Web Messaging for Open and Scalable Distributed Sensing Applications

Vlad Trifa[1,2,⋆], Dominique Guinard[1,2], Vlatko Davidovski[1], Andreas Kamilaris[3], and Ivan Delchev[2]

[1] Institute for Pervasive Computing, ETH Zurich, Switzerland
[2] SAP Research, Zurich, Switzerland
[3] University of Cyprus, Nicosia, Cyprus
vlad.trifa@ieee.org

Abstract. Future Web applications will increasingly require real-time data from the physical world collected by a myriad of sensors and actuators. Currently, integration of such devices require customized solutions due to the lack of widely adopted protocols for devices. Because the Web architecture offers a high degree of interoperability and a low entry barrier, we propose to leverage the Web to build hybrid applications that combine the physical world with Web content. Our work builds upon recent developments in Web push techniques and extends them for embedded devices with a RESTful messaging system. Our results illustrate that fully Web-based distributed sensing applications are not only feasible - but actually desirable - because Web standards offer an ideal compromise between performance and functionality.

1 Introduction

In the last decade computers have been silently pervading every corner of our lives. Among them, networks of tiny sensors that gather data about the real world – called *Wireless Sensor Networks* (WSN) – have been increasingly used in various disciplines ranging from civil engineering to biology. Because of their deeply embedded nature, WSN have the potential to revolutionize our understanding of natural and physical systems. As these systems will mature, the data they collect will increasingly need to be available on the Web in real time. A scenario for such applications is a heterogeneous city-wide Web API as shown in Fig. 1, to share real time information about the status of a city with different consumers. This data could be private and secure information (alarms, fire alerts, household energy consumption), but also public information (air and noise pollution levels, traffic status, etc).

Programming wireless sensor networks is challenging because devices have limited energy and computational resources that must be carefully managed. In addition, WSNs are highly dynamic and transient: connectivity is often unpredictable, devices disappear and reappear, new ones are added to extend the

⋆ Corresponding author.

B. Benatallah et al. (Eds.): ICWE 2010, LNCS 6189, pp. 129–143, 2010.

network or to replace failed ones. In such conditions, a high-level *data-centric* approach in which information is delivered based on content rather than explicit addressing of individual nodes can simplify the development of applications that integrate data from these devices. Although consumer electronics with Internet access are becoming increasingly popular [1], a common ground that enables seamless integration of devices with applications is still lacking. Indeed, the myriad of existing protocols and standards for networked devices turns each network into isolated islands that hardly interact with each other.

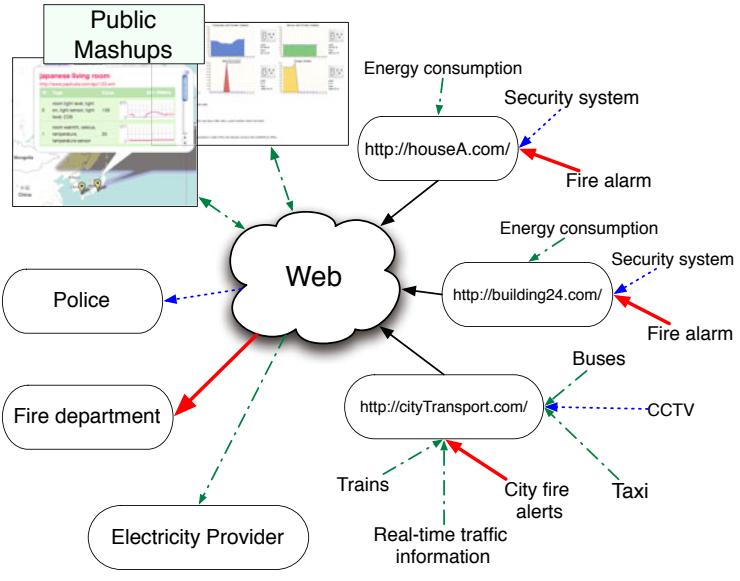


Fig. 1. Future distributed sensing applications will require a scalable infrastructure where new devices and services can be added and used with minimal effort

Because the Web is widely used, open, and easy to integrate, the Web of Things vision proposes to reuse the wide-spread Web infrastructure and standards (HTTP/XML/RSS) to connect embedded devices [2]. In this article, we extend the Web of Things paradigms to support more scalable and interoperable WSN applications by leveraging the event-driven nature of publish-subscribe systems.

A priori, HTTP seems not suited for building WSN applications because of its request/response nature. However, the recent success of Web push tools and techniques have enabled the development of event-driven applications directly over the Web. Our results show that RESTful messaging for embedded devices is a viable and scalable approach for building more open and programmable distributed sensing applications. To our knowledge, our work is the first to show the feasibility of a fully Web-based distributed sensing application that combines the recent advances in Web technologies to design an HTTP-based event-driven programming model for sensor networks.

After introducing the related work in Section 2, we survey Web messaging techniques and analyze their suitability for embedded devices in Section 3. Based on these findings, we designed RMS (Restful Messaging System), a scalable HTTP-based messaging system for distributed sensing applications which is described in Section 4. In Section 5 we evaluate the performance of RMS both with a simulation and with real devices. Finally, in Section 6 we discuss our results and their applicability for future distributed sensing, and Section 7 concludes this article.

2 Background

The Web of Things is the intersection of two fields that have been rarely associated in the past – networked embedded sensors and Web engineering. In this section we provide the required background and related work required to understand our contributions.

2.1 Networked Sensors and Actuators

Most sensor network applications share a common goal: gather, process, and store data collected by physically distributed sensors. To simplify the development and deployment of such applications, early approaches have explored the use of declarative data-centric models and query languages that consider sensor networks as a single logical unit, among which TinyDB [3]. However, such approaches are limited to heterogeneous systems and are not suited to the loosely coupled nature of the Web of Things, where new devices are added/removed at runtime.

Integration of physical things with the Web has already been proposed almost a decade ago [4,5]. However, these early projects focused mainly on embedding *information* about physical objects onto embedded Web servers and linking the objects with their virtual counterparts on the Web, which is fundamentally different from *integrating* devices into the Web as we propose here. Not only devices can be interacted with and controlled through an open Web API, but their status and functions can be searched, browsed, and used just like any other Web content. More recent projects [6] have investigated how to access embedded devices using the REST paradigm (see Section 2.2), but they mainly focused on isolated experiments and didn't address more scalable and heterogeneous systems.

The term *Sensor Web* refers to a global network of Web-connected sensors, and several projects have been proposed to build such a worldwide Sensor Web, as for example IRISnet [7], Senseweb [8], or Pachube[1]. In these projects, a unique endpoint is used to register and store data collected by many physical sensors. Such a central point of failure is against the distributed nature of the Web. Additionally, direct interaction among devices is not allowed as commands have to pass through the server. Stream Feeds [9] have proposed to extend the Web

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

feed model (RSS/ATOM) to accommodate the large size and real-time nature of sensor data streams, however beyond a promising idea, a more substantial description and evaluation with actual devices is lacking and the project seems discontinued.

In this paper, we extend the simplistic stream feed model and propose a general-purpose, distributed, and flexible publish/subscribe system that is particularly suited for the requirements of the Web of Things. The novelty of our approach lies in the fact that RMS enables to collect and use data streams from heterogeneous sensors directly over the Web. This significantly lowers the access barrier for Web developers that can now rapidly integrate real time data in their Web applications using the tools they already know – which until now required advanced knowledge in embedded systems.

2.2 RESTful Web Services

HTTP is a rather simple protocol that follows the *Representational State Transfer (REST)* architectural style [10]. REST defines a few design constraints for building Internet-scale applications that are more flexible and simpler to use. Every component of an application is an URI-addressable resource whose representation is manipulated with a uniform and fixed interface, defined by the four main HTTP verbs (GET, POST, PUT, and DELETE). Interactions are stateless, thus servers do not keep the state of applications, which improves significantly the scalability of the system. HTML is the primary resource representation which is universally understood and is a lightweight language providing hypermedia features, while XML and JSON are the preferred representation for machine readable data.

Existing middleware for sensor networks have predominantly focused on data collection and processing, therefore ad-hoc interaction with devices is difficult, if not impossible. In a RESTful application, there is no need for any special interface as the application fully blends into the Web. Interacting with it does not use any special API beyond a HTTP client to access resources and manipulate them. Because of the low access barrier, more and more Web application have been switching from SOAP to REST as basis for their API, and this observation serves as motivation to apply the same paradigm to interact with embedded devices, as we suggested in this article.

2.3 Web-Based Sensor Networks

The core idea of the Web of Things is to enable Web-based interactions with embedded devices. This requires their functionality to be accessible through URIs that can be manipulated using HTTP. For example, one could send commands to actuators (*turn on LEDs*), retrieve sensor data (*get temperature sensor reading*), or change application state (*change the sampling frequency*) directly through a RESTful API [6]. As described in [2], two solutions are possible for *Web-enabling* sensor networks: directly on devices or through a proxy.

Device-level enablement. In this case, each device runs an actual Web server that processes and serves HTTP requests directly. Using HTTP for embedded devices has been often criticized because of the high memory footprint of HTTP servers and because of the verbosity of the HTTP protocol. However, recent work has shown that embedded Web servers can run on resource-constrained devices [11], requiring as little as 8 kB of memory [12]. Additionally, as the software and operating system for embedded devices are usually event-driven, their underlying architecture lends itself well for the construction of efficient event-driven HTTP servers [12]. Such event-driven Web applications are the most desirable approach for energy-constrained devices that sleep most of the time.

Proxy-level enablement. When device-level support for HTTP is not possible or desirable, Web integration can take place on a smart proxy (referred to *gateways* hereafter) which hides the actual communication protocol used by devices behind a uniform Web interface, as proposed in [13]. Although gateways are not required for devices that support HTTP directly, they can nevertheless augment the functionality of single devices and improve the overall performance of sensing networks. Because gateways are much less constrained than sensor nodes, they can serve as distributed master nodes that can manage sensor networks. In addition, gateways can be delegated tasks that would be too expensive in terms of CPU and energy to be run on resource-constrained embedded devices. For example, caching data from the sensors for concurrent read accesses, buffer incoming request for devices, perform aggregation of data and local mashups, or manage security policies and access control to devices, could be all taken care of by gateways.

3 Web-Based Messaging

As the size of distributed sensing applications increase, so does the necessity to integrate disparate hardware and software platforms. Interfacing different messaging protocols is a complex procedure, and bridging different middlewares is prone to severely hinder the performance and scalability of such a future distributed systems. *Publish-subscribe systems* (pub/sub) are commonly used in distributed computing and large enterprise applications, because they allow decoupling data producers and consumers. Loose coupling allows more flexible and scalable systems where new entities can be easily added or removed. Highly scalable and efficient messaging protocols with various features have been proposed, however none of them directly integrates with the Web, as an additional protocol must be implemented on top of HTTP. XMPP is an XML-based messaging protocol widely used for chat servers. It is based on a decentralized network of servers that provide a multi-hop routing delivering messages from one client to the other. Because it is based on XML, it remains quite heavy for lightweight messaging with resource-constrained devices.

Only recently sensor networks have began to explore pub/sub messaging for building complex and interoperable applications that scale, as pub/sub shields applications from the underlying complexities of the WSN and provides a simple

– yet powerful – interface to interact with a WSN. MQTT-S [14] is a messaging protocol designed for tiny devices. TinyDDS [15] is another publish/subscribe middleware that enables interoperability between WSNs.

Web Syndication. A rudimentary form of Web messaging is *content syndication*. Feed formats such as RSS, or the more well-defined *Atom*, have become popular formats for exchanging machine-readable data over the Web. Atom offers a convenient metaphor to deal with time-ordered collections of entities, therefore would be particularly suited for storage, query, and retrieval of stored sensor data. However, Atom is limited by the pull-based mode, which cannot meet the requirements of event-driven applications. A push-based pub/sub protocol for the Web would simplify significantly the integration between WSNs and applications, however, such a messaging protocol that fully integrates with the Web has not yet been proposed.

Comet. *Comet* (also called HTTP streaming or server push) has become an increasingly popular technique to implement server side eventing for Web applications that circumvent the limitations of the traditional HTTP polling. Comet enables a Web server to push data back to the browser without the client requesting it explicitly by keeping the TCP/IP connection open after an initial response has been sent to the client. Since Web browsers (and HTTP) were not designed to support server-initiated notifications, Comet is a hack implemented through several specification loopholes. Comet servers and clients frequently use named channels or topics which are useful when different objects want to send data to a number of other interested parties. The main advantage of Comet is that standard Web clients (in particular browsers) can receive notifications pushed from servers in near real-time, even when behind firewalls or NAT.

Web Hooks. Web hooks are another solution for HTTP eventing that enable users to receive events and data in real time from applications through user-defined callbacks over HTTP. Once an event occurs, the application will POST data to the callback URLs specified by the users at subscription time. This pattern has been used by the PayPal service which allows you to specify a URL (on your own online commerce site) that will be triggered by PayPal once a payment has been accepted. Web hooks enables Web applications to synchronize data with other applications, but also to process, filter, or aggregate data from different sources and to notify people via email, IRC, Jabber, or Twitter. However, clients that want to receive notifications must also run a Web server where notifications will be posted. Web hooks are an elegant, clean, and RESTful solution for bi-directional Web eventing, unfortunately cannot be used when clients are behind NAT or a firewall, as they do not have a public network address.

The growing need for push-based communication on the Web is further supported by the introduction of *Web Sockets* and server-sent events in the HTML 5 specification, and also by the browser-side Web server embedded in the Opera Unite browser. An interesting recent project is RestMS[2], which offers a

[2] `http://www.restms.org`

RESTful interface to the Advanced Message Queuing Protocol (AMQP) and defines the behavior of a set of feed, join, and pipe types that provide an AMQP-interoperable messaging model. PubSubHubbub[3] (PuSH) is a lightweight and open server-to-server publish/subscribe protocol based on Web hooks as an extension to Atom and RSS. Servers can get near-instant notifications when a topic (feed URL) they're interested in is updated. However, these solutions were not designed for embedded devices as they are rather verbose. Based on these observations, we have designed RMS, a fully Web-based publish/subscribe mechanism designed to meet the requirements of distributed Web-based sensing applications, as will be described in the next section.

4 RMS: RESTful Messaging for Devices

Nowadays, integration of different WSNs is done in a fairly rigid manner: devices are tightly coupled to custom bridges that connect them to the outside world because of the lack of a common, widely adopted application protocol for sensor networks. In contrast, a uniform Web-based messaging would allow devices, gateways, brokers, and applications to transparently interact with each other in an ad-hoc manner. Devices can directly exchange information transparently with each other and with other Web resources thanks to the loose coupling of REST. Furthermore, as gateways are optional, they can easily be bypassed in case they fail, which increases the overall robustness of the whole system.

Two main classes of WSN applications exist: event-driven (where notifications are sent sporadically when an event occurs) and stream-based (sensor data is collected periodically and sent to a sink to be processed and/or stored). To support such interaction models, we developed the RESTful Messaging System (RMS), which is a lightweight pub/sub messaging suited for devices. In essence, our system is similar and directly mappable to RestMS and PubSubHubbub (PuSH). However, as we target embedded devices, we tried to keep it as simple as possible. Rather than creating a custom protocol on top of HTTP (such as XMPP or PuSH), we implement the core functionality of a pub/sub system solely using RESTful design patterns. More elaborate pub/sub protocols such as XMPP have a higher barrier of adoption and are somewhat complex for embedded devices. Also, just like SOAP-based Web services, packets are opaque therefore cannot be interpreted and acted upon by 3^{rd} party proxies.

The gateway offers a RESTful API to use the eventing system and provides the following resources to manage interactions:

- `/rms/channels` every sub-resource represents a hierarchical channel where entities can post data to. For example, `/rms/channels/ethz/ifw/floor/d/49.2` identify the channel related to the office No. 49.2 of the D floor, in our building (called ifw) at our school (ETH Zurich).
- `/rms/subscriptions` contains each subscription of entities to individual channels.

[3] `http://pubsubhubbub.googlecode.com`

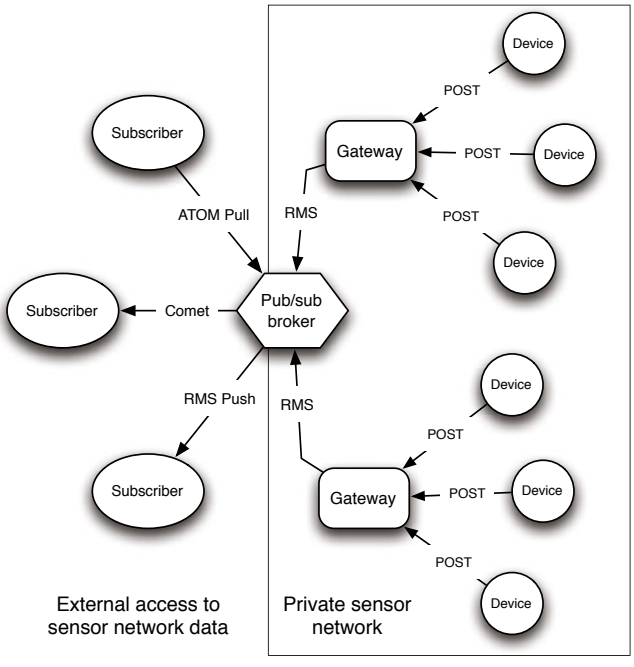


Fig. 2. The general model of Web-based messaging. Devices can POST their data on the gateway, using standard HTTP POST, AtomPub, or other proprietary protocols. Gateways will then forward it to another gateway or to a powerful broker using RMS. External users can fetch the data using the most appropriate method for their needs.

- `/rms/channels/*/publishers` contains all entities that are publishing data on the channel `*`.
- `/rms/channels/*/subscribers` contains all entities that are subscribed to events on the channel `*`.

A subscriber that wants to receive notifications about a channel, creates a new subscription by POSTing the following HTTP request to the gateway:

```
POST /rms/ethz/ifw/floor/d/49.2/subscriptions
Host: gateway_ip
Content-type: application/x-www-form-urlencoded
Content: cb-url=http://sub_ip/callback_url
```

Each new message posted on this channel will be POSTed back to all subscribers using the Web hook pattern to the URL they specified with the `cb-url` parameter. Any entity can publish data to the gateway by POSTing the following request on the gateway:

```
POST /rms/channels/ethz/ifw/floor/d/49.2
Host: gateway_ip
Content-type: application/x-www-form-urlencoded
Content: pub-url=http://device1&temperature=21
```

This device has never been registered with the gateway, so it includes its URL in the posted parameters, providing the gateway with a possibility to automatically scan it for semantic description of its capabilities and get required information. The message is posted directly to the channel `/ethz/ifw/d/49.2` without any need to previously create it. Other parameters are treated as tags (in this case `temperature`). In addition to specifying the channel or topic, publishers can also annotate messages with free text tags. Similarly, subscribers can easily filter out messages that contain or not specific tags.

Comet implementations such as CometD[4] treat topics as paths to support hierarchical relationships between topics. This is quite useful for example for permission models or aggregated data, where a subscriber to a parent topic can receive all messages from child topics. Paths can directly map with URI (e.g., `ethz/ifw/floor_d/49.2`), which can be directly integrated with the Web. Our gateway is implemented in OSGi[5] and supports Web hooks and Comet-based eventing. Comet data is accessible by replacing `rms/` by `cometd/` in the URL above. Therefore, users can can see in real time messages on a particular channel by pointing their Web browser to following URL:
`http://gateway_ip/cometd/channels/{channel_path}`

Because our gateway uses the internal eventing capabilities offered by OSGi as abstract model for notifications, additional notification protocols can be easily added (SMS, Twitter, e-mail, etc).

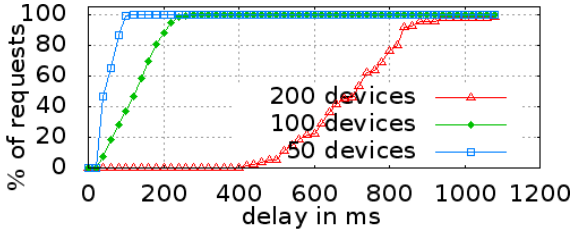
5 Evaluation

To evaluate the performance of RESTful messaging under extreme load conditions, we have conducted an experiment using simulated devices and subscribers connected through an RMS broker running a desktop computer (1.1 GHz, 2GB Ram, Gigabit Ethernet and Gentoo Linux). The HTTP clients (subscribers) were simulated on another machine (2x2.13 GHz, 8GB Ram, Gigabit Ethernet, Gentoo Linux). In the second part of this section, we describe a second experiment done to measure the performance of RMS in real-world conditions through an actual sensing system deployed using real sensor nodes.

Synthetic Load Simulation. First, we simulate many devices attached to the gateway, each one generating an event at a random interval between 1 and 5 seconds. Three runs have been performed with respectively 50 devices, 100 devices, and 200 devices attached. The time required to receive, process, and deliver all the events to one client has been measured, and results are shown in Figure 3.

Second, we evaluate the scalability of RMS with respect to concurrent subscribers for the same event triggered by a device. A test client started an event sink to receive events on respectively 50, 100, and 200 different ports, and for each port an event subscription was posted to the gateway. The gateway generated

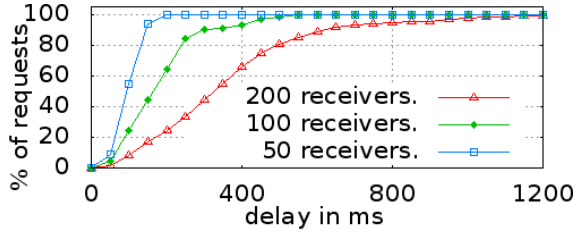
[4] `http://cometd.org/`
[5] `http://www.osgi.org`



	50 d.	100 d.	200 d.
Min	29ms	30ms	418ms
Max	127ms	284ms	1087ms
RT 50%	41ms	124ms	641ms
RT 80%	71ms	174ms	733ms
Mean	52ms	125ms	699ms

Fig. 3. *Many devices:* Response time to deliver events to a client with 50 devices (box), 100 devices (circle), and 200 devices (triangle) attached

artificial events (containing the generation time) that were delivered to all the subscribed ports. The test client measured the arrival time and from that computed the delay for each arriving event, and results are shown in Figure 4.



	50 r.	100 r.	200 r.
Min	34ms	35ms	34ms
Max	396ms	553ms	1347s
RT 50%	94ms	165ms	326ms
RT 80%	120ms	236ms	479ms
Mean	97ms	180ms	360ms

Fig. 4. *Many receivers:* Event delivery times measured for 50 subscribers (box), 100 subscribers (circle), 200 subscribers (triangle)

Third, we want to evaluate the performance of the classic request/response pattern under heavy load where many devices are attached to a gateway and a varying number of clients access simultaneously devices through HTTP. The test client started several concurrent threads that accessed the gateway and its devices randomly to simulate real clients accessing the gateway. In the first case, 4000 devices, three test runs with 100 clients, 50 clients and 25 clients, and results are shown in Table 1. In the second test 1000 virtual devices are attached to the gateway and three test runs have been performed with respectively 375 clients, 750 clients, and 1500 clients, and results are shown in Tables 2.

For a *moderate* number of devices (50 and 100) the gateway is able to dispatch RMS messages efficiently, with respectively 52 ms and 125 ms average delivery time. Doubling the number of devices results in a approximate doubling of the latency (2.4x). However, with 200 devices the performance drops significantly (5.59x slower than with 100). The third test shows that gateways can handle over 750 concurrent read requests per second from 1000 devices and 80% of these requests will be answered within 224 ms.

Table 1. Response time in milliseconds [ms] to deliver a request from a gateway with 4000 simulated devices attached to resp. 25 clients, 50 clients, and 100 clients

	25 clients	50 clients	100 clients
Min	3	3	3
Max	326	253	1248
RT 50%	16	40	56
RT 80%	44	111	172
Mean	63	67	136

Table 2. Response time in milliseconds [ms] to deliver a request from a gateway with 1000 simulated devices attached and resp. 375 clients, 750 clients, and 1500 clients

	375 clients	750 clients	1500 clients
Min	2	2	2
Max	9250	21054	21261
RT 50%	29	48	124
RT 80%	135	224	3059
Mean	614	860	1686

Real deployment. In this second experiment, we test the performance of RMS to transmit data from a real sensor network (devices used were TMotes running TinyOS), where the Web-enablement is done at the gateway level. Devices broadcast an event every second that are received at a time t_1 by a base station (A) attached to a WSN gateway running an RMS broker. Each event from the sensor network is then posted using RMS (Web hooks) to subscribers (in this case a sink laptop on the same LAN). Each event from the WSN is also caught using a spy base station (B) which starts a timer (also at time t_1). The timer is stopped when the corresponding notification is received from the WSN gateway though RMS at a time t_2. The time difference $t_2 - t_1$ corresponds to the time required to create and dispatch the RMS message from the WSN gateway and receive it on the sink.

Similarly to our results in the first experiment with simulated devices, a fully HTTP-based messaging system can transmit data from real sensor nodes with reasonable delivery times, as shown in Fig. 6. In all the cases, events needed less than 60 ms to be pushed from the publisher to the subscriber. This latency can be considered tolerable even for emergency and time-critical deployments. Obviously, the delay would be significantly larger for subscribers not on the same LAN as the RMS broker, or if the sensor network used a multi-hop topology.

WSN deployments for environmental monitoring rarely have more than 50 devices per gateway and sampling rate is rarely higher than one sample per second. Based on our results, we conclude that a fully HTTP-based solution is largely sufficient for collecting data in typical sensor network scenarios even when sub-second latency with hundreds of concurrent requests is needed.

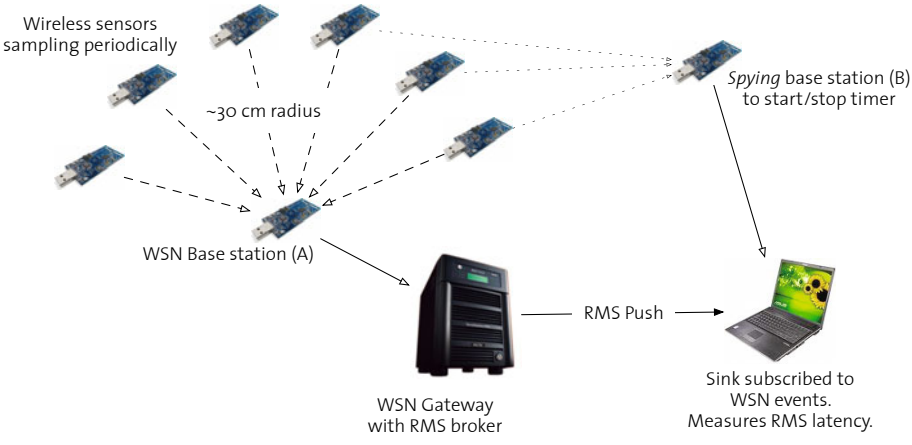


Fig. 5. Experimental setup to measure the latency of RMS. TMotes broadcast periodically numbered events caught by base station (A). The sink subscribes to all WSN events and each time it catches an event from the WSN (through its spy base station B) it starts a timer that will be stopped when it receives a RMS notification about the same event from the WSN Gateway.

6 Discussion

Nowadays, Web applications routinely integrate data from multiple sources such as RSS/Atom feeds, blogs, maps, etc. As the number of networked sensing devices will increase, so will the incentives to share and integrate the data they produce with Web applications. So far, only few projects have explored the Web of Things beyond linking physical objects with Web pages. Different middlewares have been proposed for the integration of heterogeneous sensors with applications, however they would introduce a strong coupling between components that is inappropriate with the ad-hoc nature of dynamic sensor networks, as all devices in the global network should settle on a single protocol, which is unlikely to happen.

The work presented here differs from middleware-based approaches in that we leverage *only* the ubiquitous modern Web architecture as abstraction layer for the peculiarities of various hardware and software platforms available for sensor networks. Enabling RESTful access with embedded devices significantly lower the access barrier to consume real-time data from the physical world. Simplified access to WSN data over the Web fosters the development of physical Web applications, as devices would have a Web API just like other Web resources. Programming with them could be done using highly popular and relatively simple languages such as JavaScript, DHTML, PHP, or even simple and visual mashup editors such as Yahoo Pipes. Web integration would be maximized as devices could be searched for, browsed, linked to, and used just like other Web content.

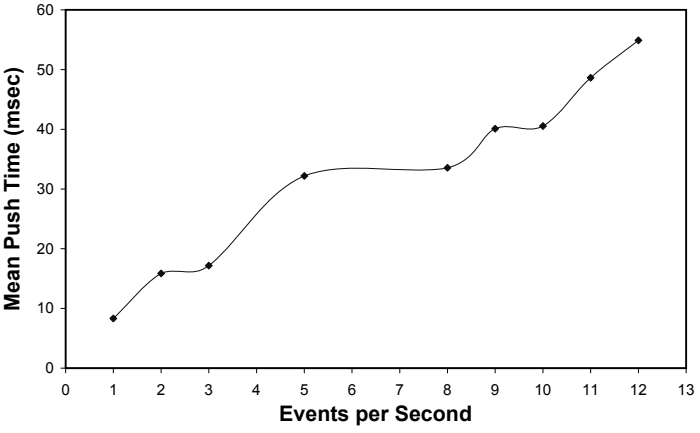


Fig. 6. Average delivery time for events generated on sensor nodes using a 1-hop subscriptions. Each devices generates one message per second, and increasing the number of devices also increases messages per second (here we use between 1 and 12 devices sending concurrent messages). Variability (not shown here) is mainly due to the physical nature of radio communication, which is hard to control.

Publishers and subscribers are loosely coupled to each other through the use of RMS. Unlike traditional pub/sub systems, a fully Web-based pub/sub system further decouples participants because only an HTTP client is needed and no additional protocol need to be implemented. Web standards maximize interoperability and loose coupling between components, which are desirable properties for scalable distributed applications. Along with the event-driven interaction model of RMS, these properties match well the dynamic nature of WSNs.

Future sensing applications will require enabling easy and timely access to physical data, and the solution we propose lowers the barriers to access WSN data while being fully integrated with the Web. As more embedded devices will be present in our daily environments, one could install gateways on home (or enterprise) routers where the number of attached devices is moderate (e.g. a WiFi, mobile phone, network attached storage...). For these appliances the performance of the system should therefore be sufficient.

In contrast to other optimized messaging systems, RMS suffers from the overhead of the HTTP protocol. Although our results are encouraging, optimizations and enhancements of the RMS broker implementation could increase the performance and throughput of the system. As RMS mainly consists of an HTTP server and client, it scales with the hardware. That is, running the broker on a more powerful machine would allow to attach more devices and serve more clients simultaneously.

7 Conclusion

As more embedded devices will be connected to the Internet, efficient solutions will be needed to collect, process, and store the data they will generate. In this

article, we have described how to reuse the ubiquitous Web standards to build a scalable infrastructure for connecting embedded devices, the Web of Things. Our solution not only supports the request-response model of HTTP, but also leverages the recent development in the real-time Web to offer an efficient Web-based publish/subscribe system.

Our main contribution is to provide quantitative results to support the idea that using HTTP as application protocol for distributed sensing applications is not only a feasible, but also a desirable solution for integrating physical devices with applications. This is especially true when integration primes over raw performance in terms of latency and throughput, as the advantages brought by using Web technologies at the device-level outweighs the loss in performance. Our results show that HTTP-based messaging can support hundreds of concurrent users accessing hundreds of devices simultaneously with a sub-second latency, which is sufficient for most monitoring applications that use sensor networks. We have pointed out how the loss in performance in comparison to traditional messaging systems could be compensated by using gateways to improve the performance, scalability, and functionality of the applications running within sensor networks.

References

1. Dunkels, A., Vasseur, J.: IP for Smart Objects Alliance. Internet Protocol for Smart Objects (IPSO) Alliance White paper (September 2008)
2. Guinard, D., Trifa, V., Wilde, E.: Architecting a mashable open world wide web of things. Technical Report 663, Institute for Pervasive Computing, ETH Zurich (February 2010)
3. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: TinyDB: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst. 30(1), 122–173 (2005)
4. 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, pp. 23–31. ACM, New York (2000)
5. 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), 365–376 (2002)
6. Luckenbach, T., Gober, P., Arbanowski, S., Kotsopoulos, A., Kim, K.: TinyREST - a protocol for integrating sensor networks into the internet. In: Proc. of REALWSN (2005)
7. Gibbons, P., Karp, B., Ke, Y., Nath, S., Seshan, S.: IrisNet: an architecture for a worldwide sensor Web. IEEE Pervasive Computing 2(4), 22–33 (2003)
8. Kansal, A., Nath, S., Liu, J., Zhao, F.: SenseWeb: an infrastructure for shared sensing. IEEE Multimedia 14(4), 8–13 (2007)
9. Dickerson, R., Lu, J., Lu, J., Whitehouse, K.: Stream Feeds: an Abstraction for the World Wide Sensor Web. In: Proceeding of the 1st Internet of Things Conference (IOT), Zurich, Switzerland (2008)

10. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
11. Yazar, D., Dunkels, A.: Efficient Application Integration in IP-Based Sensor Network. In: Proc. of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys), at SenSys '09 (2009)
12. Duquennoy, S., Grimaud, G., Vandewalle, J.J.: Consistency and scalability in event notification for embedded web applications. In: 11th IEEE International Symposium on Web Systems Evolution (WSE'09), Edmonton, Canada, Edmonton, Canada (September 2009)
13. Trifa, V., Wieland, S., Guinard, D., Bohnert, T.M.: Design and implementation of a gateway for web-based interaction and management of embedded devices. In: Proceedings of the 2nd International Workshop on Sensor Network Engineering (IWSNE'09), Marina del Rey, CA, USA (June 2009)
14. Hunkeler, U., Truong, H.L., Stanford-Clark, A.: MQTT-S - A publish/subscribe protocol for Wireless Sensor Networks. In: Proceedings of the Third International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE 2008), Bangalore, India, pp. 791–798 (January 2008)
15. Boonma, P., Suzuki, J.: Middleware support for pluggable non-functional properties in wireless sensor networks. In: SERVICES '08: Proceedings of the 2008 IEEE Congress on Services - Part I, Washington, DC, USA, pp. 360–367. IEEE Computer Society, Los Alamitos (2008)