Smews: Smart and Mobile Embedded Web Server

Simon Duquennoy
IRCICA/LIFL, CNRS UMR 8022,
INRIA Lille - Nord Europe,
Univ. Lille 1, France
simon.duquennoy@lifl.fr

Gilles Grimaud
IRCICA/LIFL, CNRS UMR 8022,
INRIA Lille - Nord Europe,
Univ. Lille 1, France
gilles.grimaud@lifl.fr

Jean-Jacques Vandewalle
Gemalto Technology & Innovations,
France
jean-jacques.vandewalle@gemalto.com

***Abstract*—In this paper, we show that Web protocols and technologies are good candidates to design the *Internet of Things*, through a user-centric architecture (the user simply has to use a standard Web browser). We detail how this *Web of Things* can handle typical embedded devices interaction needs. We discuss the technical feasibility of embedded Web servers, and, thanks to an analysis of the Web protocols, we propose new cross-layer solutions for efficient tiny embedded Web servers design. The problem of event notification for Web applications is also discussed. We finally present a prototype – named Smews – as a *proof of concept* of the *Web of Things*. Smews implements our proposals and has been embedded in tiny devices (smart cards, sensors and other embedded devices), with a requirement of only 200 bytes of volatile memory and 7 kilo-bytes of code. We show that it is significantly faster than other state of the art solutions. We made Smews source code publically available under an open-source license.**

I. Introduction

Today, more and more devices are ubiquitously running around us. Traditional communication schemes use of heterogeneous protocols, softwares and user interfaces, making hard devices interaction. Users would like to easily access public devices, whatever their implementation choices.

Numerous publications call this global devices interconnection the *Internet of Things* (IoT). This does not refer to any technology nor any network structure, but only to the idea of interconnecting objects as well as we interconnect computers with the Internet. IoT relates to heterogeneous objects, such as home automation, household electrical appliance,smart cards, sensors, routers, ... The complexity of such devices and the fullness of the services they provide is exponentially growing, making the IoT a hot research topic.

The first well-known standard for global devices interaction is the Universal Plug and Play (UPnP) [1]. UPnP is used for several services such as discovery, description, control, event notification, presentation, audio and video broadcast. Nevertheless, UPnP has several drawbacks (described more precisely in Section II-B), it does not provide any authentication mechanisms, and claims a totally flat network usage. In numerous papers [2], [3], we can read that UPnP is too heavy to be embedded in some devices, because it uses a huge number of different protocols.

We claim that IoT makes only sense using a user-centric architecture, where actions done by devices are always initiated by a user. A user can access its own set of devices, so forming a *Personal Area Network* (PAN).

In this paper, we propose solutions for IoT design. In Section II, we present a state of the art of existing protocols and solutions for IoT implementation. We propose a new user-centric architecture named WoT in Section III, and detail WoT design in Section IV. The Web of Things feasibility is discussed in Section V. In Section VI, we describe a proof-of-concept prototype for WoT implementation. We compare its performances with other state of the art solutions. We finally conclude in Section VII.

II. State of the Art

In this section we define the IoT and we describe existing standards for IoT protocols.

A. Internet of Things definition

Numerous works have been done to define the Internet of Things and to associate technologies and network architectures to this – still abstract – concept. The exponentially growing amount of devices around us require efficient interaction solutions, to allow anyone to access easily any object. The notion of Personal Area Network (PAN) is often used referring to IoT networks. A PAN is a volatile network that includes every object a pearson should interact with. The main concerns of PANs for the IoT are defined in [4], [5]. They can be summarized as follows:

- A PAN is a volatile and spatially local network, which includes the devices a user can physically access
- Objects discovery and addressing must handled by the PAN protocols
- Devices must be able to interact with the whole PAN and possibly with external networks
- Operations and interactions done into a PAN must be secured

It must be noticed that the nodes of the PAN are mainly hardware constrained devices. They often have a few kilo-bytes of volatile and persistent memory, a CPU frequency of a few MHz, hard autonomy constraints, and low-throughput physical links (*e.g.*, Bluetooth, USB, ZigBee, ...).

B. UPnP: the widespread standard for PANs

UPnP [1] is a set of protocols promulgated by the UPnP forum. It is today the most spread solution for PAN implementation. UPnP makes use of a lot of (standardized or not) protocols and technologies, such as UDP, TCP, HTTP, HTTPU,

SOAP, WSDL, ... A UPnP network is flat, *i.e.*, every objects included into a PAN have the same role and rights.

UPnP has nevertheless several drawbacks. First, no authentication protocol is proposed for UPnP. This is a critical security issue, allowing any device to configure any other node of the PAN, without any user control. Secondly, UPnP makes use of some unstandardized protocols likes HTTPU (HTTP over UDP). Finally, UPnP is quite heavy because it uses a lot of protocols that involves heavy processing (SOAP, WSDL, ...). This makes it unusable into very constrained hardware. In [3], a proxy is proposed between the PAN and sensor nodes, because sensors are unable to embed the whole UPnP stack. [2] proposes a solution, where sensors workload is deported to dedicated terminals, used as UPnP proxies.

C. Alternatives to UPnP

The JXTA technology [6] is a solution for peer-to-peer applications design, allowing to interconnect heterogeneous devices into a same network. JXTA-C [7] is a C implementation of JXTA, making it usable into contrained hardwares. Nevertheless, JXTA are not built on standardized protocols.

DPWS (Devices Profile for Web Services) is the official successor for UPnP. Its main objective is to allow secured Web services usage. The set of protocols it uses is similar to UPnP one, making it also hard to embed into tiny devices.

It is shown in [8] that an intersting solution for global devices interconnection consists in using embedded Web severs (EWS). [9], [10] show that devices with a few kB of RAM and of EEPROM are able to handle an embedded Web server.

III. Web of Things: a user-centric architecture

Fundamentally, the notion of PAN is user-centric, *i.e.*, it is built around a user. It only makes sense relatively to a user. That is why we propose to design a IoT solution in the context of user-centric networks.

A. Web of Things description

After years of hindsight, we observe that the success of the Internet is due to very simple applications usage: mainly the Web and e-mails. Nowadays, Web technologies are impressively widespread, as well as anyone is able to access Web servers from a personal computer, PDA or cell phone. Furthermore, it has been shown (see Section II-C) that it is technically possible to embed Web servers into tiny devices. We call *Web of Things* (WoT) the concept of implementing PANs thanks to Web servers installed into every nodes.

The WoT consists in using Web protocols and technologies into a client-centric PAN architecture. The computer, PDA or cell phone of the user run a client (a standard Web browser) while every node into the PAN run a Web server. The client is able to access any server included into its PAN. An object can possibly be accessed by multiple users: in such situation, it is included into several PANs.

This approach of WoT has a big advantage on UPnP: the user is the only possible initiator of actions into its PAN. This forbids an object to initiate an action with an other object, or with external networks.

B. From the World Wide Web to the Web of Things

One of the obvious advantage of the WoT is that it uses existing and widespread technologies: TCP/IP, HTTP, Web applications. This ensures a great accessibility as well as ease of development, via numerous existing and well-know design frameworks. Several fundamental differences between the World Wide Web and the Web of Things are nevertheless very sensible.

In the WWW, servers use very powerfull hardware, they are able to handle thousand of simultaneous connections and HTTP requests, involving heavy server-side processing such as pages generation. In the WoT, Web servers have huge hardware constraints. Comparatively the client is extremely powerfull (even if it is a PDA or a cell phone). The usage of Web technologies like AJAX or Comet (more precisely described in Section IV) are well-designed to deport Web application processing from the Web server to the Web browser.

The second difference between the WWW and the WoT is the network structure: the WWW is server-centric while the WoT is user-centric. Web servers on the WWW are accessible from every client, and WWW applications often focus on the management of a set of users. On the WoT, a server is accessible by a user only if it is into its PAN. The network structure of the WWW is quite static, while the WoT network structure is volatile.

IV. Designing the WoT

This section is about the WoT design in sight of devices applications needs. Because WoT applications are mainly typical Web applications, they can be designed using existing Web frameworks.

A. Web applications for constrained servers

For efficient embedded Web applications design, some design practices have to be adapted from classical WWW ones. Indeed, such applications are embedded into very constrained hardware (see Section II-A), while Web browser are still executed in relatively powerfull machines.

Web applications need to generate dynamically Web pages and contents. The contents can be generated in the server-side (using *e.g.*, SSI technologies, PhP, JSP, ...) or in the client-side (using mainly JavaScript).

The AJAX[11] model allows to design highly interactive applications with an efficient task repartition between the client and the server. The behavior of an AJAX Web application can be separated into two phases:

1) The loading phase. The client starts by collecting several static files, containing style (CSS), contents (HTML), and applicative code (JavaScript).
2) The running phase. The client (*i.e.*, the browser) executes the applicative code downloaded in the first phase, and it interacts with the server by sending asynchronous requests. Server responses are often small generated contents; they are interpreted by the client and integrated into the Web page.

This methodology reduces Web traffic because dynamically generated contents are only semantic information interpreted by the client. Formatting rules are loaded only once in the initialization phase, thus factorizing information and reducing redundancies. AJAX applications allow a workload deportation, from the Web server to the Web browser: the server sends smaller generated data while the client runs behavioral code. This is particularly interesting in our situation, where the Web server has less resources than the Web client. AJAX strengths make it incontrovertible for Web of Things applications.

B. Event notification

A common need for IoT devices consists in event notification (*e.g.*, for home automation, sensors, ...). In a first sight, HTTP seems unsuitable for such behavior. Nevertheless, a new model, named Comet [12] appeared these last years, allowing a Web application to push data from the server to the client. A Comet protocol specification exist named Bayeux [13].

A comparison of AJAX pull and push (Comet) methods is presented in [14]. It shows that, while Comet is better in terms of data consistency and traffic workload, it suffers from scalability issues in term of processing. Server-side comet management is hard for two main reasons:

- Classical Web contents generator engine (*e.g.*, for Servlets, ASP, JSP or PhP), do not allow a request handler to idle efficiently waiting for an event (thus giving back the hand to the engine). That is why dedicated frameworks and engines are developed to provide Comet support [15], [16].
- The server has to store information about each client listening for an event. Each TCP connections is kept alive until an event occurs, implying a huge memory consumption.

The scalability issue has been observed for hundreds of simultaneous connections, what should rarely occur in the context of the WoT. Morehover, traffic and responsiveness improvement is very important in the context of event notification, making it well-designed for such an usage.

C. PAN management

Before acessing a device via its embedded Web server, this last must have been detected and included into the PAN. The insertion of objects into the PAN can be done by a DHCP or a similar dedicated protocol, allowing automated IP (and possibly URI) attribution.

D. Interaction between devices

The concept of IoT allows interaction between devices. We claim that each interaction must be initiated by the central user of the PAN. Such a care allows to know who is responsible of every action. We take a simple example: a user need to print the photos stored into its camera thanks to a printer. Two solutions are available:

1) The user downloads all the photos from the camera. He secondly uploads them from its computer to the printer, and launches the print task.
2) The user gives the URI of the printer to the camera (via a dedicated input box in the Web application of the camera), and asks it to launch the print.

The first solution is very simple and easy to implement: the two Web applications only provide a download or upload page. This solution nevertheless bridles device applications because they can not explicitly interact. Furthermore, this involves several user manipulations, and uses the client computer as a data buffer.

The second solution enables a real Internet of Things, allowing devices to interact with other nodes of the PAN or with an external network, like the Internet. This solution allows any EWS to access to an other device, as soon as it knows its URI (furnished by a user, or an other EWS). By using *mashup* (building Web pages from both local and external resources), this allows very rich embedded Web applications development. As an example, the Web pages served by the camera can furnish informations about the printer ink level.

The authentication of the interaction initiator can be done thanks to a ticket mechanism. In our example, the user first gets a ticket to the printer. He then provides this ticket to the camera (by including it into the camera URI). Such a mechanism disallows devices to initiate interaction without a user to be explicitly responsible of the act.

E. Interaction with a PAN from an external network

In some situations, it should be interesting to access distant objects from an external network like the Internet. We introduce the notion of *Virtual PAN* (VPAN) as a PAN built around an access point instead of a user. As an example, a home can have a VPAN access point, directly connected to numerous objects in the home. A user can access its home personal devices through such a access point, since this last is accessible from the Internet.

V. Handling Web traffics in constrained hardware

Most of the software designed for embedded devices use event-driven approaches to fit with highly constrained hardware. Indeed, threaded models waste a lot of memory. Event-driven approaches are extremely efficient to implement stateless behaviors. Stateful behaviors are a bit more complex to implement, requiring often multiple state storage and management. Both HTTP and IP are stateless protocols, while TCP is statefull (notions of connection, sequence numbers, acknowledgments).

In this section, we identify the critical parts of the HTTP/TCP/IP protocol stacks, regarding their requirements in terms of memory and traffic.

A. HTTP: a half duplex protocol over TCP

TCP (defined in [17]) can be used for any kind of applications, allowing bidirectional reliable communications. In TCP, data can be sent asynchronously by the two hosts. TCP allows data piggybacking, *i.e.*, sending a segment containing both data

and acknowledgment. This makes TCP void acknowledgments less frequent.

In fact, when using HTTP, TCP has a particularly simple and predictable behavior: the client sends a request then it waits for a response from the server. At the HTTP level, only one of the two hosts is sending data at a time. As a consequence, on a given TCP connection, while the client or the server is sending data (resp. a request or a response), it does not receive anything else than TCP void acknowledgments (data are rarely piggybacked).

B. Impact of the TCP MSS

When establishing a TCP connection, a Maximum Segment Size (MSS) is negotiated between the two hosts. The minimal legal MSS is of 200 bytes. The most commonly used MSS is of 1460 bytes because this size fits well with ethernet packet size.

Using large TCP MSS allows to have better performances, because TCP and IP use mainly fixed size headers. This is a stateless property, but it require to manage large packets, which is an issue on memory-constrained systems.

C. Supporting HTTP keep-alive

HTTP has been designed to run over TCP. Since HTTP 1.1 [18], a keep-alive option encourage consecutive HTTP requests to use a single TCP connections. This allows to avoid numerous connections establishments and closing, which cost grows with the link latency. Indeed, connection establishment and closing involve respectively a 3-ways and a 4-ways handshakes.

The support of HTTP keep-alive require to manage multiple TCP connections simultaneously. It is a stateful property that is hard to handle in memory-lightweight event-driven systems.

D. Impact of the TCP delayed acknowledgments

TCP delayed acknowledgments is a policy used to reduce the amount of TCP traffic caused by void acknowledgments. It is implemented by most of the desktop computers TCP/IP stack (both Windows and MacOS stacks). A TCP host that implements TCP delayed ACKs only acknowledges a segment (i) 200ms after having received it or (ii) when a second segment is received.

Embedded TCP/IP stacks often don't support more than one in-flight TCP segment. In fact, when several segments are unacknowledged, the sender has to keep them into memory to be able to retransmit it in case of a packet loss. Having only one in-flight segment allows very lightweight and stateless TCP implementations, but it interacts badly with the delayed ACKs policy: in such situation, the 200ms delay will always be triggered, limiting the sending rate to 5 packets per seconds.

VI. Prototype architecture and experimentations

Starting from our study about the Web application needs (see Section IV) and our cross-layer analysis of the Web protocols (see Section V), we designed a new embedded Web sever, named *Smews*. In this section, we describe its novel Web application framework, architecture and, thanks to an implementation, we present performance measurements compared to existing solutions.

A. Web application framework

As said in Section IV, Web applications make use of dynamic contents generation. The AJAX model is based on both server-side and client-side contents generation. We propose a model where a directory is a Web applications, containing:

- Static files (of any type: html, js, css, jpg, ...)
- Server-side generators
  - Generators (native functions)
  - OgO files (PhP-like) with native code
  - Comet generators (native functions)

Thanks to a dedicated tool, all these contents are pre-processed at compile-time in order to build the final binary files, containing the Web server and the Web applications. This pre-processing also allows us to do numerous optimizations on the contents to serve (*e.g.*, checksum pre-calculations, TCP and IP headers parts pre-calculation, ...). Figure 1 shows how files are processed in our tool-chain.

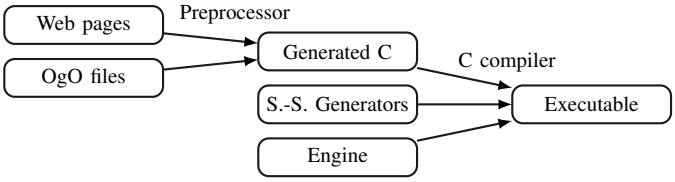

Figure 1. Smews compilation tool-chain

B. Proposed architecture

We propose to use an event-driven architecture with the granularity of IP packets. Several simultaneous connections can be handled, but never more than one packet is processed at a time. We propose a very particular buffers management for packets reception and sending.

1) Packet reception: Our event-driven approach allows to use a single and shared buffer for packets reception. This buffer can be of any size (possibly smaller than the packet sizes), while incoming data are processed sufficiently rapidly.

2) Packet sending: Unlike general purpose TCP/IP stack, an embedded Web server is ensured to always process HTTP traffic. Most of the packets sent by a Web server in such context are HTTP responses. By sharing information between the Web application and the TCP/IP stack, we are able to enhance the stack performances.

When sending a segment, a TCP stack has to keep it into memory in case of future retransmission. In our case, when it is possible, we discard every sent packet before receiving its acknowledgment. If a retransmission is needed, we retrieve again the data to send. This policy can only be applied to static Web contents (files) and to idempotent (*i.e.*, deterministic and without any board effect) contents generators. Only in other

situations, we use a shared buffer to keep unacknowledged segments into memory. For memory consumption reasons, we impose a segment size limitation in such cases.

C. Benefits of our architecture

Our architecture make only use of a few (and reasonably sized) shared buffers. The data structure used to store TCP connections states don't include any buffer, making it really small (around 40 bytes). This makes possible to handle the hot points we identified in Section V:

1) Support of multiple in-flight packets: Thanks to our policy, our server is able to have several in-flight packets for most of the Web contents it serves (static pages and idempotent dynamic contents). In other situations, the amount of in-flight segments is limited by the available memory.

2) HTTP persistent-connections handling: In our model, TCP connections data structures are relatively small because they don't each embed their own buffers. Coupled with our event-driven model where only on packet is processed at a given time, handling several simultaneous TCP connections is no more a problem. This makes possible HTTP keep-alive implementation. This also allows us to implement an efficient Comet solution for event notification.

3) Large TCP MSS handling: In input as in output, our model is able to handle large-sized packets even with small-sized buffers. This makes large TCP MSS handling easy.

D. Prototype and measurements

We put to the test our architecture and optimizations through real Web of Things applications implementation and execution in Smews. We made Smews source code publically available[1]. It is fully written in C language, and has been ported to multiple hardware architectures, allowing to run in sensors, smart cards....

We compare Smews performances with two existing embedded Web servers. First, Miniweb [19], is a tiny Web server which needs only a few tens of bytes of RAM. Its functionalities are totally minimal, but it is able to serve simple static Web contents. Secondly, uIP is a small embedded TCP/IP [20] based on protothreads [21]. It requires more ressources than Miniweb, but really provides more functionalities. It is provided with a Web server.

Both Miniweb and uIP source codes are publically available. To allow fair comparisons, we ported Smews, uiP and Miniweb to the same smart card, named Funcard. It uses a 8 bits AVR microcontroller at 8 MHz with 8 kB of RAM and 16 kB of EEPROM. The Funcard network interface is a serial line, with a bandwidth of 10 kB/s. Its latency involves a TCP RTT of around 5 ms.

E. Performances measurements

For our experiments, we use a workstation using Windows XP as operating system, and Internet Explorer 6 as Web browser, the very most common configuration of World Wide Web clients[2]. It is important to remember that Windows TCP/IP stack implements the TCP delayed ACKs strategy.

Our reference Web of Things application is a personal contacts book manager embedded in the Funcard. By using *mashup*, this application can be extended. As an example, via a simple client-side script, we can enrich World Wide Web contents with the private information: when the name of somebody in your contacts book appears in a page, you can get its detailled (and private) information by pointing the name with the mouse.

The Web application is made of four static files and three dynamic contents generators.

- **index.html, style.css, logo.png:** three static files of the main page, of respectively 752, 826 and 5805 bytes
- **script.js:** the client-side scripts, including numerous AJAX interactions, of 3613 bytes
- **cb_extract:** a generator used to retrieve the whole contacts book
- **cb_get:** a generator that returns a single field of the contacts book, specified by URL arguments
- **cb_add:** a generator that adds a new contact or updates a field (via URL arguments), and returns the current number of contacts

In uIP, we have been able to implement a simple version of the Web application, because the Web server provided with uIP does not provide any mechanism for URL arguments management. In Miniweb, only the static file of the application can be served.

Content		uIP	Mweb	Smews	Speed factor $\frac{uIP}{Smews}$	Speed factor $\frac{Mweb}{Smews}$
index.html	752 B	0.70	0.14	0.16	× 4.5	× 0.9
style.css	826 B	0.70	0.14	0.16	× 4.4	× 0.9
script.js	3613 B	1.36	0.50	0.44	× 3.1	× 1.1
logo.png	5805 B	1.76	0.78	0.66	× 2.7	× 1.2
cb_extract	1915 B	1.01	–	0.29	× 3.5	–
cb_get	28 B	0.68	–	0.06	× 10.6	–
cb_add	2 B	0.67	–	0.06	× 10.8	–
whole page		8.2	2.6	1.8	× 4.6	× 1.4

Table I
MEASURED PERFORMANCES on each content of the contacts book, for uIP, Miniweb (noted Mweb) and Smews. The speed ratio between Smews and the two reference Web servers are given.

uIP is a general purpose TCP/IP stack, so its Web server don't benefit of cross-layer protocol optimizations. Unlike Smews, uIP need to store every in-flight packet (for possible retransmissions), because it don't know any property about the HTTP data it is sending. For memory savings reasons, uIP never has more than one in-flight packet.

uIP is extremely slow in comparison with Miniweb and Smews, because of its limitation to a single in-flight packet. It needs 8.2 s to serve the main Web page. Smews is faster than Miniweb (except for the two smallest static files) and, unlike it, it provides a support for dynamic contents generators.

[1]Smews source code available at: http://www2.lifl.fr/~duquenno/Research/Smews

[2]World Wide Web browsers statistics available at http://www.w3schools.com/browsers/

Smews is 1.4 times faster than Miniweb for the whole Web page service, mainly because it handles large TCP MSS.

F. Memory consumption comparison

Server	Volatile memory			Persistent memory		
	Globals	Stack	Total	Code	RO	Total
uIP	3.2 k	118	**3.3 k**	11.4 k	916	**12.3 k**
Miniweb	54	52	**106**	3.7 k	696	**4.4 k**
Smews	118	108	**226**	7.1 k	636	**7.7 k**

Table II
SERVERS MINIMAL MEMORY CONSUMPTIONS ON OUR REFERENCE FUNCARD (IN BYTES)

Table II presents the minimal memory needs of uIP server, Miniweb and Smews, including theirs TCP/IP stack and device drivers[3]. uIP and Smews simultaneous connection number is set to one (note that Miniweb always suffer of this limitation). The measurements are done for the Funcard target.

uIP Web server needs more memory than the two standalone Web servers, Miniweb and Smews. Its minimal volatile memory consumption is of 3.2 kB. Its persistent memory footprint is also significantly higher than Miniweb and Smews ones. uIP Web server presents the worst performances and the higher memory footprint. This illustrates the overheads involved when using a general purpose TCP/IP stack with a layered implementation. Most of this server limitations (in-flight packets limitation, non-persistent HTTP connections, dynamic contents cutting before sending, etc) are compromises done to limit the memory usage.

Miniweb has the lowest memory consumption, with around 100 bytes of volatile memory and 4.4 kB of persistent memory. It is important to keep in mind that Miniweb is really functionally minimal: it is unable to send dynamically generated contents, to handle multiple TCP connection and to read HTTP requests contents. These limitations make it unusable for WoT implementation in practice.

Smews needs 226 bytes of RAM and 7.7 kB of persistent memory. When compared to usual smart cards or sensor hardwares, this memory footprint lets most of the device memory available for user Web applications. Smews has already been ported to several other hardwares than Funcards (8 bit AVR and 8 kB of RAM): WSN430 sensors (16 bit MSP430, 10 kB RAM), MicaZ sensors (8 bit AVR, 4 kB RAM) and a platform using a 32 bits Arm7 and 32 kB of RAM.

VII. CONCLUSIONS

We analyzed the needs of the *Internet of Things*, and proposed a new approach to implement it, based on Web protocols. We shown that the *Web of Things* allows efficient and rich interaction between users and devices. This approach requires to embed Web servers in surrounding devices we want to interact with.

We proposed a cross-layer study of the Web protocols in terms of traffic and memory needs. From this analysis, we designed an efficient event-driven architecture for embedded Web servers.

Finally, we implemented our proposals to prove their feasibility. Measurements done on our prototype shown that it is effectively possible to embed a Web server in very constrained hardware, with only a few kilo-bytes of persistent memory and a few hundred of volatile memory. This prototype is fully able to support *Web of Things* needs: it supports multiples simultaneous connections, persistent connections, dynamic contents service, event notification (comet), etc.

In our future works, we will focus on security issues: we would like to evaluate the costs of TLS support in embedded Web servers, mainly in terms of energy and memory usage.

[3]We used exactly the same device drivers for the three Web servers.

REFERENCES

[1] "Upnp forum," 2008, http://www.upnp.org/.

[2] Y. Gsottberger, X. Shi, G. Stromberg, T. Sturm, and W. Weber, "Embedding Low-Cost Wireless Sensors into Universal Plug and Play Environments," *EWSN, January*, 2004.

[3] H. Song, D. Kim, K. Lee, and J. Sung, "UPnP-Based Sensor Network Management Architecture," *Proc. International Conference on Mobile Computing and Ubiquitous Networking*, 2005.

[4] S. Siorpaes, G. Broll, M. Paolucci, E. Rukzio, J. Hamard, M. Wagner, and A. Schmidt, "Mobile Interaction with the Internet of Things," *Embedded Interaction Research Group*, 2004.

[5] E. Rukzio, M. Paolucci, M. Wagner, H. Berndt, J. Hamard, and A. Schmidt, "Mobile Service Interaction with the Web of Things," *13th International Conference on Telecommunications (ICT 2006), Funchal, Madeira island, Portugal, 2006c.*, 2006.

[6] S. microsystems, "Jxta technology," 2008, http://www.sun.com/software/jxta/.

[7] B. Traversat, M. Abdelaziz, D. Doolin, M. Duigou, J. Hugly, and E. Pouyoul, "Project JXTA-C: enabling a Web of things," *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, p. 9, 2003.

[8] A. Wilson, "The challenge of embedded internet design," *Real-Time Magazine*, pp. 78–80, 1998.

[9] I. Agranat, "Engineering web technologies for embedded applications," *Internet Computing, IEEE*, vol. 2, no. 3, pp. 40–45, May-June 1998.

[10] T. Lin, H. Zhao, J. Wang, G. Han, and J. Wang, "An embedded web server for equipments," *ispan*, vol. 00, p. 345, 2004.

[11] J. J. Garrett, "Ajax: A new approach to web applications," Adaptivepath, AdaptivePath, 2005.

[12] A. Russell, "Comet: Low latency data for the browser," Dojo Toolkit, Dojo Toolkit, 2006.

[13] Dojo, "Dojo fundation bayeux protocol," 2008, http://svn.xantus.org/shortbus/trunk/bayeux/bayeux.html.

[14] E. B. A. Mesbah and A. van Deursen, "A comparison of push and pull techniques for ajax," in *Proceedings of the 9th IEEE International Symposium on Web Site Evolution (WSE)*, S. uang and M. D. Penta, Eds. IEEE Computer Society, 2007, pp. 15–22. [Online]. Available: http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2007-016.pdf

[15] Dojo, "Cometd the scalable comet framework," 2008, http://cometd.com/.

[16] Mortbay, "Jetty web server," 2007, http://jetty.mortbay.org/.

[17] J. Postel, "Rfc 793: Transmission control protocol," Sep. 1981. [Online]. Available: http://www.faqs.org/rfcs/rfc793.html

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – http/1.1," United States, 1999.

[19] A. Dunkels, "The proof-of-concept miniweb tcp/ip stack," 2005, http://www.sics.se/~adam/miniweb/.

[20] ——, "Full tcp/ip for 8-bit architectures," in *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*. New York, NY, USA: ACM Press, 2003, pp. 85–98.

[21] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," in *Proc. of SenSys '06*. New York, NY, USA: ACM Press, 2006, pp. 29–42.