The Web of Things: interconnecting devices with high usability and performance

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*. This approach allows anyone to access embedded devices through a Web application, via a standard Web browser. This* Web of Things *requires to embed Web servers in hardware-constrained devices. We first analyze the traffics embedded Web servers have to handle. Starting from this analysis, we propose a new way to design embedded Web servers, using a dedicated TCP/IP stack and numerous cross-layer off-line pre-calculation (where information are shared between IP, TCP, HTTP and the Web application). We finally present a prototype – named Smews – as a* proof of concept *of our proposals. It has been embedded in tiny devices (smart cards, sensors and other embedded devices), with a requirement of only 200 bytes of RAM 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.*

1 Introduction

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

In the literature, this global devices interconnection is called 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.

Nowadays, more and more services are provided on the Internet using Web applications. We propose to use this same model for surrounding devices. We call *Web of Things* (WoT) the idea of accessing surrounding devices through Web applications. This concept provides a great accessibility for devices and makes them easier to program. Furthermore, embedded Web contents can be merged with Internet ones, opening new perspectives for Web applications.

The Web of Things require to embed Web servers in small devices. At first sight, implementing a HTTP/TCP/IP stack on devices with only a few hundreds of bytes of RAM and a few kilo-bytes of EEPROM seems unsuitable. We show that an event-driven architecture allows to implement extremely lightweight Web servers with efficient cross-layer optimizations. We detail these optimizations and measure their benefits.

This paper is organized as follows. Section 2 is a state of the art of existing Internet of Things solutions. It also introduces the concept of Web of Things. In Section 3, we describe how WoT applications can be designed. Section 4 presents an analysis of the traffics an embedded Web server has to manage, in order to identify critical points for embedded Web server design. We present a new event-driven architecture for embedded Web servers in Section 5. In Section 6, we prove the feasibility of Web of Things via a prototype. We measure the benefits of our proposals by comparing our prototype to state of the art embedded Web servers. We finally conclude in Section 7.

2 State of the Art

In this section we define the Internet of Things and we describe existing standards and solutions for IoT protocols.

2.1 The Internet of Things

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 schemes allowing 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 and spatially local network that includes every objects a pearson should interact with. The main concerns of PANs are defined in [21, 17]. They can be summarized as follows: (i) objects discovery and addressing must handled by the PAN protocols (ii) devices must be able to interact with the whole PAN and possibly with external networks and (iii) 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 kilobytes of volatile and persistent memory, a CPU frequency of a few MHz, hard energy constraints, and low-throughput physical links (*e.g.*, Bluetooth, USB, ZigBee, ...).

2.2 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 protocols and technologies, such as UDP, TCP, HTTP, HTTPU, Web services, 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, it does not propose any authentication protocol. 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 like HTTPU. Finally, UPnP is quite heavy because it is based on heavy protocols (SOAP, WSDL, ...). This makes it unusable into very constrained hardware. In [11] and [22], a proxy is proposed between the PAN and sensor nodes, because sensors are unable to embed the whole UPnP stack.

2.3 Alternatives to UPnP

The JXTA technology [14] is a solution for peer-to-peer applications design, allowing to interconnect heterogeneous devices into a same network. JXTA-C [23] 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 [24] that an intersting solution for global devices interconnection consists in using embedded Web severs (EWS). It is shown in [12, 7] that devices with a few kB of RAM and of EEPROM are able to run a Web server.

2.4 Towards a Web of Things

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.

The *Web of Things* (WoT) consists in using Web protocols and technologies into a client-centric PAN architecture. The computer, PDA or cell phone of the user runs a client (a standard Web browser) while every node into the PAN runs an Embedded 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. A user may give long-term rights to objects (via a ticket), thus allowing machine-to-machine interaction. In fact, Web technologies are not only suitable for human-machine interactions. The WoT is based on existing widespread technologies: TCP/IP, HTTP, Web applications. This ensures a great accessibility as well as ease of development, via numerous existing design frameworks. Secure interactions can be implemented with HTTPS.

2.5 Embedding Web servers in tiny devices

Two points make possible the usage of Web servers in constrained devices. First, unlike Internet Web servers, embedded ones do not have to handle thousand simultaneous connections and requests. Secondly, in the WoT, the Web client is more powerfull than the server. By using AJAX and Comet (presented in Section 3), we are able to deport Web application processing from the server to the client.

Several works have been done about embedded TCP/IP stacks, resulting in prototypes like TinyTCP [3], mIP [19] or lwIP and uIP [6]. They only implement a subset of TCP/IP RFCs, but they are able to communicate with usual IP nodes. They provide a socket-like interface for embedded applications. Embedded Web servers can be written over such a TCP/IP stack. uIP is a simple event-driven TCP/IP stack is based on protothreads [8] and only requires a few kilo-bytes of RAM.

An other approach consists in designing a Web server that includes its own dedicated TCP/IP stack, allowing better performances and lower memory costs. Numerous prototypes of such embedded Web servers have been proposed: WebIt [12], iPic Web server [20] or Miniweb [7]. Most of them are provide only poor functionalities: minimalist TCP support, limitation to simple static Web pages, etc. They nevertheless show that functionnay minimal Web servers can run with only a few tens of bytes of RAM.

Miniweb and uIP particularly drawn our attention, and, because their source code are publically available (note that a Web server is provided with uIP), we choose them as references for our experiments in Section 6.

3 Designing Web of Things applications

In this section, we present some basics about embedded Web applications design. Then, we introduce two typical Web of Things applications we choose as references for our analysis and experiments.

3.1 AJAX: the incontrovertible Web application methodology

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

The AJAX[10] 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 (*i.e.*, the browser) collects several static files, containing style (CSS), contents (HTML), and applicative code (JavaScript).

2. The running phase. The client executes the applicative code downloaded in the first phase, and interacts with the server by sending asynchronous requests, allowing the Web page to update itself dynamically.

Figure 1 shows the repartition of HTTP content lengths returned during the two distinct AJAX phases when browsing on well-known AJAX applications (*e.g.*,, Gmail[1], Google Calendar[2] and Yahoo! mail[3]). Results show that during the first phase, numerous large-sized contents (mainly static files) are served (average returned size about 8 kB). During the second phase, small-sized contents (mainly generated by the server) are received by the client (average returned size about 600 bytes).

AJAX reduces Web traffic because dynamic 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 browser: the server sends small generated data while the client runs behavioral code. This is particularly interesting for Web of Things applications, where the Web server often has less ressources than the Web client.

[1]Gmail: `http://mail.google.com`
[2]Google Calendar: `http://www.google.com/calendar`
[3]Yahoo! mail: `http://mail.yahoo.com`

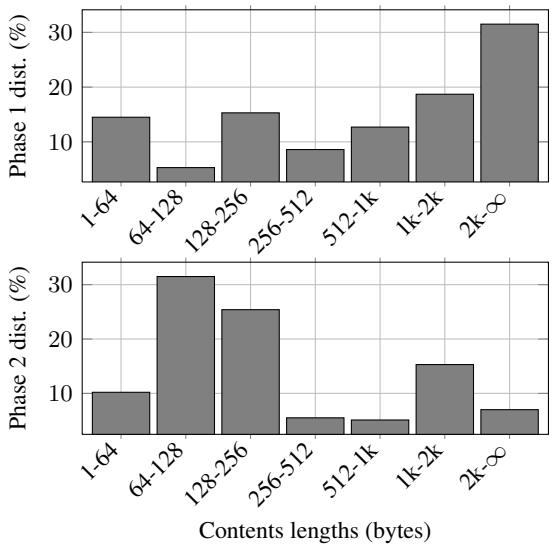

Figure 1. HTTP contents lengths distribution for sample AJAX applications in phase 1 (loading) and 2 (running)

3.2 Mashup: building Web applications from multiple sources

A new trend in Web applications design is called *mashup*. It consists in building Web pages using resources retrieved from multiple Web servers. The most often, the merge is done using AJAX. As an example, anyone can talk about a movie in its personal blog, and, thanks to mashup, include information about near movie theaters. Such information are provided by an external server which provides a JavaScript API for such an usage. Mashup opens new perspectives for Web applications, and is particularly well-designed for WoT, allowing heterogeneous devices to build together a rich user interface.

3.3 Comet: event notification in Web context

A common need for embedded devices consists in event notification (*e.g.*, for home automation, sensors, ...). In a first sight, HTTP seems unsuitable for such behavior because it is based on a simple request/response model. Nevertheless, a new model named Comet [18] appeared these last years, allowing a Web application to push data from the server to the client. A Comet protocol specification already exists and is named Bayeux [4].

A comparison of AJAX pull and push (Comet) methods is presented in [13]. It shows that, while Comet is better in terms of data consistency and traffic workload, it suffers of scalability issues in term of processing. Server-side comet

management is hard for two main reasons. First, 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 designed to provide Comet support [5, 15]. Secondly, 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.

Scalability issues have 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.

3.4 Web of Things applications examples

Based on the Web technologies we presented, interesting Web of Things applications can be designed. We propose two examples of such applications.

3.4.1 A shared calendar preserving privacy

On-line Web calendars are a great example of widely used Web 2.0 applications. They mainly have two strengths: (i) they are accessible by their users anywhere they are located and from any workstation, PDA or cell phone and (ii) they allow users to share information about their availability. On-line calendars also have issues are in terms of privacy and connectivity. First, they require to store private information on Internet servers. Secondly, they are not accessible without an Internet access.

By building a Web of Things application on a smart card coupled with an on-line Web application, we propose a simple solution to both privacy and connectivity issues. Thanks to mashup, private information details can be stored and managed by the smart card in a secured environment near the user. Other information are shared on-line. Moreover, even with no Internet connexion, the calendar can be accessed and modified by its owner. Once an Internet access is found, public information are updated from a subset of the smart card information. By placing the calendar in a personal device, we also enrich one of the main strengths of Web calendars: their accessibility.

3.4.2 A personal contacts book

Our second application example is a personal contacts Book manager. Private information about the user contacts are stored and managed by the smart card, and are accessible even with no Internet access. By using *mashup* (see Section 3), 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.

4 Handling Web traffics in tiny devices

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 efficient to implement stateless behaviors. Stateful behaviors are 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 stack, regarding their requirements in terms of memory and traffic.

4.1 HTTP: a half duplex protocol over TCP

TCP (defined in [16]) 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).

4.2 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. A MSS of 1460 if often used because it 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 require to manage large packets, which is an issue on memory-constrained systems.

4.3 Supporting HTTP keep-alive

HTTP has been designed to run over TCP. Since HTTP 1.1 [9], 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 requires to manage multiple TCP connections simultaneously. It is a stateful property that is hard to handle in memory-lightweight event-driven systems.

4.4 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.

5 An event-driven architecture for an efficient tiny Web server

Based on the WoT applications needs (Section 3) and on our analysis of the Web protocols (Section 4), we designed a new embedded Web sever, named *Smews*. In this section, we describe its novel architecture and optimizations.

5.1 Smews presentation

Smews source code is publically available[4]. It is fully written in C language, and has been ported to multiple hardware architectures, allowing to run in sensors, smart cards and other devices. Smews implements all the proposals we describe in the next sections, and will be used for our experiments in Section 6.

For study, our reference target is a smart card called Funcard, using 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.

5.2 An embedded Web server based on off-line pre-calculations

As mentioned in Section 3, 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 application, containing static files (*e.g.*, html, js, css, ...) and server-side generators. Server-side generators implemented by naive c functions or by OgO files (PhP-like files).

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. Figure 2 shows how files are processed in our tool-chain.

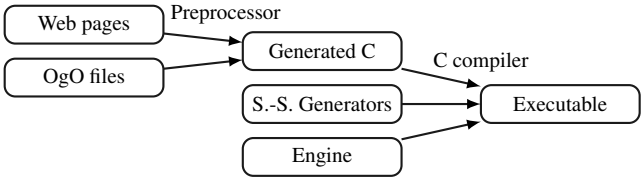

Figure 2. Smews compilation tool-chain

The pre-processing phase of static contents allows us to propose off-line optimizations such as protocol checksums or headers pre-calculation.

5.2.1 HTTP, TCP and IP headers off-line generation

HTTP, TCP and IP headers have constant parts that our tool-chains knows at compile-time. Each of these parts is statically build off-line, allowing the embedded TCP/IP stack to run faster. It only has to complete variable parts of headers (destination IP and port, sequence numbers, etc.). This optimization is used for static contents as well as for server-side generated contents. The major part of HTTP headers are known at compile-time and can be pre-generated.

5.2.2 TCP checksums chunks pre-calculation

It is has been shown [2] that checksum calculation is a critical part of TCP/IP stacks in terms of processing. That is why some tiny embedded Web servers like Miniweb [7] compute TCP checksums on static Web contents at compile-time. This is done by a dedicated tool, in charge of pre-calculating each IP packet Miniweb will send when serving a given file. This approach is very interesting, but has a main drawback: it forces to fix the TCP MSS off-line to the lower legal value: 200 bytes. Indeed, the client maximal MSS is unknown at compile time.

We measured the time spent by our prototype for various phases when sending a IP packet. These measurements are synthesized in Table 1, and show that more than 80 % of the processing time is spent for TCP checksumming.

As Miniweb, we propose to compute TCP checksums off-line, but, instead of pre-generating entire IP packets, we compute checksums on files by data chucks of a given size.

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

Processing phase	Time
IP header	0.3 ms
TCP header	0.4 ms
Segment checksum	2.9 ms
Total time	3.6 ms

Table 1. CPU Time spent by our prototype in different sending phases

At runtime, the server only has to sum the partial checksums in order to retrieve the final TCP checksum. This approach does not set the TCP MSS at compile-time.

Chunk size	Memory cost	Proc. time	Time saved
-	0 B	2.92 ms	0 %
4 B	704 B	1.48 ms	40 %
8 B	352 B	0.78 ms	59 %
16 B	176 B	0.39 ms	70 %
32 B	88 B	0.23 ms	75 %
64 B	44 B	0.16 ms	77 %
128 B	22 B	0.11 ms	78 %

Table 2. Chucked checksum pre-calculation impact on processing time

Table 2 shows the performances obtained with various chucks sizes. With a chunks size of 32 bytes, only 0.23 ms are spent for checksum calculation, saving 75 % of the total running time. The chunks size has to be well chosen, because too large chucks will badly fit with various MSS.

5.2.3 HTTP options parsing tree

When receiving a HTTP request, the Web server has to parse its options. We propose to achieve this work using a parsing tree, thus avoiding the need to completely store the request in memory before processing it. This also avoids the usage of multiple and heavy string comparisons. The input is simply compared byte after byte with the tree. Because the web contents are known at compile time, the tree synthesizes every possible URL. The list of possible URL arguments for each web content is also fully pre-processed. Such an approach allows to decode the whole HTTP request without any buffer, and with a minimal processing cost.

5.3 Proposed architecture

The architecture we propose is event-driven 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.

5.3.1 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.

5.3.2 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.

5.4 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 (less than 40 bytes). This makes possible to handle the critical points we identified in Section 4:

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.

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 possible. This makes possible HTTP keep-alive implementation. This also allows us to implement an efficient Comet solution for event notification.

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, even with um pre-calculations.

6 Experiments

We put to the test our architecture and optimizations through real Web of Things application implementation and execution in Smews. We compare Smews performances with the two state of the art solutions we presented in Section 2: Miniweb and uIP. Miniweb functionalities are minimal, but it only require a few tens of bytes of RAM. uIP Web server needs more ressources than Miniweb, but really provides more functionalities. To allow fair comparisons, we ported the three servers to the same target: the Funcard, described in Section 5.1.

As far as we know, no benchmark have been proposed for embedded Web servers. We compared the performances of the three servers on the contacts book application we presented in Section 3. We also made the source code of the applications publically available in order to provide a benchmark for other works.

6.1 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[5]. Remember that Windows TCP/IP stack implements the TCP delayed ACKs strategy.

The contacts book 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.

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.

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

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 3. 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.

This limitation is the reason why uIP is extremely slow in comparison with Miniweb and Smews. It needs 8.2 seconds 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. Smews is 1.4 times faster than Miniweb for the whole Web page service, mainly because it handles large TCP MSS. The reason why this factor is greater than individual factors (varying from 0.9 to 1.2) is that, unlike Smews, Miniweb do not support keep-alive connections.

6.2 Memory consumption comparison

Table 4 presents the minimal memory consumptions of the three servers, including theirs TCP/IP stack and device drivers[6]. For these measurements, Miniweb and Smews use an input buffer of only one byte. uIP uses a buffer of 1500 bytes in order to support a MSS of 1460 bytes. uIP and Smews simultaneous connection number is set to one (note that Miniweb always suffers of this limitation). Measurements are done on our reference target, the Funcard (presented in Section 5.1).

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 4. Servers minimal memory consumptions on our reference Funcard (in bytes)

On our reference smart card, uIP Web server needs more

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

volatile and persistent memory than the two monolithic Web servers, Miniweb and Smews. Miniweb has the lowest memory consumption, with around 100 bytes of volatile memory and 4 kB of persistent memory. It is important to know that Miniweb is unable to send generated contents, handle multiple TCP connection or various MSS and decode HTTP requests. It can not support a Web application, but is a nice low-bound in term of ressources requirements for embedded Web server implementations.

Smews requires 226 bytes of volatile memory and 7.7 kB of persistent memory. When compared to usual embedded devices, this memory footprint frees most of the 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.

7 Conclusions

We analyzed the needs of the *Internet of Things* and proposed a new approach to design it based on Web technologies, named *Web of Things*.

We proposed a study of the Web protocols in terms of traffic and memory needs, thus identifying the main issues for embedded Web server implementation. From this analysis, we designed an efficient event-driven architecture for embedded Web servers. This architecture intensively uses off-line pre-calculation, and, by sharing information between IP, TCP, HTTP and the Web application, it allows new cross-layer optimizations.

Thanks to our prototype, we prove the relevance of the *Web of Things* and the efficiency of our proposals, both in terms of performance and of memory requirements. Smews makes a gap in term of performance with the state of the art solutions Miniweb and uIP. It is able to run with only a few kilo-bytes of persistent memory and a few hundred of bytes of RAM. This prototype is able to run rich Web applications: 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.

References

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

[2] M. Allman, V. Paxson, and W. Stevens. Rfc 2581: Tcp congestion control, 1999.

[3] G. H. Cooper. Tinytcp, 2002. `http://www.csonline.net/bpaddock/tinytcp/`.

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

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

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

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

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

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

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

[11] 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.

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

[13] 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)*, pages 15–22. IEEE Computer Society, 2007.

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

[15] Mortbay. Jetty web server, 2007. `http://jetty.mortbay.org/`.

[16] J. Postel. Rfc 793: Transmission control protocol, Sept. 1981.

[17] 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.

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

[19] S. Shon. Protocol implementations for web based control systems. *International Journal of Control, Automation, and Systems*, 3:122–129, March 2005.

[20] H. Shrikumar. Ipic - a match head sized webserver., 2002.

[21] 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.

[22] 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.

[23] 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*, page 9, 2003.

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