A Resource Oriented Architecture for the Web of Things

Dominique Guinard
Inst. for Pervasive Computing
ETH Zurich
and SAP Research, Zurich
Email: dguinard@inf.ethz.ch

Vlad Trifa
Inst. for Pervasive Computing
ETH Zurich
and SAP Research, Zurich
Email: trifa@inf.ethz.ch

Erik Wilde
School of Information
UC Berkeley
Email: dret@berkeley.edu

***Abstract*—Many efforts are centered around creating large-scale networks of "smart things" found in the physical world (e.g., wireless sensor and actuator networks, embedded devices, tagged objects). 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, Python, JavaScript, PHP) can be used to easily build applications involving smart things and users can leverage well-known Web mechanisms (e.g., browsing, searching, bookmarking, caching, linking) to interact and share these devices. In this paper, we begin by describing the *Web of Things* architecture and best-practices based on the RESTful principles that have already contributed to the popular success, scalability, and modularity of the traditional Web. We then discuss several prototypes designed in accordance with these principles to connect environmental sensor nodes and an energy monitoring system to the World Wide Web. We finally show how Web-enabled smart 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, for the most part, disconnected from the Web and form a myriad of small incompatible islands. Increasingly, embedded devices and consumer electronics as for example Chumby,[1] IoBridge,[2] or Nabaztag[3] get Internet (and sometimes Web) connectivity but, however, cannot be controlled and monitored without 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 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 are considering smart things as first-class citizens of the Web. We position the Web of Things as a refinement of the Internet of Things by integrating smart things not only into the Internet (the network), but into the Web (the application layer).

To achieve this goal, we propose to reuse and adapt patterns commonly used for the Web, and introduce an architecture for the Web of Things. We embed Web servers [2], [3], [4] on smart things and apply 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 and some of the recent real-time Web technologies 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. Smart things are 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 users understand well [8]. Applications can be built upon them using well-known Web languages and technologies (e.g., HTML, JavaScript, Ajax PHP, Python, Mashup tools, etc.) 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 implement the REST architectural style, we finally propose the use of *Smart Gateways* [9] which are embedded Web servers that abstract communications 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

[1]`http://www.chumby.com`
[2]`http://www.iobridge.com`
[3]`http://www.nabaztag.com`

several prototypes. We begin by applying these to sensors, actuators and energy meters and show how simple Web applications 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 bar-codes) to objects to direct the user to pages on the Web containing information about the objects [10]. 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 [11]. 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 reusable 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. The advent of WS-* Web Services (SOAP, WSDL, etc.) led to a number of work towards deploying them on embedded devices and sensor networks [12], [13]. 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 [14] 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 [15]. However it focuses mainly on the discovery of devices and not on how to provide their functionality on the Web. Closer to our work, [16] and in particular [17] 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.

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 Web of Things. 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 back-ends 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 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 an active research topic and complex interactions patterns (e.g, transactions) are not yet established as specifications or design patterns. Although the design goals of RESTful systems and their advantages for a decentralized and massive-scale service system align well the field of pervasive computing: millions to billions of available resources and loosely coupled clients, with potentially millions of concurrent interactions with one service provider. Based on these observations, we argue that RESTful architectures are the most effective solution for the Web of Things, as they scale better and are more robust than RPC-based architectures.

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 [18]. 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*) [19] 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*[4], 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*[5], which provide hooks to receive push notifications in regular browser-based applications. HTML5's *Web Sockets*[6] provide a full-duplex communication in the form of a connection opened in the Web browser and accessible through a JavaScript API.

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.

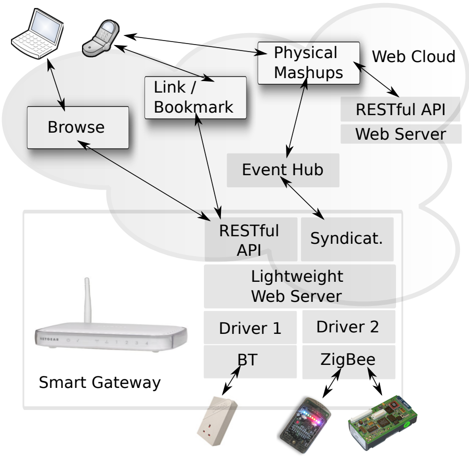

Fig. 1. Web and Internet integration with Smart Gateways (bottom), direct integration (upper-right).

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 based on HTTP in both cases.

As an example, consider a request to a sensor node coming from the Web through 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.

V. Prototyping the Web of Things

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. 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 as described before. The final layer is the Web user interface.

The Ploggs Smart Gateway is 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[7]) 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. Additionally to discovering the Ploggs and mapping

[4]http://code.google.com/p/pubsubhubbub/
[5]http://dev.w3.org/html5/eventsource/
[6]http://dev.w3.org/html5/websockets/
[7]http://code.google.com/p/mongoose

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 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
`http://example.com/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://example.com/SmartMeters.json`
as well. As a second step, the client selects the device it wants to interact with identified by a URI (C1):
`http://example.com/SmartMeters/RoomLamp.json`
By issuing a `GET` request on this resource it gets back its JSON representation as shown on Figure 2. In the response message of Figure 2 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 OPTIONS method from the HTTP standard, 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 JSON representation {``status``:``off``}, the lamp is turned off. Overall, Web-enabling 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.

```
HTTP/1.1 200 OK
Content-Type: application/json
{
  "name": "RoomLamp", "watts": 60.52,
  "KWh": 40.3, "maxWattage": 80.56,
  "links": [{"aggregate": "../all"},
  {"status": "/status"}, ...]
}
```

Fig. 2. A sample HTTP response sent back to the client. The message contains HTTP headers as well as a JSON document in the message body.

B. *Direct Access and Syndication of Wireless Sensor Networks*

The Sun SPOT platform[8] is a wireless sensor network particularly suited for rapid prototyping of WSNs (Wireless Sensor Networks) applications. Each Sun SPOT has a few sensors (light, temperature, accelerometer, etc.), actuators (digital outputs, LEDs, etc.), and a number of internal components (radio, battery). A RESTful Web server is used to make the sensors, actuators and internal components available as REST resources. We have implemented the two Web integration architectures presented in Figure 1. In the first case, an embedded Web server runs on each node and serves directly HTTP, 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. In the second case — called *synchronization-based* driver — we use the idea of Smart Gateway that translates REST requests to proprietary protocols. The smart gateway has a local copy of the devices and minimizes the actual communication between devices and the Web by locally caching the status of devices. In both cases, devices are accessed transparently using actual HTTP requests. Just like 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 allow to serve only a JSON representation of their resources. To avoid too large workload on the node we also 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 at the same time be monitored by subscribing to feeds (node push). 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`

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

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* syndication onto an external powerful server.

C. Physical Mashups

By implementing the suggested architecture for the Ploggs and the Sun SPOTs, 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 mixing both virtual resources and smart things, and describe two such applications below.

1) An Energy-aware Web Dashboard: 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*[9] project is to offer a Web dashboard that enables people to control and experiment with the energy consumption of their appliances.

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)*[10] 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://example.com/SmartMeters/all.json`
on a regular basis. It then feeds the resulting JSON document to the corresponding graphs widgets which can directly parse JSON.

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[11], a Firefox plugin written on top of the Dojo AJAX library for creating 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, but it can easily access RESTful services available on the Web. Thus, creation of Clickscript building blocks (or widgets) based on Web of Things devices is straightforward, just as is their combination into mashups. The mashup shown in Figure 3 gets the room temperature by `GET`ting the Sun SPOT temperature resource. If this is smaller

[9]`http://www.webofthings.com/energievisible`
[10]`http://code.google.com/webtoolkit/`
[11]`http://www.clickscript.ch`

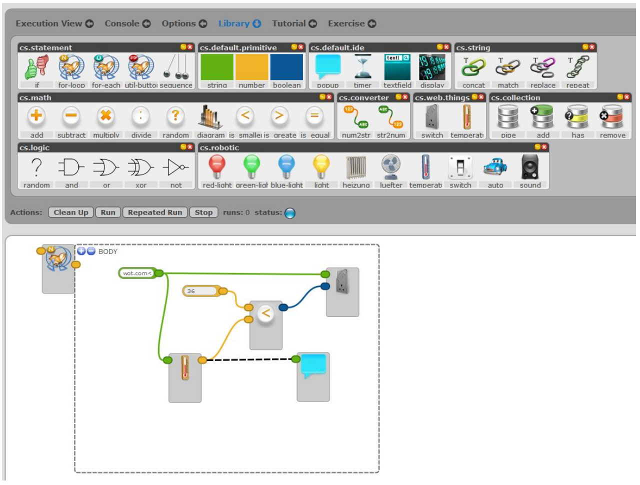

Fig. 3. Using the Web-based Clickscript Mashup Editor to create a physical mashup by connecting building blocks directly from a browser.

than 36 degrees Celsius, it `PUT`s `status=off` to a Plogg which turns off the fan it is connected to.

VI. Evaluation

Our results suggest that the verbosity of the HTTP protocol does not prevent highly efficient applications to be implemented, even when low-power wireless nodes communicate using HTTP in place of highly optimized and compressed messages. In applications where the raw performance and battery life-time are critical, for example when nodes run on battery in large-scale and long-lived deployments — as is often the case with wireless sensor network applications — optimized protocols that minimize network connection and latency will remain the best option. However, when devices are connected to a power source and when sub-second latency can be tolerated, then the advantages of HTTP clearly outweigh the loss in performance and latency.

In many pervasive scenarios, humans are the main actors that interact with digitally augmented environments. In most of these cases, the loss in performance is hardly perceivable by humans. To support this statement, we have implemented a simple scenario where a user (on the same machine) issues a GET request to read the current light sensor value of a Sun SPOT located one radio hop away from a gateway. We compare the two different architectures described in Section IV and show the round-trip time for each request in Figure 4. In the first case, each request is routed through the proxy to the embedded HTTP server running on the remote Sun SPOT where it is served. In this case, the average round-trip time over 10’000 consecutive request is 205 milliseconds (min 97 ms, max 8.5 seconds).

In the second case, we use a synchronization-based architecture — that is each Sun SPOT periodically sends its sensor readings to the proxy where they are cached locally. Each request is then served directly from this cache without accessing the actual device, in which case the average round-trip time was 4 ms (min 2 ms, max 49 ms). This has the

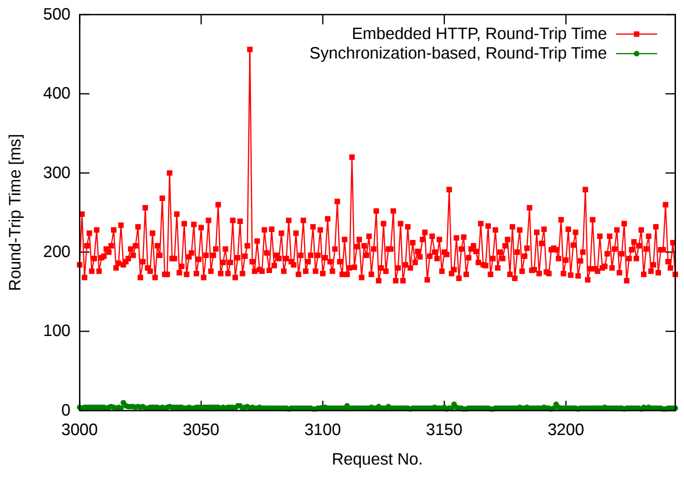

Fig. 4. Round-trip time for concurrent GET requests on a single sensor. With the embedded HTTP running on the device, the response time is higher because the HTTP request must be propagated to the device over a radio connection, while it is served directly from the gateway in the synchronization version.

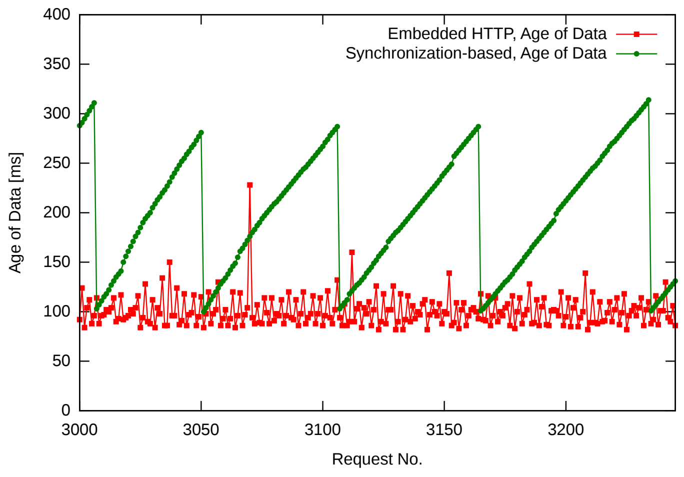

Fig. 5. Data age for a given sensor reading. With the synchronization-based driver it grows until refreshed, while with embedded HTTP the data age is only approximately the radio message propagation time from the Sun SPOT.

advantage of fully decoupling the Sun SPOT from the outside world, as it only needs to send an update packet with a frequency short enough to ensure the validity of data. On the other hand, the staleness of the retrieved data will depend on the frequency of updates sent by the device, while routing the HTTP request has the advantage of returning always the most recent sensor reading when the request was processed. Figure 5 shows the age of sensor data (the radio propagation delay with the embedded server, and the delay + time since last update with the sync-based solution).

High-performance caches that can easily scale with the number of concurrent HTTP requests are common nowadays. This means that using a synchronization-based mechanism, thousands of HTTP clients can retrieve simultaneously sensor data from a single device with extremely low response time, while still preserving the freshness of the data collected under a reasonable bound for many applications. Obviously, this will not hold true for non-cacheable (write) requests that must be sent to devices (e.g., turn on LEDs, change application state). As many distributed monitoring applications are usually read-only during their operation (sensors collect data, but users cannot change their status), our architecture exhibits a scalability level for sensing applications that has not been reached before. This enables new types of applications where physical sensors can be shared with thousands of users with little, if any, impact on the latency and data staleness, for example tracking public transportation with sub-second accuracy.

a) Early Qualitative Evaluation: The Plogg RESTful Gateway and the Sun SPOTs have been used by two external development teams, which hints some of the qualitative advantages developers can gain from the proposed the architecture. 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 the developers, 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 Web browsers as one of the most prevalent benefits.

VII. Discussion and Open Challenges

Web 2.0 mashups have significantly lowered the entry barrier for the development of Web applications. Certainly, a resource-oriented approach is not the universal solution for every problem. Scenarios with 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 required, Web standards can simplify integration of the real-world data with Web content. Applying the same design principles that contributed to the success of the Web — in particular openness and simplicity — leverages the versatility of the Web as a common ground to network devices and applications. 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 and physical things can be bookmarked, browsed, searched for, and used just like any other Web resource.

Based on our experience, we suggest that the drawbacks of Web architectures are largely offset by the simplification of the design, integration, and deployment processes [4], especially in comparison with other protocols for embedded devices, such as SOAP-based Web services. Although HTTP introduces a communication overhead and increases average response latency, in many pervasive scenarios this overhead does not affect user experience [17], [12]. For example, we have argued [9] that the performance of using HTTP as a data exchange protocol is mostly sufficient for common pervasive scenarios, especially when only few users are accessing the same resource simultaneously (200 ms average response time was reported with 100 concurrent users on a 1.1 GHz server).

HTTP was designed as an architecture where clients initiate interactions. This 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. Adapting the client-server architecture is now a core research topic in the Web community [19]. Standards such as HTML5 are also going towards asynchronous bi-directional communication and emphasize on how relevant it is to further explore lightweight Web-based messaging systems.

Another important challenge for the global Web of Things is the search and discovery of smart things. Consider billions of things connected to the Web. 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, are often moving from one context to the other and their HTML representations are less keywords-rich than traditional Web pages. This problem is not inherent to smart things but more generally a problem in describing services on the Web. Recent developments in semantic descriptions that can be embedded in HTML representations such as Microformats[12] or RDFa will certainly help making better sense of the services offered in the Web of Things.

VIII. Conclusion

In this article we describe the basics of the Web of Things architecture. By formalizing the various design parameters to consider, we provide a practical framework for users to build their own WoT devices and applications. We illustrate with concrete examples how various applications types can be built on top of the proposed architecture, and propose how the emerging real-time Web techniques can be applied to develop Web-compliant, highly interactive and integrable physical mashups. Finally, we provide an initial performance analysis and discussions to support the future research efforts that will make the Web of Things a reality.

Acknowledgements

The authors would like to thank Simon Mayer and Thomas Pham for their work on the RESTful Sun SPOTs as well as Lukas Naef for providing us with the initial Clickscript environment.

[12]`http://www.microformats.org`

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. Vandewalle, "The Web of Things: Interconnecting Devices with High Usability and Performance," in *Proc. of the International Conference on Embedded Software and Systems, ICESS '09*, Hangzhou, Zhejiang, China, 2009, pp. 323–330.

[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'09)*, Pittsburgh, USA, 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*, 1st ed. O'Reilly Media, May 2007.

[7] C. Pautasso and E. Wilde, "Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design," in *Proc. of the 18th International World Wide Web Conference (WWW'09)*, Madrid, Spain, 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 *In Proc. 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 *In Proc. of the SIGCHI conference on Human factors in computing systems (CHI)*. New York, NY, USA: ACM, 1999, pp. 370–377.

[11] 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*, vol. 3, no. 1, pp. 66–74, Jan.-March 2004.

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

[13] L. M. S. de Souza, P. Spiess, D. Guinard, M. Koehler, S. Karnouskos, and D. Savio, "SOCRADES: A Web Service based Shop Floor Integration Infrastructure," in *Proc. of the First International Conference on the Internet of Things (IOT'08)*. Zurich, Switzerland: Springer, 2008.

[14] 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.

[15] "Towards a RESTful Plug and Play Experience in the Web of Things," Santa Clara, USA.

[16] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim, "TinyREST - A protocol for integrating sensor networks into the internet," in *Proc. of the Workshop on Real-World Wireless Sensor Network: REALWSN*, Stockholm, Sweden, 2005.

[17] 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 (MASS'04)*, Lauderdale, Florida, 2004, pp. 340–348.

[18] E. Wilde, "Feeds as Query Result Serializations," School of Information, UC Berkeley, Berkeley, California, Tech. Rep. 2009-030, April 2009.

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