers. From the Web clients' perspective, the actual Web-enabling process is fully transparent, as interactions are HTTP in both cases.

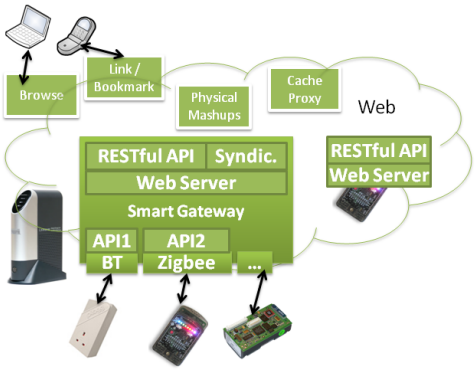

Fig. 1. Web and Internet Integration with Smart Gateways (left), direct integration (right).

As an example, consider a request to a sensor node coming from the Web trough the RESTful API. The gateway maps this request to a request into the proprietary API of the node and transmits it using the communication protocol understood by the sensor node. A Smart Gateway can support several types of devices through a driver architecture as shown on Figure 1 where the gateway supports three types of devices and their corresponding communication protocols. Ideally, gateways must have a small memory footprint to be integrated into embedded computers already present in buildings such as Wireless routers or *Network Attached Storage (NAS)* devices.

Aside from connecting limited devices to the Web, a Smart Gateway can also provide more complex functions to devices such as orchestrate the composition of several low-level services from disparate devices into higher-level services available through the RESTful API. For example, if an embedded device measures the energy consumption of appliances, the Smart Gateway could provide a service that returns the total energy consumption as a sum of the data collected by all the devices connected to the gateway.

V. Evaluation by Prototyping

In this section we present several prototypes we have developed to illustrate our proposed architecture for the Web of Things.

A. A Smart Gateway for Smart Meters

With this prototype we start by illustrating the application of the WoT architecture for monitoring and controlling the energy consumption of households. We used intelligent power sockets called *Plogg* that can measure the electricity consumption of the devices plugged into them. Each Plogg is also a wireless sensor node that communicates over Bluetooth or Zigbee. However, the integration interface offered by the Ploggs is proprietary, which makes the development of the applications using Ploggs rather tedious, and does not allow for easy Web integration.

The Web-oriented architecture we have implemented using the Ploggs is based on four main layers as shown in Figure 2. The first layer is composed of appliances we want to monitor

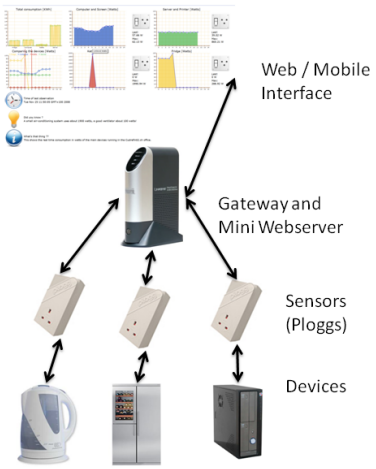

Fig. 2. Devices extended attached to the Ploggs power outlets communicating with a Smart Gateway offering the Ploggs functionalities as RESTful services.

and control through the system. In the second layer, each of these appliances is then plugged to a Plogg sensor node. In the third layer, the Ploggs are discovered and managed by a Smart Gateway as described before. The final layer is the Web user interface.

The Ploggs Smart Gateway is a C++ embedded component whose role is to automatically find all the Ploggs in the environment and make them available as Web resources. The Gateway first discovers the Ploggs on a regular basis by scanning the environment for Bluetooth devices. The next step is to make their functionality available as RESTful resources. A small footprint Web server (Mongoose[1]) is used to enable access to the Ploggs' functionalities over the Web. This is done by mapping URIs to native requests of the Plogg Bluetooth API.

Besides discovering the Ploggs and mapping their functionalities to URLs, the gateway has two other important features. First, it offers *local* aggregates of device-level services. For example, the gateway offers a service that returns the combined electricity load of all the Ploggs connected to it at any given time. The second feature is that the gateway can represent resources in various formats. By default an (X)HTML page with links to the resources is returned to ensure browsability. Using this representation the user can literally "navigate" through the structure of smart meters to identify the one she wants to use and directly test them by clicking on links (e.g. for the HTTP `GET` method) or filling forms (e.g. for the `POST` method). The gateway can also represent resources as JSON results to ease the integration with other Web applications.

To illustrate how we apply the HTTP standards and REST, let us briefly describe an example of interaction between a client application (e.g. written in AJAX) and the Ploggs' RESTful Smart Gateway. First, the client contacts the root URI of the application:

[1]`http://code.google.com/p/mongoose`

```
GET /EnergieVisible/SmartMeters/RoomLamp
[...] HTTP/1.x 200 OK
Content-Type: application/json
{
  "deviceName":  "RoomLamp",
  "currentWatts":  60.52,
  "KWh":  40.3,
  "maxWattage":  80.56
  "links":
  [{"aggregate":  "../all"},
  {"load": "../load"},
  {"status":  "/status"}]
}, {...}]
```

Fig. 3. A sample HTTP response sent back to the client. The packet contains the usual HTTP headers (including the HTTP verb or method: `GET`), as well as a JSON document as the body part.

`http://.../EnergieVisible/SmartMeters/`
with the `GET` method. The client gets back as a result the list of all the SmartMeters connected to the gateway. The selection of the suitable format for the client is achieved during a content negotiation phase (C2, C3), specified in HTTP. Thus, alongside with the `GET` request, the client sets the `Accept` field of the HTTP request to a weighted list of media types it can understand, for example to: `application/json;q=1`, `application/xml;q=0.5`. The server will try to serve the best possible format and will describe it in the `Content-Type` of the HTTP response.

Since the required format is a key parameter, we suggest supporting content negotiation directly in the URI as well in order to make it more natural for everyday users, directly testable and bookmarkable. Thus, our gateway supports requests such as
`http://.../EnergieVisible/SmartMeters.json`
as well. As a second step, the client selects the device it wants to interact with identified by a URI (C1):
`http://.../EnergieVisible/SmartMeters/RoomLamp.json`
By issuing a `GET` request on this resource it gets back its JSON representation as shown on Figure 3. In the response message of Figure 3 the client finds energy consumption data (e.g. current consumption, global consumption, etc.) as well as hyperlinks to related resources. Using these links the client can discover other related "services", fulfilling the constraint (C4) and enabling the discovery of resources.

As an example by contacting:
`http://.../RoomLamp/status`
with the standard OPTIONS method the client gets back the methods allowed on the status resource (e.g. `Allow: GET, HEAD, POST, PUT`). By sending the `PUT` method to this URI alongside with the payload `status=off`, the lamp is turned off.

The Web-enabling of the Ploggs allows to build fully Web-based energy monitoring applications, but also enables simple but very useful interactions such as bookmarking connected

appliances and being able to turn on/off or monitor them from any device with a Web browser.

B. Direct Access and Syndication of Wireless Sensor Networks

The Sun SPOT platform[2] is a wireless sensor network particularly suited for rapid prototyping of WSNs (Wireless Sensor Networks) applications. The RESTful architecture we designed and implemented [4] for the Sun SPOTs is composed of two main parts: an embedded Web server on each node, and a (reverse) proxy server to forward the HTTP requests from the Web to the SPOTs, that is from the IP network of the Web to the IEEE 802.15.4 network of the Sun Spots and vice-versa.

Each Sun SPOT has a few sensors (light, temperature, accelerometer, etc.), actuators (digital outputs, LEDs, etc.), and a number of internal components (radio, battery). The role of the embedded Web server is to make the sensors, actuators and internal components available as REST resources. Unlike for the Ploggs' implementation, we wanted the Sun SPOT nodes to directly provide a RESTful interface, without a Smart Gateway that translates REST requests to proprietary protocols, thus we implemented an embedded HTTP server directly on each sensor node, making it an independent and autonomous Web of Things device. As for the Ploggs, requests for services are formulated using URIs (C1). For instance, typing a URL such as

`http://.../spot1/sensors/light`

in a browser, requests the resource "light" of the resource "sensor" of "spot1" with the verb `GET` which illustrates that the natural structure of embedded devices maps quite well to resources.

The limited computing and storage capabilities of the nodes have two consequences. First they only serve a JSON representation of their resources. Secondly to avoid too large workload on the node we implemented a syndication mechanism for the sensors. As mentioned before, this also better fits the interaction model of sensor networks. Thus, the nodes can be controlled (e.g. turning LEDs on, enabling the digital outputs, etc.) using synchronous HTTP calls (client pull) but can also be monitored by subscribing to feeds (node push). More concretely, the subscription to a feed is done by creating new "rules" on sensor resources, e.g. by `POST`ing a threshold and the URI of an Atom(Pub) server to

`http://.../spot1/sensors/light/rules`

Every time the threshold is met, the sensor node pushes a JSON message to the given Atom server using AtomPub. This allows for thousands of clients to monitor a single sensor by *outsourcing* the processing onto an intermediate, powerful server.

C. Web-Enabled Everyday Things

The *Electronic Product Code (EPC)* Network [23] is a set of standards established by industrial key players towards a uniform platform for tracking and discovering RFID tagged objects and goods. This network offers, amongst other components, a standardized server-side EPCIS (EPC Information Service) which is in charge of managing and offering access to track and trace RFID events. Implementations of the EPCIS (such as Fosstrak [23]) offer a standard query and capture API through WS-* Web Services.

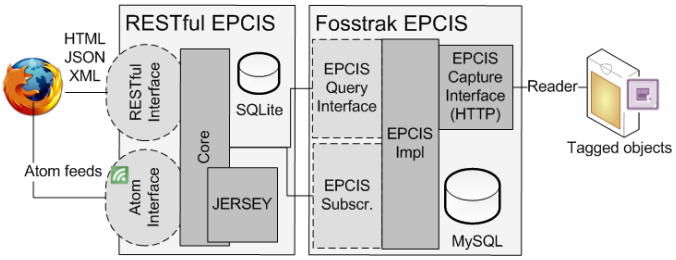

Fig. 4. Architecture of the RESTful EPCIS based on the Jersey RESTful framework and deployed on top of the Fosstrak EPCIS.

Ideally, in the Web of Things not only embedded devices, but also everyday things should be available. Thus, we decided to use the presented concepts to turn the EPCIS into a WoT Smart Gateway providing access to a global network of tagged everyday objects. This also helps to better grasp the benefits of a seamless Web integration (based on REST) versus using the Web as a transport (as done for WS-* Web Services).

As a client, the EPCIS offers three core features. First it offers an interface to query for RFID events. As mentioned before, this interface is accessible through a WS-* Web Service. While this enables to create clients using several languages supporting Web services, it makes it impossible to directly query for RFID events using Web languages such as JavaScript or HTML. More importantly it does not allow for exploring the EPCIS using a Web browser, searching for tagged objects or exchanging links pointing to traces of tagged objects. Thus, we implemented a RESTful translation of the EPCIS WS-* interface.

As shown on Figure 4, the RESTful EPCIS is a software module based on Jersey[3]. Jersey is a software framework for building RESTful applications. It is especially interesting since it complies with the JAX-RS (JSR 311) standard for building RESTful Web services. Clients of the RESTful EPCIS such as browsers or Web applications can query for tagged objects directly using REST and its uniform HTTP interface (C1, C2). Requests are then translated into WS-* calls on the standard EPCIS interface. This allows for the RESTful EPCIS to serve data provided by any implementation of the EPCIS standard. In our case we use Fosstrak[4], an open source implementation of the standard.

The first benefit of the RESTful EPCIS is that every RFID event, reader, tagged-object or location is turned into a Web resource and gets a globally resolvable URI which uniquely identifies it and can be used to retrieve various representations (C1). Thus EPCIS queries are transformed into compositions of these identifiers and can be directly executed in the browser (C5), sent by email or bookmarked. As an example, a factory

[2] `http://www.sunspotworld.com`

[3] `http://jersey.dev.java.net`

[4] `http://www.fosstrak.org`

manager who wants to know what tagged objects enter his factory can bookmark a URI like:
`http://.../epcis/rest/location/urn:company:factory1/reader/urn:company:entrance:1`
Furthermore these URIs are linked together (C4) in order to reflect the relationships of the physical world. This makes the RESTful EPCIS directly browsable (C4). Indeed, in addition to the XML representation of tagged objects offered by the standard it also provides (X)HTML and JSON representations (C3). With the HTML representation, end-users can literally browse tagged things and their traces simply by following hyperlinks (C4) as they would browse the Web of documents. For example, a location offers links to co-located RFID readers.

A standard EPCIS also offers an interface to subscribe to events. Through a WS-* operation, clients can send a query along with an endpoint (i.e. a URI) and subscribe for updates. Every time the result of the query changes, an XML packet containing the new results is sent to the endpoint. While this mechanism is practical, it requires for clients to run a server that listens to the endpoint and thus cannot be used by average users or cannot be directly integrated to a Web browser. To improve this, the RESTful EPCIS offers a RESTful subscription interface over HTTP (C2) and serves query updates through AtomPub as shown on Figure 4. This way end-users can formulate queries by browsing the hyperlinked EPCIS and obtain the updated results represented as Atom feeds which browsers can understand and directly subscribe to (C3). As an example a product manager could create a feed in order to be automatically notified in his browser whenever one of his product is ready to be shipped, he could then use the URI of the feed in order to send it to his most important customers for them to follow the goods' progress as well. A simple but very useful interaction which would require a dedicated client to be developed and installed by each customer in the case of the WS-* based EPCIS.

D. Physical Mashups

By implementing the suggested architecture for the Ploggs, the Sun SPOTs and the EPC Network, we enable the seamless integration of these physical things into the Web, and enable a new range of applications based on this unified view of the Web. We consider these applications as "physical mashups" where Web 2.0 technologies and patterns can be applied to easily build applications (i.e. a Web page making use of several other Web resources to create a new application). We describe two concrete prototypes of physical mashups.

1) Web Dashboards: In this first example we have created a mashup to answer an increasingly important need for households to understand their energy consumption and to be able to remotely monitor and control it.

The idea of the "Energie Visible"[5] project is to offer a Web dashboard that enables people to control and experiment with the energy consumption of their appliances. The dashboard is shown on the upper part of Figure 2 and offers six real-time and interactive graphs. The four graphs on the right side provide detailed information about the current consumption of all the appliances in the vicinity of the gateways.

Thanks to the Ploggs Web integration, the dashboard can be implemented using any Web scripting language. In this particular case it is built as a Google Web Toolkit (GWT)[6] application which is a robust platform for building Web mashups and offers a large number of easily customizable widgets. To dynamically draw the graphs according to the current energy consumption, the application only needs to issue an HTTP `GET` request to the gateway
`http://.../EnergieVisible/SmartMeters/all.json`
on a regular basis. It then feeds the resulting JSON document to the corresponding graphs widgets which can directly parse JSON.

The "Energie Visible" prototype (Web UI and Smart Gateways) was deployed at the headquarters of a private foundation working on sustainability[7] and has now been running for 8 consecutive months. The aim of the project was to help visitors and members to better understand how much each device consumes in operation and in standby. The Ploggs are used to monitor the energy consumption of various devices such as a fridge, a kettle, several printers, a file-server and computers and screens. A large display in the office enables people passing by to experiment with the energy consumption of the devices. The staff can also use the system by browsing to the Web UI on their desktop computer.

2) A Physical Mashup Editor: Tech-savvy users can create Web mashups using a "mashup-editor" such as Microsoft Popfly or Yahoo Pipes. These editors usually provide visual components representing Web sites and operations (add, filter) that the user only need to connect together to create a new application. We wanted to apply the same principles to allow users to create physical mashups without requiring any programming skills.

Our implementation is based on the Clickscript project[8]. A Firefox plugin written on top of the Dojo AJAX library and allowing people to create Web mashups by connecting resources (Web sites) and operations (greater than, if..then, loops, etc.) building blocks together. Since it is written in JavaScript, Clickscript cannot use resources based on WS-* Web Services or low-level proprietary service protocols. However it can easily access RESTful services available on the Web. Thus, it is very straightforward to create Clickscript building blocks (or widgets) based on Web of Things devices and similarly straightforward for non computer scientist users to create their own physical mashups. The mashup shown in Figure 5 gets the room temperature by `GET`ting the Sun SPOT temperature resource. If this is smaller than 36 degrees Celsius,

[5] A video of the project is available on `http://tiny.cc/xh5OC`

[6] `http://code.google.com/webtoolkit/`

[7] Cudrefin 02: `http://www.cudrefin02.ch/`

[8] `http://clickscript.ch`

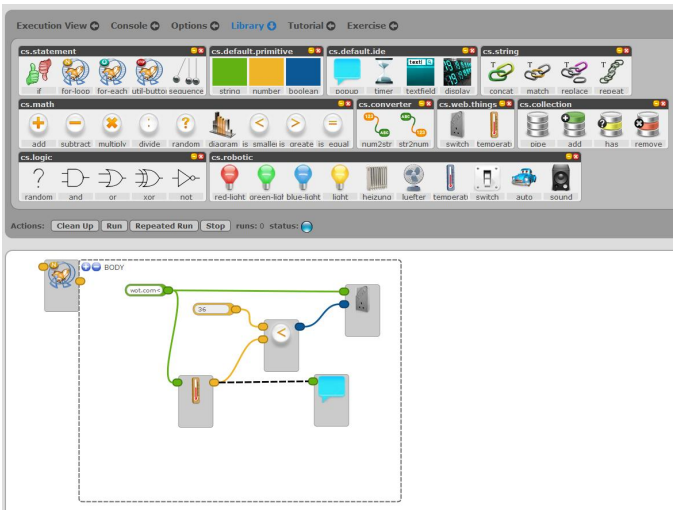

Fig. 5. Using the Clickscript Mashup Editor to create a physical mashup by connecting building blocks directly from a browser.

it `PUTs status=off` to a Plogg which turns off the fan it is connected to.

VI. Discussions and Future Work

Thanks to the loose-coupling, simplicity, and scalability of RESTful architectures, along with the wide availability of HTTP libraries and clients, RESTful architectures are becoming one of the most practical integration architecture. This makes it desirable to use Web standards for also interacting with smart things. Although HTTP introduces a communication overhead and increases average response latency, it is still sufficient for many pervasive scenarios where longer delays do not affect user experience [19], [14]. Previous work [9] has shown that the performance of using HTTP as a data exchange protocol is largely sufficient for common pervasive scenarios, especially when only a few concurrent users are accessing the same resource simultaneously (200 ms mean response time with 100 concurrent users on a 1.1 Ghz server running our Smart Gateway). We have also shown that caching techniques can significantly improve the performance of concurrent sensor data reading by using tools used for massively scalable Web sites [9]. These techniques can be directly applied to Web devices given that devices have on-board HTTP support [5].

Web 2.0 mashups have significantly lowered the entry barrier for the development of Web applications, which is now accessible to non-programmers. As demonstrated by the success of the Web, the development of a set of simple, reusable, and modular software components can greatly facilitate the integration of embedded devices of all kinds. It shall be noted that a resource-oriented approach should not be religiously considered as the miracle solution for every problem. In particular, scenarios with very specific requirements such as high performance real-time communication, might benefit from tightly coupled systems based on traditional RPC-based approaches. However, for less constrained applications where ad-hoc interaction and serendipitous re-use are necessary, Web standards allow any device to speak the same language as other resources on the Web. This makes much easier the integration of the real-world with any other Web content, so that physical things can be bookmarked, browsed, searched for, and used just like any other Web resource.

Based on our personal experience, the drawbacks of Web architectures are fairly compensated by the notable simplification of the application design, integration, and deployment processes [4], in particular when comparing RESTful devices with other systems for embedded devices, such as WS-* Web services. As an example the Plogg RESTful Gateway and the Sun SPOTs have been used by external developement teams who read about our project on our Web site. In the first case, the idea was to build a mobile energy monitoring application based on the iPhone and communicating with the Ploggs. In the second case, the goal was to demonstrate the use of a browser-based JavaScript Mashup editor with real-world services. According to interviews we conducted with these developers, their experience confirmed ours. They enjoyed using the RESTful smart things, in particular the ease of use of a Web "API" versus a custom "API". For the iPhone application a native API to Bluetooth did not exist at that time. However, like for almost any platform an HTTP (and JSON) library was available. One of the developer mentioned a learning curve for REST but emphasized the fact that it was still rather simple and that once it was learnt the same principles could be used to interact with a large number of services and possibly soon devices. They finally noted the direct integration to HTML and Web browser as one of the most prevalent benefits.

As mentioned earlier, HTTP was designed as a client-server architecture where clients pull data. This interaction model works fine for control-oriented applications, however, monitoring-oriented applications are often event-based and thus smart things should also be able to push data to clients (rather than being continuously polled). Using syndication protocols such as Atom and AtomPub improves the model when monitoring, since devices can publish asynchronously data using AtomPub on an intermediate server, nevertheless clients still have to pull data from Atom servers. Overcoming the client-server architecture is now a core research topic in the Web community [21]. Standards such as HTML5 are also going towards asynchronous bi-directional communication [22], therefore it is very relevant to further explore lightweight Web-based messaging systems.

Another major challenge for a global Web of Things is search and discovery of smart things. Consider billions of things connected to the Web either directly or indirectly (e.g. through Smart Gateways). Discovery by browsing HTML pages with hyperlinks becomes literally impossible in this case, hence the idea of searching for smart things. Searching for things is significantly more complicated than searching for documents, as things are tightly bound to contextual information such as location, and are often moving from one context to the other.

Beyond location, smart things also need to have means to

describe themselves in order to be (automatically) discovered: how to describe a thing on the Web so that both humans and machines can understand what services it provides. This problem is not inherent to smart things neither to the Web but more generally a problem in describing services. On the Web and for a Web of Things, languages such as Microformats[9] and RDFa will certainly help.

VII. Conclusion

In this paper, we suggested that Web technologies are — contrary to popular belief — a suitable protocol for building applications on top of services offered by smart things. After summarizing the core design principles of the modern Web architecture, we have proposed an architecture for the Web of Things based on the concepts of REST, syndication for smart things and Smart Gateways and demonstrated them with several prototypes.

Thanks to the loose-coupling, simplicity, and scalability of RESTful architectures and the wide availability of HTTP libraries and clients, RESTful architectures are becoming one of the most ubiquitous and lightweight integration architecture. Because of this, using the Web standards to interact with smart things seems to be adapted. Although HTTP introduces a communication overhead and increases average latency, it is sufficient for many pervasive scenarios where such longer delays do not affect user experience.

The advantages offered by introducing support for Web standards directly at the device-level are beneficial for developing a new generation of networked devices that are much simpler to program and reuse. Applying the same design principles that were detrimental to the success of the Web, in particular openness and simplicity, can significantly leverage the ubiquity and versatility of the Web as a common ground to network devices and applications. Furthermore, as most mobile devices have already Web connectivity and Web browsers, and most programming languages support HTTP, we tap in the huge Web developer community as potential application developers for the Web of Things.

Acknowledgements

The authors would like to thank Mathias Mueller and Patrik Fuhrer for their contribution on the RESTful EPCIS, to Thomas Pham for his work on the RESTful Sun SPOTs as well as to Lukas Naef for providing us with the Clickscript environment.

References

[1] E. Fleisch and F. Mattern, *Das Internet der Dinge*, 1st ed. Springer, Jul. 2005.

[2] J. W. Hui and D. E. Culler, "IP is dead, long live IP for wireless sensor networks," in *SenSys '08: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*. New York, NY, USA: ACM, 2008, pp. 15–28.

[3] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle, "The Web of Things: interconnecting devices with high usability and performance," in *6th International Conference on Embedded Software and Systems (ICESS'09)*, Hangzhou, Zhejiang, China, May 2009.

[4] D. Guinard, V. Trifa, T. Pham, and O. Liechti, "Towards Physical Mashups in the Web of Things," in *Proceedings of the 6th International Conference on Networked Sensing Systems (INSS 2009)*, 2009.

[5] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology*, vol. 2, no. 2, pp. 115–150, May 2002.

[6] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly, 2007.

[7] C. Pautasso and E. Wilde, "Why is the web loosely coupled? a multi-faceted metric for service design," in *Proceedings of the 18th International World Wide Web Conference*, 2009, pp. 911–920.

[8] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, B. Serra, and M. Spasojevic, "People, places, things: web presence for the real world," *Mob. Netw. Appl.*, vol. 7, no. 5, pp. 365–376, 2002.

[9] 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 *Proceedings of the 2nd International Workshop on Sensor Network Engineering (IWSNE'09)*, Marina del Rey, CA, USA, June 2009.

[10] W. Roy, F. K. P., G. Anuj, and H. B. L., "Bridging physical and virtual worlds with electronic tags," in *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM, 1999, pp. 370–377.

[11] P. Ljungstrand, J. Redström, and L. E. Holmquist, "Webstickers: using physical tokens to access, manage and share bookmarks to the web," in *DARE '00: Proceedings of DARE 2000 on Designing augmented reality environments*. New York, NY, USA: ACM, 2000, pp. 23–31.

[12] P. Schramm, E. Naroska, P. Resch, J. Platte, and H. Linde, "Integration of limited servers into pervasive computing environments using dynamic gateway services," Computer Engineering Institute, University Dortmund, Germany, Tech. Rep. 0202, 2002.

[13] B. Traversat, M. Abdelaziz, D. Doolin, M. Duigou, J. Hugly, and E. Pouyoul, "Project JXTA-C: Enabling a Web of Things," in *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, 2003, pp. 282–290.

[14] N. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao, "Tiny web services: design and implementation of interoperable and evolvable sensor networks," in *SenSys '08: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*. New York, NY, USA: ACM, 2008, pp. 253–266.

[15] M. L., S. P., G. D., K. M., K. S., and S. D., "Socrades: A web service-based shop floor integration infrastructure," in *Internet of Things 2008, First International Conference for Academia and Industry*, 2008.

[16] A. Kansal, S. Nath, J. Liu, and F. Zhao, "SenseWeb: an infrastructure for shared sensing," *IEEE Multimedia*, vol. 14, no. 4, pp. 8–13, 2007.

[17] V. Stirbu, "Towards a RESTful Plug and Play Experience in the Web of Things," in *IEEE International Conference on Semantic Computing*, Aug. 2008, pp. 512–517.

[18] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim, "TinyREST - a protocol for integrating sensor networks into the internet," *in Proc. of REALWSN*, 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.112.5129

[19] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin, "pREST: a REST-based protocol for pervasive systems," in *Proc. of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, Oct. 2004, pp. 340–348.

[20] E. Wilde, "Feeds as query result serializations," School of Information, UC Berkeley, Berkeley, California, Tech. Rep. 2009-030, April 2009.

[21] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle, "Consistency and scalability in event notification for embedded web applications," in *11th IEEE International Symposium on Web Systems Evolution (WSE'09)*, Edmonton, Canada, September 2009.

[22] I. Hickson, "Server-sent events," World Wide Web Consortium, Working Draft WD-eventsource-20090423, April 2009.

[23] C. Floerkemeier, M. Lampe, and C. Roduner, "Facilitating RFID development with the accada prototyping platform," in *Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops*. IEEE Computer Society, 2007, pp. 495–500.

[9] http://www.microformats.org