Design and Implementation of a Gateway for Web-based Interaction and Management of Embedded Devices

Vlad Trifa[1,2], Samuel Wieland[1], Dominique Guinard[1,2], and Thomas Bohnert[2]

[1] Institute for Pervasive Computing, ETH Zurich
[2] SAP Research CEC Zurich
8008 Zurich, Switzerland
Corresponding author: vlad.trifa@ieee.org

Abstract. Wireless Sensor Networks provide unprecedented possibilities for monitoring and interacting with the real-world. Unfortunately, the lack of open and simple standards for ad-hoc collaboration between heterogeneous embedded devices makes it difficult to build large-scale deployments; every particular application requires complex integration work, and therefore technical expertise, effort and time. Inspired by the success of Web 2.0 mashups, we propose a similar lightweight approach for interacting with networked devices. In particular, we describe a gateway architecture that enables to access sensor nodes through a RESTful interface. With this approach, interacting with a sensor node becomes as easy as typing a URI in a Web browser. By reusing the architectural principles of the modern Web, we show how one can built a loosely coupled infrastructure for the Web of Things that scales well and extends the current Web to the real world.

1 Introduction

Although the field of wireless sensor networks (WSN) is still in its infancy, companies that sell embedded sensing systems are already flourishing (Sentilla, Arch Rock, Streetline, etc). These devices provide unprecedented possibilities for monitoring and interacting with the real-world and could be an invaluable help in many disciplines. Unfortunately, most sensor network projects have focused on building vertical solutions designed for very specific applications that run as isolated, small scale testbeds. As a consequence, the lack of commonly agreed standards for sensor networks has resulted in a wide variety of hardware and software platforms that are usually incompatible. Without a set of simple and open standards for ad-hoc networking and interaction with embedded devices, building and maintaining large-scale sensing applications on top of embedded devices requires extensive effort and expert knowledge, in particular when devices from different constructors are used. To hide the complexity and heterogeneity of devices and network protocols used, various middlewares for sensor networks have been proposed [1]. Unfortunately, most existing approaches are too complex for non-experts, and also are based on tightly coupled components, which strongly affects the scalability and evolvability of the whole system. As a consequence, a worldwide Sensor Web (also called the *Internet of Things*) where billions of "smart" objects are shared and can be easily reused, still remains unfeasible.

As demonstrated by the success of the Web, loosely coupled approaches posses high scalability and robustness - which are fundamental properties for building a worldwide network of devices. Furthermore, the real value of such applications comes from the sharing and integration of data among heterogenous devices. Based on these considerations, we describe in this article a generic and easy to use middleware to enable ad-hoc interaction between embedded devices. Our key requirements are to maximize reuse and sharing of embedded devices, while minimizing the time needed for fast prototyping applications that run on top of physical sensors. Because the number of embedded devices with direct Internet connectivity is rapidly growing, we propose to fully leverage the existing and ubiquitous Web standards to build a middleware for embedded devices.

Our approach is based on *smart gateways*, which are lightweight and extensible software components that enable Web-based interactions with all kinds of embedded devices. In addition, gateways can be linked together to form hierarchical trees that can be further mapped to physical locations. In this manner our approach can highly scale, and at the same time support location-based services. The role of gateways is not to replace existing sensor networks, but to facilitate finding and reusing of shared devices using standard Web technologies. From the perspective of a web programmer, devices become web resources that can be addressed and used to build mashups. More experienced programmers can change the driver implementation that controls the communication between the sensor network and the gateway to fit their particular needs. For WSN developers, the gateway simplifies greatly the process to export data and functionality of the WSN to be provided to end users on the web. Using our gateway system do not require to change existing deployments, but only to implement a specific driver into the gateway to interact with the sensor network.

For certain applications tight coupling using proprietary solutions remains the most desirable choice for building high-peformance systems with real-time requirements (as for example in the industrial automation or banking domain). Specific tight-end coupling can be used in back-end for such very particular tasks (based on proprietary solutions). These systems can expose their functionality in a high-level abstraction that can be accessible from the web in a plug-and-play manner so that it can be used directly over HTTP. However, much simpler loosely coupled approaches are to be preferred for tasks with more modest requirements (which represent most use cases for data monitoring and home applications), because of their inherent flexibility and intuitive use.

2 Related Work

The idea of linking physical objects with the Web is not new, and early approaches used physical tokens (such as bar codes or RFID tags) to retrieve information about objects they were attached to [2, 3]. For example, in the Cooltown project [4] each thing, place, and person have an associated Web page with information about them. Shaman was an early gateway system that enabled low-power devices to be part of wider networks [5]. With advances in computing technology, tiny Web servers could be embedded in most devices [6]. The idea of each device having its own Web page is appealing because device pages could be indexed, searched, and accessed by search engines, and this directly from a Web browser. However, static indexing of mobile devices is not possible when new devices appear and disappear continuously. Besides, the goal of earlier work in Web-enabled devices was to provide an online representation of real things for humans (real-time status displayed on a HTML page), but no attempts to seamlessly integrate devices into the Web as proactive units nor enabling sharing and reuse of device-level functionalities has been mentioned.

Many technologies for building distributed applications on top of heterogeneous devices have been proposed. Among them, the now classical systems are CORBA, JINI, or RMI. JXTA [7] is a set of open protocols for allowing devices to collaborate in a peer-to-peer fashion. JXTA was among the first real attempt to bridge physical objects in the world with the Internet. More recently, Web services have also been used to interconnect devices on top of standard Web protocols [8]. However, these approaches are based on tightly coupled solutions, where each element had full knowledge about the other peers and the functions they offered. Unfortunately, such solutions are too rigid to deal efficiently with the constraints and requirements of mobile embedded devices, in particular for ad-hoc interaction with new devices with unknown properties.

Several systems for integration of sensor systems with the Internet have been proposed [9–11]. SenseWeb [12] is a platform for people to share their sensory readings using Web services to transmit data onto a central server. Pachube [13] offers a similar community Web site for people to share their sensors and uses more open data formats. Unfortunately, these approaches are based on a centralized repository and devices need to be registered before they can publish data, thus are not sufficiently scalable. Prehofer et al. [14] recently proposed a Web-based middleware that is similar to our approach, however, they used the Internet only as a transport protocol, and no references to use a fully Web-like approach has been mentioned. Also, an interesting approach to use the web architecture and the semantic Web technologies can be found in [15].

However, most of the existing Web-based approaches use HTTP only to transport data between devices, whereas HTTP is in fact an application protocol. Projects that specifically focus on re-using the founding principles of the Web as an application protocol are still lacking. As pointed out in [16], creation of devices that are Web-enabled *by design* would facilitate the integration of physical devices with other content on the Web, in which case there would be no need for any additional API or descriptions of resource/function. The approach found in [17] is similar to ours, but focuses mainly on the discovery of devices and a more systematic approach and system evaluation is lacking.

3 Infrastructure for the Web of Things

Even though more and more embedded devices are being connected to the Internet [6, 18], a common ground to enable them to communicate using a uniform abstraction is still lacking. As illustrated by the growing popularity of open source communities and of the do-it-yourself technology, it is very likely that billions of sensors and physical objects throughout the world could be soon available on the Internet. The central research question remains - how can one build a (globally) scalable network of physical devices where these devices can interact together without any *a priori* knowledge about each other? As many appliances have built-in Web servers, and Web access is available practically everywhere with mobile phones, we propose to reuse the core principles of the modern Web architecture for sensor networks to take advantage of the scalability, reliability and "mashupability" properties of the Web. These principles are summarized under the REST architectural style as described in [19].

The success of todays Web can be explained to a great deal to its simplicity and openess. Only a few easy guidelines describe how Web components should be developped and integrated into the Web - and following them leads to so-called RESTful applications:

- Resources are uniquely addressable through URIs.
- The format and type of the resource representation is described through MIME-types which give the communication partner the possiblity to chose the most suitable resource representation available (example: language aware websites).
- Resources can be accessed/modified with HTTP instructions `PUT`, `POST`, `GET`, `DELETE`that are comparable to traditional database operations. All the instructions are self explaining by their name which leads to a much better understanding of Web interfaces.

4 Gateway System Design

The gateway architecture was designed with three main goals in mind: simplicity, extensibility and modularity. Simplicity and extensibility to enable users to extend and customize the gateway to their needs. Modularity so that internal components of the gateway can interact only through small interfaces, thus allowing the evolution and exchange of individual parts of the system. Our gateway implementation is written mainly in Java with several driver components in C/C++ or Python. The architecture is composed out of three major layers - the presentation layer (4.2), the control layer (4.3) and the device abstraction layer 4.4) - each responsible for a well defined set of tasks. Figure 1 shows a high level overview of the gateway.

4.1 Core

The core specifies the "kernel" of the gateway providing utility classes for all the components of the architecture. To ensure that there is only one core available

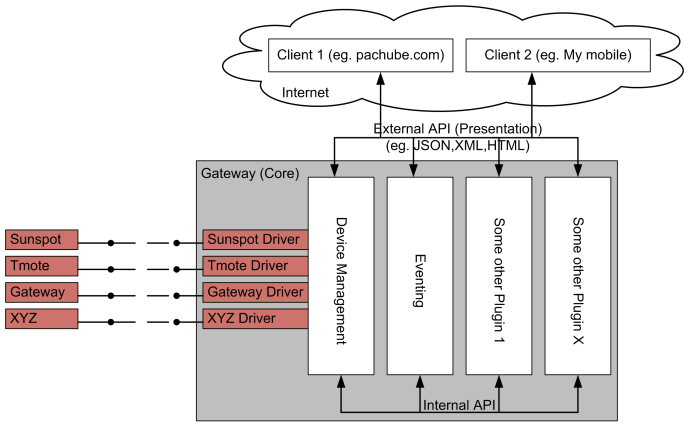


Fig. 1. High level overview of the gateway architecture. The graph shows the REST "External API" towards external clients, The plugin architecture (Eventing, ...) and the device abstraction (Sunspot Driver, Tmote Driver, ...)

at runtime, the "Singleton-pattern" [20] is used where the core can be accessed through the static method *public static Core getInstance()*. At startup configuration files are loaded, the HTTP server for the presentation layer (4.2) is started and the plugins for the control layer (4.3) are initialized. At shutdown the core makes sure that the current configuration of the gateway gets stored back to the configuration files and that all the resources occupied are freed again.

4.2 Presentation

The presentation layer makes the gateway components accessible to the outside world. It is a thin layer on top of the control layer (4.3) managing requests from clients through a REST interface. We have used the Restlet framework[3] as HTTP server in order to model the most important HTTP operations (`GET`, `POST`, `PUT`, `DELETE`).
All the resources on the gateway can currently be retrieved in four different formats (XML, RDF, HTML and plain text) allowing clients to choose their preferred format. In HTTP requests, the MIME type is used to specify the desired format. On the server side, in a first phase an XML is generated and then translated with XSLT stylesheets on the fly. This allows machines to use semantically enriched RDF formats whereas humans can retrieve an HTML representation. In addition new formats can be easily added by using XSLT stylesheets.
In order to make the devices attached to the gateway accessible through the Web, a mapping from device names to URI is performed by the presentation layer. A device with the name "sensor1" will be mapped to "/sensor1". This allows users to browse dynamically the device list. Requests from the web onto

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

such a mapping will be redirected to the responsible device driver that then is responsible to handle the request accordingly (4.4).

4.3 Control

The control layer is composed of several independent components called plugins. A plugin is a software component that is loaded at startup of the gateway through the core (4.1). Users can write their own plugins and place them (packed as a jar file) into the classpath of the gateway. The jar file has to contain a "marker"-file *props/plugins.xml* that contains a descriptor for the plugin with the mandatory fields Version, ID, and Dependency (IDs of plugins that need to be loaded before this plugin). The "marker" contains the class name that will be used later for the class-loading. A sample descriptor has the form:

```
<Descriptor>
   <ID>ch.ethz.inf.vs.gateway.plugin.devices.Devices</ID>
   <Version>1.0.0</Version>
   <Lazyload>true</Lazyload>
   <Dependencies/>
</Descriptor>
```

Plugins are allowed to depend on other plugins (example: the eventing plugin depends on the device management plugin). To keep a loose coupling between plugins, a lightweight synchronization mechanism is implemented with the "observer pattern" [20]. The observer (plugin depending on some other plugin) will be registered on the observable. As soon as the observable changes its state, an internal synchronization method is invoked informing all the observers about the change (example: the devices plugin calls this method whenever a device is removed or added).

Device management plugin The Device Management plugin maintains a high level view on devices registered at the gateway by using the device abstraction (4.4). New device drivers are loaded through dynamic class loading. By sending a `PUT` request to the device management with the parameters *class* (fully qualified name of the class implementing the device driver) and a unique device identifier *ID* new instances of the driver instances can be invoked. Other plugins (or other layers within the architecture) can query and access the devices currently registered at the devices plugin by using the unique id.
On each device driver, a "heartbeat" protocol is installed that requires the device to keep its status "alive". This means that at regular intervals, a watch-dog is invoked and iterates over all the drivers and invokes the method *isAlive(): boolean.* If a driver does not respond with *true*, the driver gets removed from the device list. Note that the implementation does not poll the real device for aliveness - the aliveness test within the driver towards the physical device is usually performed using an optimized manner depending on the device.

Eventing plugin Many sensors read their state in a regular interval and a programmer usually has to check periodically whether this state has changed or not. When using sensors over the Internet, polling is inefficient and creates unnecessary load that can be avoided by using an asynchronous publish/subscribe model.
From the clients perspective, this is a simple `POST`-request to the event registration URI with three mandatory parameters - the leasetime to specifying how long the registration shall be valid, the keyword specifying the type of event the registration is for and the callback giving an address where to deliver the events (see Figure 2). Consider the tree structure from Figure 3. The client registers at the top node (floor1) for a "fire"-event triggered by any of the sensors in the subtree of the "floor1"-gateway. As soon as a "fire"-event is triggered on the gateway, the client will be notified about the event through the callback address provided in the registration request. In our case, the "floor1"-gateway is also connected to other gateways in its subtrees. It therefore has to register on these gateways as well for fire-events. This registration works in exactly the same manner as with the client registration, but this time with the "floor1"-gateway as client on the different "room"-gateways.

```
POST /_eventing/registration HTTP/1.1
Host: ip_of_the_registering_client
leasetime=6000
callback=client_callback_address
keyword=fire
```

Fig. 2. Example for a client event registration. The registration is valid for 6 seconds (6000ms). Events of type "fire" will be sent to the callback "client_callback_address".

In the current implementation of the gateway, we use this simple eventing approach. However, more advanced approaches would be more appropriate for larger scale implementations. We are currently investigating message broker methods such as MQTT[4] or XMPP[5].

4.4 Device Abstraction

Like in almost any modern operating system, the gateway provides an abstraction for devices. For higher level applications, any type of device looks the same, even if the underlying implementation differs. The mechanism is comparable to the one used in Section 4.3.
The device abstraction is illustrated on the left part of Figure 1. Different specialized drivers are used to communicate through their "proprietary" protocol with

[4] `http://mqtt.org/`
[5] `http://xmpp.org/`

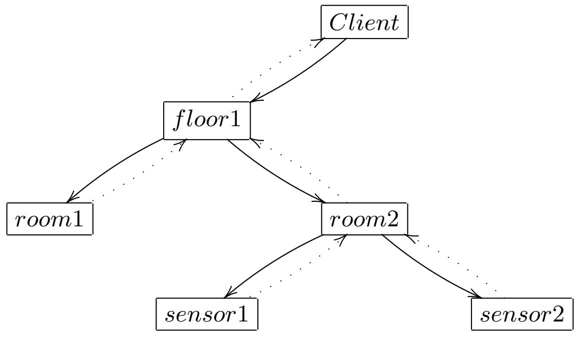


Fig. 3. Sample gateway hierarchy with a gateway at the top of the tree (floor1) with two gateways in the subtree (room1 and room2). The solid arrows indicate the path the registration message takes through the gateway. The pointed arrows indicate the path the event message follow.

the respective physical device (example: SunspotDriver communicates through ZigBee with the SunSpot). However, from a higher level perspective all the drivers implement the abstract class *Device* allowing the device management plugin to treat all the devices in the same manner.
For devices that are already Web-enabled (that is they support HTTP over TCP/IP), the driver implementation can simply forward requests from the presentation layer (4.2) to the physical device. However, when the device is not Web-enabled (e.g. sunspots do not have an IP stack), the driver is responsible to translate the web request into a protocol understood by the physical device.
A crucial feature of a device driver is the capability to act as a resource representation (or proxy) for the underlying physical device. Consider a temperature sensor changing its temperature seldomly. Instead of polling the device every time a client requests the temperature, the driver can store the temperature and return the value directly, thus minimizing the actual communication with devices. This caching mechanism is very useful for shared access to quasi real-time sensor data collected with low-power devices.
Along with the eventing mechanism (4.3) and with the fact that all devices are accessible as Web resources, it is easy to compose higher level devices by combining devices from lower level devices (for example the "RoomState" virtual resource in Section 5, which is actually a combination of physical sensors).

5 Experimental Design

To illustrate the different properties of the proposed gateway architecture, we have implemented a simple prototype scenario illustrated in Figure 4. The gateway was installed on a laptop computer (2 x 1.6GHz, 2GB RAM), where we attached a SunSpot base station for ZigBee communication and a Tikitag RFID reader[6] to read the Tikitag RFID tags. As SunSpots per se are not IP capable, a reverse proxy (described in [21]) is used to multiplex ZigBee communication streams so that the SunSpots can access Web content directly through HTTP. The virtual RoomState device simulates a temperature regulator of the room. This regulator can increase or decrease the temperature, and is capable of returning the current value (both directly using REST).

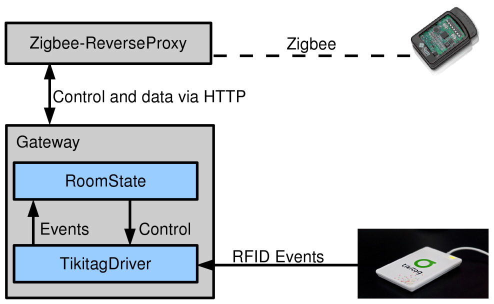


Fig. 4. Prototype setup. The gateway maintains a virtual roomState device. The Tikitag-reader is attached through the Tikitag driver.The SunSpot connects to the gateway with HTTP through the reverse proxy.

The RoomState device has registered for events sent by the Tikitag driver. By placing different Tikitags onto the RFID reader, users can dynamically adjust the value of the roomstate (e.g. using tag A increases the temperature on the RoomState device by one degree, whereas tag B decreases the value accordingly). The SunSpot periodically polls the RoomState device on the gateway using a `GET`on the URI of the RoomState resource to read the current value (the temperature of the room). The value is then displayed using different LED colors (e.g., blue when the temperature is below 10 degrees, green when between 10 and 25 degrees and red when over 25 degrees). Additionally, the SunSpot can also be used as an input to control the room temperature, for example by using the acceleration sensor onboard (shaking the sunspot for 5 seconds will issue a `POST`request on the RoomState resource). This simple system illustrates a trivial mashup built on top of completely different devices, where their interaction is only defined by calling different URIs.

6 System Performance Evaluation

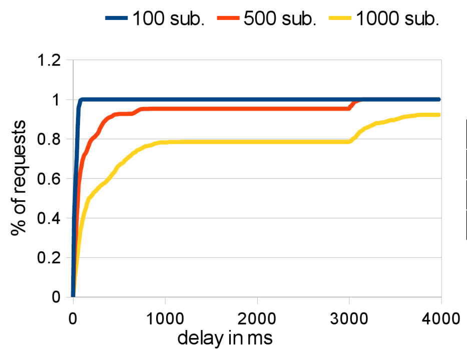
Based on the prototype we described in Section 5, in this section we evaluate a few interesting aspects with a larger evaluation setup. We have implemented the gateway on two different test platforms.

[6] `http://www.tikitag.com/`

- Two computers both running gentoo gnu linux with sun-jre-1.5.0.17 linked by a 1Gbit ethernet. The gateway was installed on the test server (1.1GHz, 2GB RAM, 1Gbit NIC). The test client (2 x 2.13GHz, 8GB RAM, 1Gbit NIC) simulated several clients "calling" the server.
- One NSLU running debian gnu linux with sun-jre-1.5.0 as test server (133MHz, 16MB RAM, 100Mbit NIC) and a test client (2 x 2.13GHz, 8GB RAM, 1Gbit NIC).

6.1 Subscriber Churn

In this test, we evaluate the gateway eventing mechanism under high churn (meaning that many clients are registering and leaving). The parameters for testbed 1 (Figure 5) have been set to 100, 500 and 1000 simultaneous clients, for testbed 2 (Figure 6) to 20, 40 and 100.



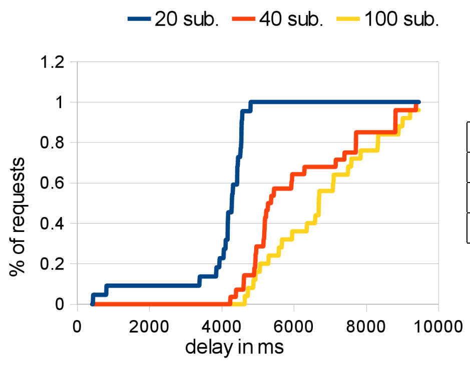
	100 sub.	500 sub.	1000 sub.
RT 50%	20ms	50ms	170ms
RT 80%	50ms	200ms	3020ms
Mean	24ms	243s	1325ms

Fig. 5. Eventing mechanism in an environment with high churn. The x-axis shows the delay in miliseconds, the y-axis the percentage of requests fulfilled at the delay (testbed 1 with 1.1GHz server).

When the gateway is installed on the NSLU (setup 2), the results might indicate a performance problem. To verify this assumption, we must run the tests also on different hardware with comparable CPU power. On the faster server, however, the test results indicate that the gateway scales nicely with a large number of simultaneous clients. With a moderate increase of computational power (CPU and memory), a much larger number of clients could be supported.

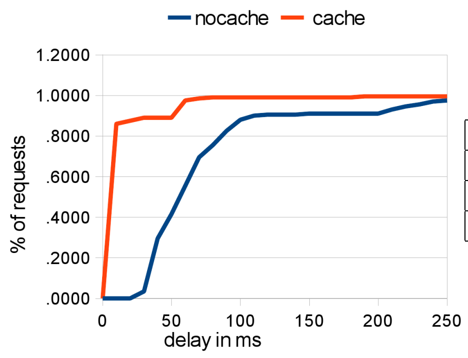
6.2 Caching

To speed up the response time when delivering a page that has to be generated, a caching mechanism has been used within the gateway. As long as the representation did not change, the generated representation will be cached and the



	20 sub.	40 sub.	100 sub.
RT 50%	4280ms	5280ms	6690ms
RT 80%	4540ms	7720ms	8320ms
Mean	3942ms	6319ms	6800ms

Fig. 6. Eventing mechanism in an environment with high churn. The x-axis shows the delay in miliseconds, the y-axis the percentage of requests fulfilled at the delay (testbed 2 with 133MHz NSLU server).



	no cache	with cache
RT 50%	55ms	5ms
RT 80%	90ms	10ms
Mean	80ms	14ms

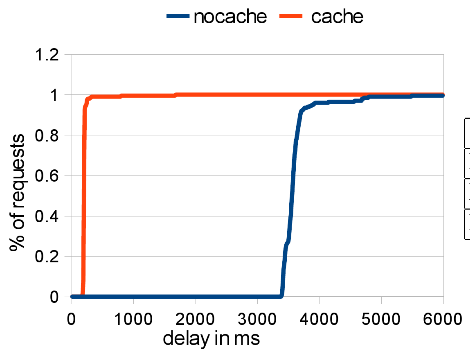
Fig. 7. Response times measured with/without cache on testbed 1 (1.1GHz server).

cache will be returned upon subsequent requests. In testbed 1 (Figure 7) and testbed 2 (Figure 8) 150 `GET`requests have been performed sequentially.

In both testbeds caching reduced response time significantly.

7 Discussions

In this article, we have described a lightweight gateway architecture to enable Web-based interaction with low-power devices that do not have direct connections with IP-based networks. As the gateways can be easily extended to incorporate new devices, users can easily create drivers for any embedded device or sensor network. This will enable to expose their functionality as URI-identified resources that are directly accessible on the Web, thus can be manipulated using HTTP.



	no cache	with cache
RT 50%	3560ms	195ms
RT 80%	3640ms	205ms
Mean	3604ms	211ms

Fig. 8. Response times measured with/without cache on testbed 2 (133MHz NSLU).

We have shown that the gateway eventing infrastructure can scale enough to support many simultaneous clients with a reasonable speed. In most applications, room-level gateways will have only a few devices and users connected simultaneously, therefore are sufficient for most use cases where a delay of a few seconds is tolerable. The tests with the NSLU device have shown possible bottlenecks when dealing with constrained devices, so further work is required to reduce the computational requirements. Nevertheless, the results we have obtained are encouraging and the overhead introduced by our system when gateways are connected directly to the Internet are negligible. However, the overhead increases quickly when gateways are combined into complex trees, unless all the gateways in the system can be accessed directly from the Internet.

In addition, we have shown how caching of sensor values reduced response time significantly. This is especially true for reading data from physical devices that seldomly change their state. However, in the case where the cache entry is invalidated right after generation by a state change on the device, the cache maintenance overhead could degrade overall performance of the gateway. For most monitoring application where devices are shared publicly, data from devices can be accessed directly using the resource URI, while in fact the data is being generated by the gateways, and no real communication with the device is taking place. This process is fully transparent to Web clients, so this contributes highly to the global scalability of the system, even when the gateway is running on a small device such as the NSLU.

Web standards allow any device to speak the same language as other resources on the Web, making it much easier to integrate the real world with any other Web content, so that physical things can be bookmarked, browsed, googled, and used just like any other Web page. However, a resource-oriented approach shall not be religiously considered as the miracle solution for all problems. In particular, tightly coupled system that involve very specific functionalities and need high performance would still benefit more from the traditional RPC-based approaches. Nonetheless, 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 within Web applications.

8 Conclusions and Future Work

In this paper we have shown how the Web can be used to build a network of heterogenous, yet interoperable, devices that can be found and used both by machines and humans. By using a resource-oriented approach and REST, instead of SOAP-based Web Services, we fully leverage the existing Web standards to increase the modularity and interoperability of the different components. This results in more flexible applications that scale well, but also that blend symbiotically with the existing Web without requiring any change. We have described a prototype of such a fully web-compliant architecture suited for wireless sensor networks. We have used this prototype to illustrate how one can very easily combine real-time information from different physical devices without requiring any advanced knowledge in Web Services and complex middleware systems. Our proposal is to reuse the core principles of the Web (REST) as a basis for a lighter and standardized middleware for networking all kinds of devices. By considering sensor networks as distributed Web applications, one can shift from monolithic applications towards a fully decentralized and flexible solution. Such an open and standardized middleware would encourage widespread adoption by maximizing reuse and lowering the access barrier for people to use and develop applications on a much wider ecosystem of interconnected devices. In the long run, extending the Web to easily integrate physical devices will reshape the Internet into a multipurpose collection of physical and virtual resources that can be easily (re)combined at run-time to solve any task at hand.

References

1. Wang, M.M., Cao, J.N., Li, J., Dasi, S.K.: Middleware for wireless sensor networks: A survey. Journal of Computer Science and Technology **23**(3) (May 2008) 305–326
2. Roy, W., P., F.K., Anuj, G., L., H.B.: 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) 370–377
3. Ljungstrand, P., Redström, J., Holmquist, L.E.: 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) 23–31
4. Kindberg, T., Barton, J., Morgan, J., Becker, G., Caswell, D., Debaty, P., Gopal, G., Frid, M., Krishnan, V., Morris, H., Schettino, J., Serra, B., Spasojevic, M.: People, places, things: web presence for the real world. Mob. Netw. Appl. **7**(5) (2002) 365–376
5. Schramm, P., Naroska, E., Resch, P., Platte, J., Linde, H.: Integration of limited servers into pervasive computing environments using dynamic gateway services.

Technical Report 0202, Computer Engineering Institute, University Dortmund, Germany (2002)

6. Borriello, G., Want, R.: Embedded computation meets the World Wide Web. Commun. ACM **43**(5) (2000) 59–66
7. Traversat, B., Abdelaziz, M., Doolin, D., Duigou, M., Hugly, J., Pouyoul, E.: Project JXTA-C: Enabling a Web of Things. In: Proceedings of the 36th Annual Hawaii International Conference on System Sciences. (2003) 282–290
8. Priyantha, N.B., Kansal, A., Goraczko, M., Zhao, F.: 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) 253–266
9. Gibbons, P., Karp, B., Ke, Y., Nath, S., Seshan, S.: IrisNet: an Architecture for a Worldwide Sensor Web. IEEE Pervasive Computing **2**(4) (2003) 22–33
10. Balazinska, M., Deshpande, A., Franklin, M., Gibbons, P., Gray, J., Nath, S., Hansen, M., Liebhold, M., Szalay, A., Tao, V.: Data management in the worldwide sensor web. Pervasive Computing, IEEE **6**(2) (2007) 30–40
11. Open Geospatial Consortium Inc.: OGC Sensor Web Enablement: Overview and High Level Architecture. White paper OGC 07-165 (2007)
12. Kansal, A., Nath, S., Liu, J., Zhao, F.: SenseWeb: an infrastructure for shared sensing. IEEE Multimedia **14**(4) (2007) 8–13
13. Haque, O.: Pachube. Online at `http://www.pachube.com`
14. Prehofer, C., van Gurp, J., di Flora, C.: 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. (2007)
15. Vazquez, J.I., de Ipiña, D.L., Sedano, I.: Soam: A web-powered architecture for designing and deploying pervasive semantic devices. IJWIS - International Journal of Web Information Systems **2**(3-4) (2006)
16. Wilde, E.: Putting things to rest. Technical Report UCB iSchool Report 2007-015, School of Information, UC Berkeley (November 2007)
17. Stirbu, V.: Towards a RESTful Plug and Play Experience in the Web of Things. IEEE International Conference on Semantic Computing (Aug. 2008) 512–517
18. Dunkels, A., Vasseur, J.: IP for Smart Objects Alliance. Internet Protocol for Smart Objects (IPSO) Alliance White paper (September 2008)
19. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine, Irvine, California (2000)
20. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (November 1994)
21. Guinard, D., Trifa, V., Pham, T., Liechti, O.: Towards Physical Mashups in the Web of Things. In: Proceedings of the 6th International Conference on Networked Sensing Systems (INSS). (2009)