Towards the Web of Things: Web Mashups for Embedded Devices

Dominique Guinard
Institute for Pervasive Computing
ETH Zurich, Switzerland
and SAP Research Zurich
dguinard@inf.ethz.ch

Vlad Trifa
Institute for Pervasive Computing
ETH Zurich, Switzerland
and SAP Research Zurich
vlad.trifa@ieee.org

ABSTRACT

In the "Internet of Things" vision, the physical world becomes integrable with computer networks. Embedded computers or visual markers on everyday objects allow things and information about them to be accessible in the digital world. However, this integration is based on competing standards and requires custom solutions, thus requires extensive time and technical expertise. Based on the success of Web 2.0 mashup applications, we propose a similar approach for integrating real-world devices to the Web, allowing for them to be easily combined with other virtual and physical resources. In this paper we discuss possible integration method, in particular how the REST principles can be applied to embedded devices. Then we illustrate these principles with two concrete implementations: on the Sun SPOT platform and on the Ploggs wireless energy monitors. Finally, we show how RESTful interactions can be leveraged to quickly create new prototypes and mashups that combine the physical and virtual world.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous

Keywords

Web of Things, REST, embedded devices, real-world mashups, Web, Internet of Things

1. INTRODUCTION

In the last decade, a tremendous progress in the field of embedded systems has given birth to a myriad of tiny computers, where virtually any type of sensors/actuators can be attached. By inter-connecting these devices using low-power wireless communication, a brand new world of possible applications is unveiled. Networks of physically distributed computers, usually called wireless sensor networks (WSN), would be invaluable tools for monitoring the physical world. Unfortunately, due to the lack of standards most projects in this field are based on different - and usually incompatible - software and hardware platforms. Within such an heterogeneous ecosystem of devices, development of simple applications still requires extensive skills and time. Moreover, for each new deployment a large amount of work must be devoted to reimplement basic functions and application specific user interfaces, which is a waste of resources that could be used by developers to focus on the application logic. Ideally, developers should be able to quickly build applications only by recombining ready-made building blocks, just like with LEGO bricks.

In spite of the increasing popularity of open source communities, progress in networked objects is still being limited by the lack of clear, standardized, and interoperable communication protocols. For the realm of the "Internet of Things" to materialize (and be scalable), there is an unmet need for a common language that can be understood by my fridge, your TV set, and her car. The Internet is a stunning example of a global network of computers that interoperate smoothly together in spite of the large amount of different software and hardware platforms available, and there is a growing number of embedded devices that can connect directly to the Internet. Based on these observations, we propose to leverage the existing and ubiquitous Web protocols as common ground where real objects could interact with each other. One of the advantages of using Web standards is that devices will be able to finally "speak" the same language as other resources on the Internet, therefore making it very easy to integrate physical devices with any content on the Web. The mashup paradigm has been successfully applied to fast prototype valuable applications, however, a similar model is missing for physical computing.

Our contribution in this article is to propose two ways to integrate real-world things into the existing Web by turning real objects into RESTful resources that can be used directly over HTTP. First, we describe how an actual Web server can be implemented on tiny embedded devices to turn them into RESTful resources. Second, when computational resources are too limited or devices do not offer a RESTful interface, we propose to use an intermediate gateway that can offer a unified REST API to access these devices, by hiding the actual communication protocols used to interact with them. Finally, we illustrate our approach with real prototypes we have built on top of this ecosystem of RESTful devices.

Our main aim is to lay the basis of the future Web of Things. By providing practical guidelines on how to blend real-world devices into the existing Web, devices and their properties become browsable with any Web browser, with no need to install any additional software or driver. Moreover, simple mashups that combine real-time data from physical devices and other Web content can be built with much less effort than required by existing approaches. Just like mashups have significantly contributed to the "democratization" of the Web, we hope that physical mashups will drastically lower the entry barrier for developing home-made applications with devices, thus accelerating the acceptance of the Web of Things.

2. RELATED WORK

With advances in computing technology, tiny Web servers can be embedded in most embedded devices [4]. The idea of each thing having its own Web page is appealing because Web pages could be indexed by search engines, then searched and accessed directly from a Web browser. In the Cooltown project [6], each thing, place, and person have an associated Web page with information about them. JXTA is an open network computing platform designed for peer-to-peer computing that can be implemented on all kinds of devices [15], but is more an overlay network which is not related to the Web in particular. More recently, Web services have also been used to interconnect devices on top of standard Web protocols [11].

The SenseWeb project [5] is a platform for people to share their sensor readings using Web services to transmit the data to a central server. Pachube[1] offers a similar community Web site for people to share their sensors and uses more open data formats. These approaches are based on a centralized repository and devices need to be registered before they can publish data, thus are not sufficiently scalable and are more concerned with data storage and retrieval. Prehofer et al. [10] recently proposed a Web-based middleware that is similar to our approach, however Internet is used only as a transport protocol.

In most early Web-based approaches, HTTP is used only to transport data between devices, while in fact HTTP is an application protocol. Projects that specifically focus on reusing the founding principles of the Web as an application protocol are still not common. Creation of devices that are Web-enabled *by design*, would facilitate the integration of physical devices with other content on the Web. As pointed out in [17], REST-enabled devices would not require any additional API or descriptions of resources/functions. An early gateway system similar to ours has been proposed in [13], but was very limited in terms of functionality. The approach found in [14] is the first to our knowledge to take a very similar approach to ours, but mainly focuses on the discovery of devices rather than the functionalities offered by these real-world devices.

3. WEB-ORIENTED 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 merely as a transport protocol – as done when using WS-* Web services –, we would like to make devices an integral part of the Web by using HTTP as application layer. For this purpose, we make the functionalities of real-world embedded devices available through a RESTful API over HTTP, as described in Section 3.1. In this paper we propose two alternative methods to enable REST based interaction with embedded devices. First, devices are directly made part of the Web, by implementing a web server on them directly (Section 3.2). In the second, devices are connected through a Smart Gateway which translates requests across protocols (Section 3.3).

[1]`http://www.pachube.com`

3.1 Sensor Nodes as RESTful Resources

The architectural principle that lies at the heart of the Web, namely Representational State Transfer (REST) as defined by Roy Fielding [2], shares a similar goal with more well known integration techniques such as WS-* Web services (SOAP, WSDL, etc), which is to increase interoperability for a looser coupling between the parts of distributed applications. However, the goal of REST is to achieve this in a more lightweight and simpler manner, and focuses on resources, and not functions as is the case with WS -* Web services. In particular, REST uses the Web as an application platform and fully leverages all the features inherent to HTTP such as authentication, authorization, encryption, compression, and caching. This way, REST brings services "into the browser": resources can be linked and bookmarked and the results are visible with any Web browser, with no need to generate complex source code out of WSDL files to be able to interact with the service.

To achieve this, REST proposes two basic rules[2]:

1. The application model is transformed from operation-centric into a data-centric one. This means "everything" that offers services becomes a resource (e.g. a temperature sensor is a resource of a the sensor node resource) that can be identified unambiguously using URIs.

2. The four main operations provided by HTTP (GET, POST, PUT, DELETE) are the only available operations on resources, they define a uniform interface with well-known and shared semantics.

The simplicity of REST and its seamless integration into global networks makes it an ideal candidate for creating "tactical, ad-hoc integration over the Web" [9]. These advantages mainly explain why REST services are the technological basis for an increasing number of Web 2.0 services as those offered by Flickr, Twitter, Facebook, Del.icio.us, Google and Amazon. Traditionally, REST has been used to integrate websites together. However, the lightweight aspect of REST makes it an ideal candidate for resource-constrained embedded devices to offer services to the world [7, 8]. Since many such devices usually offer rather simple and atomic functionalities (for example reading sensor values), modeling them using REST is often straightforward.

3.2 Integration Through Direct API Access

Although REST seems suited for embedded devices, these do not always have an IP (Internet Protocol) address and are thus not directly addressable on the Internet. However, it is very likely that more and more real-world devices will become IP-enabled and have embedded HTTP servers (in particular with 6LowPAN), making them able to understand the Web languages and protocols [4, 1]. Such Web-enabled devices can be directly integrated and make their RESTful APIs directly accessible on the Web. This integration process is shown on Figure 1. Each device has an IP address and runs a Web Server on top of which it offers a RESTful API to the mashup developer.

[2]A comprehensive description of the REST principles and on the creation of Resource Oriented Architectures is provided in [12]

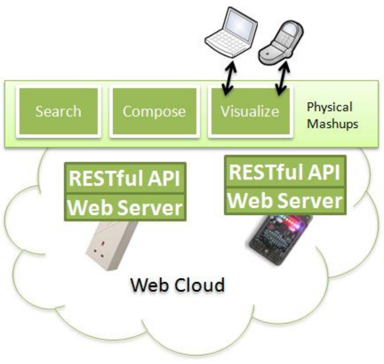


Figure 1: Direct integration of IP real-world devices. Each device embeds a Web Server and offers its functionality through a RESTful API.

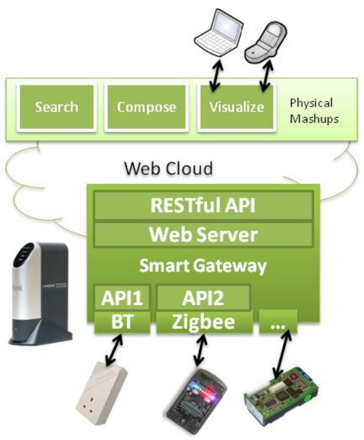


Figure 2: Integration architecture with Smart Gateways making the functionalities of real-world devices available on the Web through a RESTful API.

3.3 Integration Through API Access on Smart Gateways

While such Web-enabled devices are likely to be widely spread in the near future, direct integration of real-world devices into the Web is still a rather cumbersome task. In particular, when devices do not support IP or HTTP as is usually the case with wireless sensor networks (WSN), a different integration pattern is needed. As shown on Figure 2, we propose to use the concept of *Smart Gateways* as intermediate element that bridges the Web with devices that do not talk IP [16]. Smart Gateways have one main goal: they abstract the proprietary communication protocols or APIs of embedded devices and offer their functionalities accessible via a RESTful API. Each gateway has an IP address and runs a Web server, and understands the proprietary protocols of different devices that are connected to it through the use of dedicated drivers. 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 in the proprietary API of the node and transmits it using the communication protocol the sensor node understands (e.g.

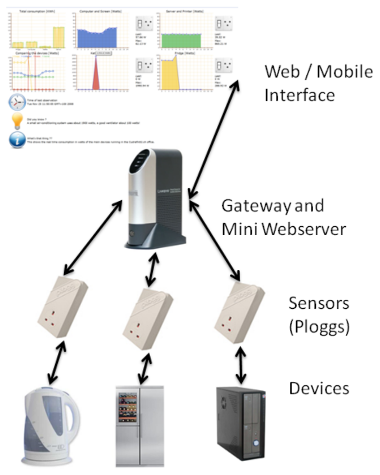


Figure 3: Devices extended attached to the Ploggs power outlets communicating with a Smart Gateway offering the Ploggs functionalities as RESTFul services.

Zigbee[3]). A Smart Gateway can support several types of devices through a driver architecture as shown on Figure 2 where the gateway supports three types of devices and their corresponding communication protocols. Technical details of the Smart Gateways can be found in [16]. Ideally, a Smart Gateway needs to keep a small footprint so that it can be integrated to computers already present in the environment such as Network Attached Storage (NAS) devices or Wireless routers. As an example we successfully tested the deployment of our gateway on an NSLU2 NAS[4].

Aside from connecting limited devices to the Web, a Smart Gateway can also add more functionalities to devices. Gateways can be used for orchestrating the composition of several low-level services into higher-level services available from the RESTful API, that is creation of mashups using device-level services. For instance, if an embedded device offers monitoring of the energy consumption of appliances, the Smart Gateway could provide a service that returns the sum of all the energy consumption monitored by all the embedded devices connected to the gateway.

4. IMPLEMENTATION

In order to empirically analyze and test the potential of the RESTful approach for real-world services and how our approach could become the basis for the Web of Things, we implemented the architecture on two WSNs plateforms: the Sun SPOT sensor nodes[5] and the Ploggs Energy Sensors[6]. In this section we describe the architecture of both implementations and then focus on how these were used to create mashups.

4.1 RESTful Ploggs

In our first implementation, we illustrate the usage of a Smart Gateway (see Section 3.3). For this purpose, we use

[3]`http://www.zigbee.org/`
[4]`http://en.wikipedia.org/wiki/NSLU2`
[5]`http://www.sunspotworld.com/`
[6]`http://www.plogginternational.com/`

intelligent power outlets called Plogg which can measure the electricity consumption of the devices that are plugged into them. Each Plogg is also a wireless sensor node that communicates over Bluetooth. This makes the Ploggs especially suited for energy monitoring at the appliance level. However, the integration interface offered by the Ploggs is proprietary which makes the development of the applications on top of the Ploggs rather tedious.

The Web-oriented architecture we have implemented using the Ploggs is based on four main layers as shown in Figure 3. The first layer is composed of appliances we want to monitor 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. The Smart Gateway embeds a lightweight Web server which offers the monitoring and control functionalities of the Ploggs through URL, in a RESTful manner. The last layer is the user interface where the mashup actually occurs and is described in Section 4.3.1.

4.1.1 Ploggs Smart Gateway

The Ploggs Smart Gateway is a component written in C++ 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. It then filters the identified devices according to their Bluetooth identifier. The next step is to make their functionalities available though simple URLs, and for that a small footprint Web server is used to enable access to the sensors' functionalities over the Web. This is done by mapping URLs to native requests on the Plogg Bluetooth API. For instance,
`http://webofthings.com/energymonitor/ploggs/kitchen`
is automatically bound by the Gateway to a method that runs a low-level call that first initiates a bluetooth connection, and then connects to the Plogg named "Kitchen", and polls the Plogg for reading the current load of energy measured. For URLs to be served on the Web, the Gateway embeds a small footprint Web server. After evaluating several options[7], we decided to use Mongoose, a 35 Kb cross-platform Web server[8].

Beyond discovering the Ploggs and mapping their functionalities to URLs available on the Web, the gateway has two other main features. First, it can offer *local* mashups or aggregates of device-level services. For example, the Ploggs' Smart Gateway offers a service that returns the combined electricity load of all the Ploggs found at any given time. The second feature is that the gateway can represent the output of services on resources in various formats. As is often the case in Resource Oriented Architectures [12], an (X)HTML page is returned by default to ensure the browsability of the results. Results could also be returned in a more interoperable format called JSON (JavaScript Object Notation)[9]. JSON is an alternative to XML often used as a data exchange format for Web mashups. Since JSON is a lightweight format we believe it more adapted to devices with limited capabilities. As an example, the monitoring data of all the Ploggs currently available can be retrieved by accessing the following URL:
`http://webofthings.com/energymonitor/ploggs/*.json`

[7]`http://tinyurl.com/compareWS`
[8]`http://code.google.com/p/mongoose`
[9]`http://json.org/`

```
[{
  "deviceName":  "ComputerAndScreen",
  "currentWatts":  50.52,
  "KWh":  5.835,
  "maxWattage":  100.56
},
  "deviceName":  "Fridge",
  "currentWatts":  86.28.,
  "KWh":  4.421,
  "maxWattage":  288.92
}, {...}]
```

Figure 4: 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. For simplicity, only the JSON part is shown.

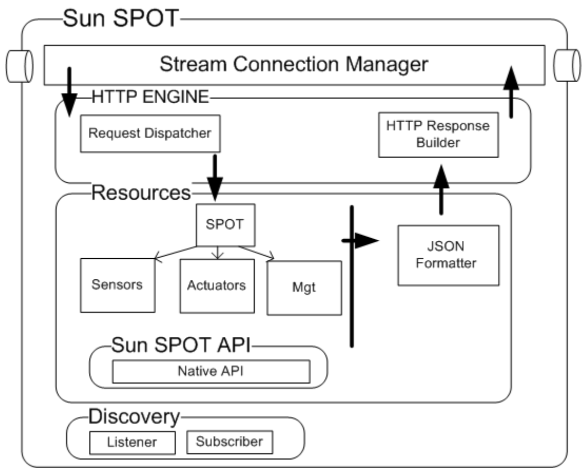


Figure 5: Architecture deployed on the Sun SPOTs.

As a result, the gateway calls all the Ploggs and wraps the results in the form of a JSON document, and is shown in Figure 4.1.1.

4.2 RESTful Sun SPOTs

The Sun SPOT plateform is a wireless sensor node particularly suitable for rapid prototyping of WSNs applications. Sun SPOTs run a small footprint Java Virtual Machine that enables the nodes to be programmed using the high-level Java programming language (Java Micro Edition CLDC[10]). The RESTful architecture we designed and implemented for the Sun SPOTs is composed of two main parts: a software stack embedded on each node, and a proxy server to forward the HTTP requests from the Web to the SPOTs.

4.2.1 Embedded Stack

Each Sun SPOT offers a number of sensors (light, temperature, accelerometer, etc.), a number of actuators (digital outputs, LEDs, etc.) and a number of internal components (radio, battery). The goal of the embedded web server is to make both the sensors and actuators available as REST resources. Unlike for the Ploggs' implementation, we

[10]`http://java.sun.com/javame/`

```
GET /spot1/sensors/light HTTP/1.1
Host: localhost:8080
[...]
HTTP/1.x 200 OK
Server: Noelios-Restlet-Engine/1.0..11
Content-Type: text/plain; charset=ISO-8859-1
device: spot1
resource: /sensors/light
method: GET
Gateway-Location: Office B 7.1.60
{"values":
[       {"lightlevel":[80]},
        {"threshold":[-1,37]}
]}
```

Figure 6: A sample HTTP request and response exchanged between a client and SPOT. The packet contains the usual HTTP headers (including the HTTP verb or method: GET) as well as a JSON document a body part.

wanted the Sun SPOT nodes to provide directly a RESTful interface (see Section 3.2), without a Smart Gateway that translates REST requests to proprietary protocols. We implemented an embedded HTTP server directly on the Sun SPOTs nodes (nanohttpd). The server natively supports the four main operations of the HTTP protocol GET, POST, PUT, DELETE, i.e the main *verbs* of REST. The HTTP server is deployed on each sensor node, making it an independent and autonomous Web device.

As for the Ploggs, requests for services (i.e. verbs on resources) are formulated using a standard URL. For instance, typing a URL such as
`http://webofthings.com/spot1/sensors/light`
in a browser, requests the resource "light" of the resource "sensor" of "spot1" with the verb GET. When the embedded webserver gets such a request, it will dispatch it to the corresponding resource handler as shown on Figure 5. The resource then reads the current light level using the native SunSPOT API and sends it to a formatter component. This component formats the results using JSON, and wraps it into an HTTP packet that is sent back to the client. An extract of the resulting HTTP packet is shown on Figure 6. Alternatively, our implementation allows the results to be transmitted asynchronously to a URL when the values reach a certain threshold, which is configurable through the REST API as well.

4.2.2 Proxy Server

Since Sun SPOTs do not yet support the IP (Internet Protocol) stack, we were not able to integrate them completely into the Web. Their radio communication is based on the IEEE 802.15.4 standard. The Web is not directly linked to this protocol, thus a proxy that bridges the Web requests (from TCP/IP) and to the devices over the IEEE 802.15.4 link is needed.

Furthermore, to allow mobile mashups, we wanted the nodes to be able to travel from one place to the other, which requires a dynamic discovery process to find new nodes and register their basic information (their MAC address and/or URL, a short description, etc). This process is carried out by a discovery component, which broadcasts invitation messages on a regular basis on a dedicated port. On their side, the nodes listen to this port and can decide to subscribe to the broadcasting proxy server. Then, the proxy registers the node's address and when receiving an HTTP request from the Internet, it reads the request URL and maps it to one of the registered nodes. In case the node is busy, it also serves as a buffer by queuing requests and resubmitting them later. In order to deal with URL and HTTP, the proxy uses functionalities developed on top of RESTlet, a lightweight REST framework implemented in Java[11]. We expect 6LowPAN implementations to be available for the SunSPOT soon, thus a proxy will not be needed anymore for connecting the SPOTs to the Internet.

4.3 Mashups in the Web of Things

Based on this substrate of real-world devices that offer a RESTful API, we can now easily integrate the functionalities of both the Sun SPOTs and the Ploggs to easily create new composite applications. We provide in this section two concrete example of mashups that can be created on top of these embedded devices. We classify these examples in two categories: *physical-virtual mashups* (also called cyber-physical systems) and *physical-physical mashups*. In the first categories, we present two prototypes of user interfaces running on a computer and consuming services from the real-world. In the second category, we present a prototype of a physical user interface, or ambient user interface that uses services from the real-world.

4.3.1 Physical-Virtual Mashups

In the first example of prototype implementation, we built an AJAX management interface on top of the Sun SPOT RESTful API. In the second example, we create a mashup web UI that can be used to monitor the energy consumption of household appliances.

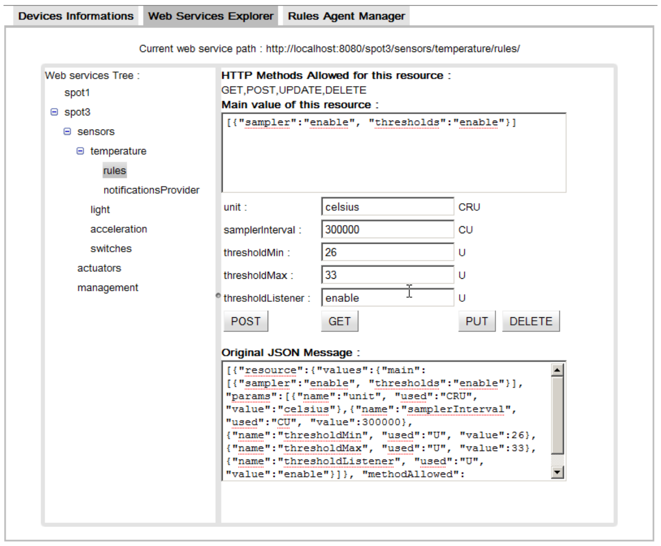
4.3.1.1 Sun SPOT Resources Manager.

While a minimal presentation requirement for real-world devices in the Web of Things is to offer a (X)HTML interface to browse their resources, it might not always be sufficient. Indeed, the sole HTTP verbs that can easily be used from HTML pages are GET and POST. Furthermore, using the plain HTML interface each page can be browsed to explore the Spots' resources results in initiating a new communication link with the involved Spots. Since these initializations are rather expensive in terms of battery life, it would be desirable to reduce them as much as possible by having the Spot communicating more data at once and by caching this data on the client side (or on the proxy).

To overcome both these limitations (limited verbs and expensive communication) and illustrate a solution, we built an AJAX (Asynchronous Javascript and XML) interface on top of the RESTful Sun SPOT API, shown on Figure 7. AJAX Web pages present two main advantages in this case. First, they can initiate HTTP calls with any of the HTTP verbs (e.g. with PUT and DELETE). Second, these calls can be executed asynchronously and the results can be displayed only when needed, thus offering a straightforward way of reducing communication with the device. Note that while such an interface is not a mashup *per se*, it greatly helps building mashups on top of the Sun SPOTs as it en-

[11]`http://www.restlet.org/`

ables mashup developers to test all the functionalities provided by the sensors from their Web browser. The *Resource Manager* interface offers to mashup builders a set of tools to experiment and configure the Sun SPOT RESTful API. Using the left-side of the UI the resource hierarchy is represented as a tree reflecting the hierarchy of the physical world (e.g. a temperature sensor is the "son" of a sensor node). The right-side of the UI allows for testing and configuring the functionalities of the selected resource using all the available HTTP verbs. As an example, on the left-side of Figure 7 the mashup builder can create a new rule to be applied to the temperature sensor of Spot3. Whenever this rule is trigerred, the Spot will POST the rule result to the specified URI. The user can create a rule by filling the provided HTML form and clicking on the HTTP verb he wants to use, POST in this case. These forms are dynamically generated based on the JSON message received from the device.



Current web service path : http://localhost:8080/spot3/sensors/temperature/rules/

HTTP Methods Allowed for this resource :
GET,POST,UPDATE,DELETE

Main value of this resource :

```
[{"sampler":"enable", "thresholds":"enable"}]
```

unit :	celsius	CRU
samplerInterval :	300000	CU
thresholdMin :	26	U
thresholdMax :	33	U
thresholdListener :	enable	U

Original JSON Message :

```
[{"resource":{"values":{"main":
[{"sampler":"enable", "thresholds":"enable"}],
"params":[{"name":"unit", "used":"CRU",
"value":"celsius"},{"name":"samplerInterval",
"used":"CU", "value":300000},
{"name":"thresholdMin", "used":"U", "value":26},
{"name":"thresholdMax", "used":"U", "value":33},
{"name":"thresholdListener", "used":"U",
"value":"enable"}]}, "methodAllowed":
```

Figure 7: Using the AJAX Resource Explorer, users can explore directly the resources provided by the device. This example shows how the UI can be used to create new rules on the temperature sensor.

4.3.1.2 *Energy Visualizer.*

In this second example we create a mashup that fullfills an increasingly important need for households. Indeed, a major burden for people who want to save energy is to identify how much energy is consumed by home appliances. *"How much does my computer consume in operation / when it is powered off or in standby? Is the consumption of my energy-saving lamp significantly lower in the long run than the normal lamp I've got there?"* Such questions are key to understand where energy can be saved with simple efforts. Currently available solutions, such as traditional LCD power monitors, are helpful but do not really fit the needs of most individuals. They lack the ability to compare consumption of individual devices into a single place, on a screen or a mobile phone in an appealing and simple manner. Furthermore, they do not offer many options for remote monitoring and control.

The idea of the Energy Visualizer prototype we have built is to offer a dashboard user interface on the Web that enables people to control and experiment with the consumption of their appliances. We wanted the user interface to be attractive, easily accessible (no additional software to learn or install) and to display real-time data about the energy consumption rather than snapshots, thus decided to use a dynamic Web page illustrated in Figure 8. It 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 currently in the vicinity of the gateways. The two remaining graphs show the total consumption (kWh), and respectively a comparison (on the same scale) of all the running appliances. Finally, a switch button next to the graphs enables the user to power on and off the devices over the Web.

This dashboard is built as a mashup that uses the RESTful Plogg API in a Google Web Toolkit application[12]. The Google Web Toolkit (GWT) is a great platform for building web mashups since it offers a large number of easily customizable widgets. For the graphs shown on Figure 8, we use the Open Flash Chart GWT Widget Library. This library offers a comprehensive set of graph widgets that are customizable by feeding them JSON documents.

To dynamically draw the graphs according to the current energy consumption, the mashup application calls the Ploggs Smart Gateway every 10 seconds by issuing a GET HTTP request to all the Ploggs
`http://webofthings.com/energymonitor/ploggs/*.json`
or by requesting the energy consumption of a single Plogg with a GET call to
`http://webofthings.com/energymonitor/ploggs/fridge.json`
It then feeds the resulting JSON document (shown on Figure 4) to the corresponding graphs. Furthermore, a click on the switch button next to the graphs on Figure 8 can stop the corresponding appliance by sending the following HTTP packet shown on Figure 9. The effect of this call will be to stop delivering power to the device attached to the TV plogg, i.e. in this case it will turn the TV off.

```
PUT /energymonitor/ploggs/tv HTTP/1.1
Host: webofthings.com
status: off
```

Figure 9: An HTTP request using the RESTful Plogg API to turn a TV off.

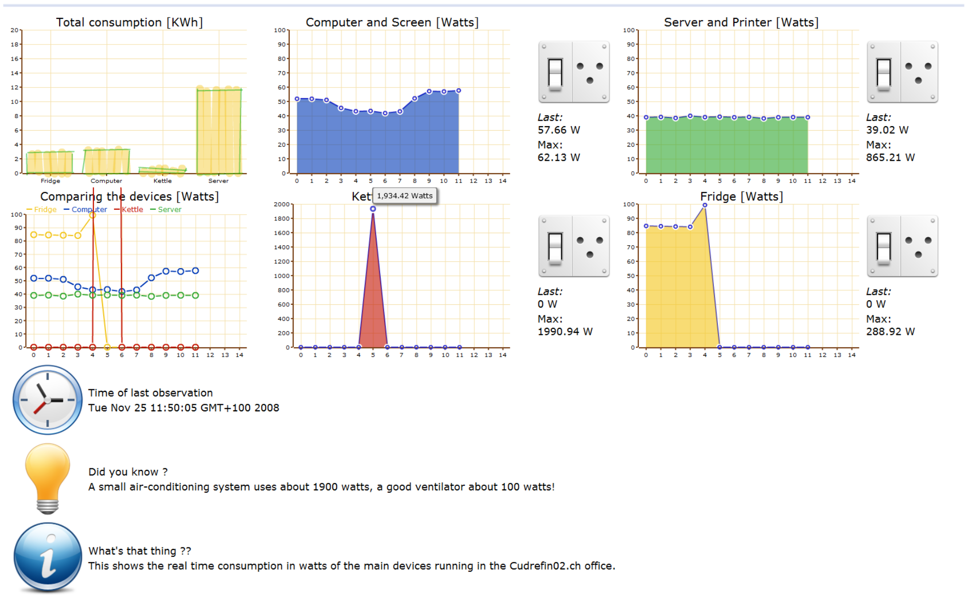
4.3.2 *Physical-Physical Mashups*

This last prototype demonstrates how real-world services provided by physical devices can be combined together using the underlying technologies of mashups, without even requiring a computer or HTTP browser.

4.3.2.1 *Ambient Meter.*

The prototype is an ambient device that displays the level of energy consumption of the place it is currently located in by changing its color. It can be taken from one place to the other and adapts to the place it monitors automatically, without the need for human intervention. Depending on the total amount of energy consumed in the room it is located in, the Ambient Meter changes its color from very green (i.e.

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



Time of last observation
Tue Nov 25 11:50:05 GMT+100 2008

Did you know ?
A small air-conditioning system uses about 1900 watts, a good ventilator about 100 watts!

What's that thing ??
This shows the real time consumption in watts of the main devices running in the Cudrefin02.ch office.

Figure 8: The monitoring and control web user interface for the Ploggs.

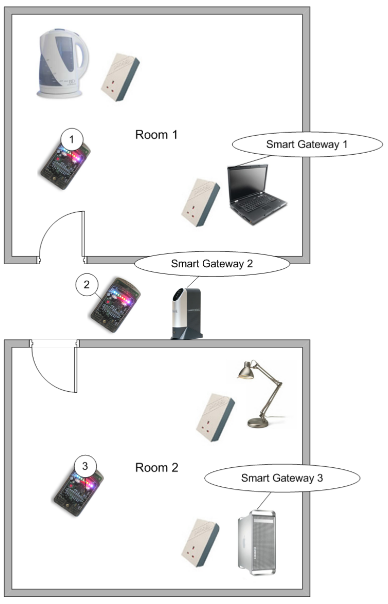


Figure 10: Demonstration settings of the Ambient Meter. Every 20 seconds the Ambient Meter (implemented as a Sun SPOT) polls the URL for an assessment of the energy consumption observed by all the Ploggs the Smart Gateway discovered.

the amount of energy consumed in the room is low) to very red (i.e. a lot of energy is currently consumed in this place).

The Ambient Meter is implemented on a Sun SPOT which uses the Ploggs for energy monitoring as well as the Ploggs Smart Gateway for resolution of its current location. It uses an HTTP connector we implemented in the RESTful Sun SPOT API to contact the RESTful Plogg API. Every 5 seconds, the Ambient Meter will poll the following URL using the GET method on
`http://localhost/energymonitor/load.json`
When the meter is located in Room 1, as shown on Figure 10 (step 1) it is bound to the Smart Gateway 1, meaning that localhost in this context is bound to the address of Smart Gateway 1. Thus, the result of the call is going to be the JSON representation of the energy consumption of all the Ploggs discovered by the Ploggs' Smart Gateway 1. Placed in the hallway, the Ambient Meter binds itself to Smart Gateway 2. Using the same URL as before it will get the energy consumption of all the devices monitored. Again, the same process occurs in Room 2, where the Ambient Meter gets the load of the desktop computer and the lamp. Integrating all the real-world devices of this prototype would have been rather time consuming if the Smart Gateways, the Ploggs and the Sun SPOTs were only offering their native (proprietary) APIs. Thanks to the RESTful approach the integration work was reduced to building a simple Web mashup, where all the services are invoked by means of simple and lightweight HTTP requests.

5. DISCUSSION AND FUTURE WORK

In this paper we have contributed to a step towards the realization of the Web of Things. By creating RESTful APIs to integrate the services offered by devices and objects in the real world such as wireless sensor networks, embedded devices and household appliances with any other Web content. We have described two ways to integrate devices to the Web using REST: direct integration based on the advances in embedded computing, and a Smart Gateway-based approach for resource-limited devices. We have further illustrated these methodologies by implementing them on two different platforms. Finally, we show how an eco-system of RESTful devices can facilitate significantly the creation of cyber-physical mashups. In the meanwhile, using REST and the Web to connect devices offers a very flexible and powerful mechanism to fast prototype all kinds of applications.

Still, it is important to note that REST services also have certain limitations and do not always solve problems in a straightforward manner. For instance, the inherent simplicity of REST paradoxically complicates the creation of complex services. While REST services are well adapted for simple and atomic services, which cover the greatest part of services available on embedded computers, their limitations become evident when it comes to modeling services which require complex input and/or deliver complex outputs. Based on our own experience and the experience of others [9] in more traditional integration patterns such as WS-* Web Services, we suggest that WS-* services are to be preferred for complex real-world integration and rather static use-cases, such as those involving complicated business processes or requiring high reliability, for example composing a manufacturing process on several machines. However, for smaller and more user-oriented applications, the RESTful approach offers many advantages such as light and simple use, browsability of services, and a much looser coupling. In our opinion, this illustrates very well the type of applications mashups are suited to.

Providing a substrate of RESTful and Web oriented embedded devices is only a step towards a global Web of Things. While we believe it greatly simplifies the development of adhoc applications, the RESTful approach also introduces new challenges, such as dealing with the vast variety of data formats that HTTP payloads can contain (e.g. XML, JSON, raw ASCII, etc.). Furthermore, as more and more devices will become part of the Web new challenges will appear. Some of these, directly related to mashups, are shown on the upper parts of Figures 1 and 2. How will we be able to search amongst an increasing number of dynamic devices, and in particular how will we be able to identify the device we want to interact with? This point has been identified as a particularly important problem for future ubiquitous environments [3]. Even if real-world devices are to offer Web Servers and Web pages, searching for them is not entirely similar to searching textual information on the Web. In particular, context information such as location, time, type of use is central for searching real-world services. Thus, a scalable dynamic search mechanism that takes into account the physicality of real-world services will be necessary. Similarly, building mashups consuming services on the Web is not entirely similar to building mashups consuming real-world services. If we really want end-users to be able to build mashups we need to provide them with higher abstractions, such as mashup editors like Microsoft Popfly or Yahoo Pipes adapted to the real-world.

6. ACKNOWLEDGMENTS

The authors would like to thank the European Commission and the partners of the European IST FP6 project "Service-Oriented Cross-layer infRAstructure for Distributed smart Embedded devices" (SOCRADES - www.socrades.eu) for their support. A special thanks goes to Thomas Pham for his work on the Sun SPOT RESTful architecture, to Markus Weiss for his advices on the Energy Visualizer Prototype, and to Samuel Wieland for his help on the Smart Gateway Architecture.

7. REFERENCES

[1] A. Dunkels and J. Vasseur. Ip for smart objects alliance. Internet Protocol for Smart Objects (IPSO) Alliance White paper No.2, September 2008.

[2] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Techn.*, 2(2):115–150, 2002.

[3] H. Gellersen, C. Fischer, D. Guinard, R. Gostner, G. Kortuem, C. Kray, E. Rukzio, and S. Streng. Supporting device discovery and spontaneous interaction with spatial references. *Journal of Personal and Ubiquitous Computing.*

[4] J. Hui and D. Culler. Extending IP to Low-Power, wireless personal area networks. *Internet Computing, IEEE*, 12(4):37–45, 2008.

[5] A. Kansal, S. Nath, J. Liu, and F. Zhao. SenseWeb: an infrastructure for shared sensing. *IEEE Multimedia*, 14(4):8–13, 2007.

[6] 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.*, 7(5):365–376, 2002.

[7] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim. Tinyrest - a protocol for integrating sensor networks into the internet. Stockholm, Sweden, 2005.

[8] S. Mäkeläinen and T. Alakoski. *Fixed-Mobile Hybrid Mashups: Applying the REST Principles to Mobile-Specific Resources*, pages 172–182. 2008.

[9] C. Pautasso, O. Zimmermann, and F. Leymann. RESTful Web services vs. "big" Web services: making the right architectural decision. In *WWW '08: Proceeding of the 17th international conference on World Wide Web.* ACM, 2008.

[10] C. Prehofer, J. van Gurp, and C. di Flora. Towards the web as a platform for ubiquitous applications in smart spaces. In *Second Workshop on Requirements and Solutions for Pervasive Software Infrastructures (RSPSI), at Ubicomp*, 2007.

[11] 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*, pages 253–266, New York, NY, USA, 2008. ACM.

[12] L. Richardson and S. Ruby. *RESTful Web Services.* O'Reilly Media, Inc., May 2007.

[13] P. Schramm, E. Naroska, P. Resch, J. Platte, H. Linde, G. Stromberg, and T. Sturm. A service gateway for networked sensor systems. *Pervasive Computing, IEEE*, 3(1):66–74, Jan.-March 2004.

[14] V. Stirbu. Towards a restful plug and play experience in the web of things. In *IEEE International Conference on Semantic Computing*, pages 512–517, Aug. 2008.

[15] 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*, pages 282–290, 2003.

[16] V. Trifa, S. Wieland, and D. Guinard. Design and implementation of a gateway for web-based interaction and management of embedded devices. In *Submitted to DCOSS.*, 2009.

[17] E. Wilde. Putting things to REST. Technical Report UCB iSchool Report 2007-015, School of Information, UC Berkeley, November 2007.