Cross-layer enhancement of Web servers dedicated to small devices

Simon Duquennoy, Gilles Grimaud, Jean-Jacques Vandewalle

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Cross-layer enhancement of Web servers dedicated to small devices

Simon Duquennoy — Gilles Grimaud — Jean-Jacques Vandewalle

N° 0349

February 2008

Thème COM

Rapport technique

ISSN 0249-0803 ISRN INRIA/RT--0349--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche LILLE - NORD EUROPE

Cross-layer enhancement of Web servers dedicated to small devices

Simon Duquennoy*, Gilles Grimaud* , Jean-Jacques Vandewalle†

Thème COM — Systèmes communicants
Équipes-Projets POPS

Rapport technique n° 0349 — February 2008 — 24 pages

Abstract: Nowadays, embedded systems are more and more present around us, with an increasing accessibility need. Instead of using dedicated protocols, involving dedicated client and server softwares, we claim that embedding a Web server into these systems allows a great accessibility, interoperability, maintenability and easiness of development. So anyone can access and configure a router, a sensor or a domotic system from any computer or personal digital agent, using only a standard Web browser. Using modern Web methodologies like AJAX, interactive applications can be served.

However, this solution is technically hard to apply, because of the hardware limitations of targeted embedded systems (often some MHz of CPU frequency and only a few kilo-bytes of RAM), in contrast to the heaviness of usual Web servers (and Web protocols).

In this report, we first present a cross-layer analysis of the TCP/IP protocols when used for dynamic Web applications over HTTP. We test existing embedded Web servers and we analyze theirs performances. Starting from these analysis, we propose new solutions for an efficient and memory-lightweight Web server conception. We implemented all our propositions, giving a new embedded Web server, able to serve efficiently dynamic Web applications with a RAM usage of less than one kilo-byte without any underlaying operating system.

Key-words: embedded systems, Web server, communication stack, TCP/IP, AJAX

* IRCICA/LIFL, CNRS UMR 8022, Univ. Lille 1, INRIA Lille-Nord Europe, POPS research group
† Gemalto Technology & Innovations, France

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley, 59650 Villeneuve d'Ascq
Téléphone : +33 3 59 57 78 00 — Télécopie : +33 3 59 57 78 50

Amélioration transversale de serveurs Web dédiés aux pérphériques très contraints

Résumé : De nos jours, les systèmes embarqués sont de plus en plus nombreux et ont un besoin grandissant d'accessibilité. Au lieu d'utiliser des protocoles dédiés, imposant l'utilisation de logiciels clients et serveurs, nous soutenons que l'utilisation de serveurs Web sur ces cibles en augmente l'accessibilité ainsi que la maintenabilité et facilite le développement. Ainsi, tout le monde peut accéder et configurer un routeur, un capteur de terrain ou un système domotique depuis n'importe quel ordinateur ou PDA, via un simple navigateur Web. En utilisant les technologies modernes du Web telles qu'AJAX, des applications interactives peuvent être servies.

Cependant, cette solution est techniquement difficile à mettre en oeuvre à cause des limitations matérielles des systèmes embarqués ciblés (souvent un CPU à quelques MHz et seulement quelques kilo-octets de RAM), comparées à la lourdeur des serveurs Web classiques (et des protocoles du Web).

Dans ce rapport, nous présentons tout d'abord une analyse transversale des protocoles TCP/IP lorsqu'utilisés pour servir des applications Web dynamiques. Nous testons des serveurs Web embarqués existants et analysons leurs performances. A partir de cette analyse, nous proposons de nouvelles solutions pour concevoir des serveurs Web efficaces et peu consommateurs de mémoire. Nous avons implémenté toutes nos propositions, engendrant un nouveau serveur Web embarqué, capable de servir efficacement des applications Web dynamiques avec une consommation en RAM de moins d'un kilo-octet, sans aucun système sous-jacent.

Mots-clés : systèmes embarqués, serveur Web, pile de communication, TCP/IP, AJAX

1 Motivation

1.1 Introduction

Embedded systems like routers, sensors or domotic systems are growing around us. We need efficient ways to interact with them. This increasing interaction need makes unadapted the classical solution consisting in the development of dedicated client/server softwares. These specialized applications are efficient but involve high development costs. Their evolution is difficult, they are heavy to maintain, and they need to be deployed on every potential client machine.

An other approach consists in installing a Web server with a TCP/IP stack in the embedded system. Server applications become regular Web applications, easy and fast to develop (compared to a dedicated application). This solution presents several advantages:

- The installation phase on the client side is avoided. Today, a majority of workstation, PDA or cell phone run a Web browser.
- The Web applicative support is, thanks to the Internet, uniform and widespread. This makes applications development easy. Moreover, Web applications guarantees a good portability when accessed from heterogeneous clients.
- Deployment of new applications is simplified and it can be done directly using an integrated upload Web interface.
- A Web application is easy to maintain and to update. Clients stay unchanged. A new version may consist only in a new set of contents the Web server will be able to send.
- TCP and IP protocols allow the Web server to be directly connected to existing networks, as the Internet.

1.2 Context

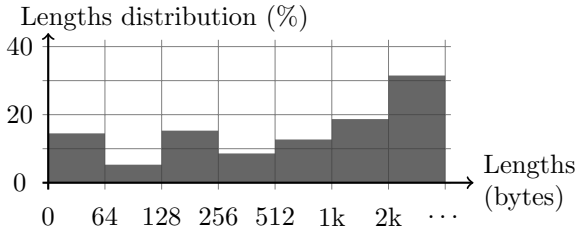
Embedded systems like sensors, domotic elements or smart card are physically constrained (often some MHz of CPU frequency, a few kilo-bytes of RAM and hundreds of kilo-bytes of persistent memory, like EEPROM or Flash). Using a general purpose Web server (*e.g.*, Linux, Apache and JEE or Windows, IIS and .net) onto this kind of hardware is hard because of the heaviness of HTTP and TCP/IP, and of the memory footprint of these operating systems and softwares (a typical workstation TCP/IP stack has a volatile memory usage of hundreds of kilo-bytes [19]). The Web has initially been designed for powerfull and memoryfull servers, using multi-threaded operating systems. However, in our situation, the embedded Web server is not accessed simultaneously from thousands of clients, like Web servers on the Internet. This makes possible a lightweight implementation.

Thanks to the Internet, Web technologies evolution is very fast. In the recent years, a new application development methodology appeared, named AJAX [9]. This methodology involves a workload deportation, from the Web server to the Web browser. This is particularly interesting in our situation, where the Web server often has less ressources than the Web client. The behavior of an AJAX Web application can be separated into two phases:

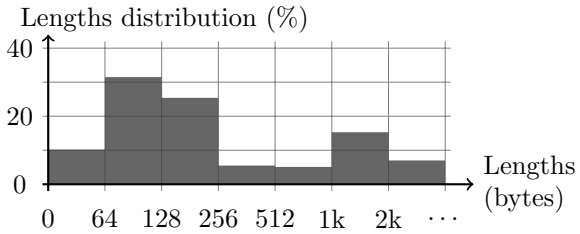
1. The loading phase. The client starts by collecting several static files, containing style (CSS), contents (HTML), and applicative code (JavaScript).
2. The running phase. The client executes the applicative code it downloaded in the first phase, and it interacts with the server by sending asynchronous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

The AJAX model allows to design highly interactive applications with a task repartition between the client and the server. It also reduces Web traffic by sending non-formatted data. Formatting rules are loaded once in the initialization phase, factorizing informations and reducing redundancies.

Before working on embedded Web servers design, it is important to analyze the traffic they have to manage. This analysis allows to identify hot points where the server needs to be efficient. Today, most of modern Internet Web applications use the AJAX model.



(a) Phase 1: initial application loading



(b) Phase 2: application execution

Figure 1: Contents lengths distribution for sample AJAX applications

Figure 1 shows the repartition of HTTP content lengths returned when browsing onto three popular AJAX applications (Gmail[1], Google Calendar[2] and Yahoo! mail[3]), during the two distinct AJAX phases. Results of this analysis show that during the first phase, numerous large-sized contents (mainly static files) are served (average returned size around 8 KB). During the second phase, small-sized contents (mainly generated by the server) are received by the client (average returned size around 600 bytes). Note that these measurements are done on workstation rich Web applications. In embedded systems context, Web applications might send even smaller generated contents.

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

Because the AJAX model seems to be well designed for embedded Web applications, an embedded Web server must be efficient during the two AJAX phases, *i.e.*, when sending *large-sized* static files as well as *small-sized* generated contents.

In Section 2, we present a state of the art of embedded TCP/IP stacks and embedded Web servers. Section 3 analyzes TCP/IP performances when used for HTTP. Section 4 makes a detailled analysis of the two embedded Web servers we choose as references and identifies theirs strengths and weaknesses. Based on our servers studies, we propose new solutions to improve dynamic Web application service in Section 5. Possible future works are presented in Section 6. We finally conclude in Section 7.

2 State of the art

This section presents a state of the art of existing works related to embedded Web servers. We present works about TCP/IP protocols specialization for embedded devices, about small-sized TCP/IP stack and about embedded Web servers.

2.1 Adapting TCP/IP protocols to embedded systems constraints

Numerous works have been done to specialize TCP/IP for embedded systems. Distributed caches usage [8] allows to reduce retransmissions costs, while TCP/IP specific header compression [8, 20] reduces protocols overheads. The main drawback of these methods is also their strength: they modify protocols, making them non TCP/IP compliant. These works are not compatible with our initial aims because they cannot be integrated into unmodified Web architectures.

2.2 TCP/IP stacks for embedded systems

Both TCP and IP are described by RFCs [14, 15]. TCP includes a lot of complex mechanisms (congestion avoidance, acknowledgments, transmission windows, retransmissions, etc.), making it hard to use in embedded systems.

2.2.1 MIP

By implementing a subset of these RFCs, it has been shown [19] that communication with classical TCP/IP stack stays possible. The stack presented in this work, called mIP, is functionally totally minimized (no congestion control, retransmissions, etc.) and it is able to run an embedded Web server properly, using around one kilo-byte of RAM. The source code of mIP is not provided by the authors, and the performances of their implementation is not compared to other existing works.

2.2.2 μIP and lwIP

Dunkels proposes two embedded TCP/IP stacks [5]. LwIP is fully RFCs-compliant and it uses a layered protocol implementation. Its list of supported

protocols contains IP, TCP, UDP and ICMP and it can easily be extended. Meanwhile, μIP implements a subset of TCP/IP with a monolithic approach, and can only support a limited set of protocols (IP, TCP and ICMP). μIP is smaller than lwIP in terms of memory usage. Implementations of theses stacks are provided by the authors, as well as applications examples. Because of the small size of μIP, and because it is publically available with a Web server, we choose this embedded TCP/IP stack as one of our references for our experiments.

2.3 Embedded Web servers

2.3.1 Hardware Web servers

Hardware Web severs have been proposed [10, 16], based on hardware-software co-design. These works are out of the scope of our study, because they cannot be used on any kind of hardware. Fully software-based embedded Web servers have the big advantage to be portable to any hardware and used for any kind of embedded systems.

2.3.2 Software Web servers

As opposed to general purpose Web servers, embedded ones are not constrained in being executed in user-space over an operating system, using its provided general-purpose socket interface. They often do not use any operating system, using their own dedicated TCP/IP stack and directly accessing their own device drivers. As a consequence, they can be optimized at every software level of the system, unlike classical workstation Web servers.

Several works [1, 11, 4, 10] show that it is possible to embed a Web server into constrained hardwares. They often detail the implementation, depending on their targets architectures. They explain how Web pages are stored (in RAM, in EEPROM) after having been pre-processed by a dedicated tool. Most of these servers provide a simple (and often inextensible) way to integrate generated contents: tags are included in HTML pages and substituted at run-time by the server (*e.g.*, by a sampled temperature, a hardware state information or a statistic). Performances of these servers are not compared to others and their source code is rarely published.

The proof-of-concept miniWeb server [6] particularly drawn our attention. Its goal is to provide a memory minimal Web server, simply able to serve a few static documents. It uses a monolithic implementation, *i.e.*, with its own dedicated TCP/IP stack, tightly coupled with the whole Web server. MiniWeb source code is provided by the authors. Because of its extremely low memory consumption and its interesting design choices, we choose this server as a reference for our study, as well as the μIP web server.

3 Analysis of TCP/IP used for HTTP

TCP is a general purpose protocol for connected communications. It uses complex mechanisms (retransmissions and acknowledgments management, congestion control, sliding window, etc.) to allow reliable communications. In this

section, we present an analysis of TCP/IP traffics when HTTP is used as the application layer protocol.

This analysis is fully independent of the link layer protocol, because embedded Web servers can use TCP/IP above various protocols (*e.g.*, Ethernet, PPP, SLIP) with their own properties and overheads.

3.1 TCP behavior

TCP can be used for any kind of applications, allowing bidirectional reliable communications. In TCP, data can be sent by the two hosts in a random order. TCP allows data piggybacking, *i.e.*, sending a segment containing both data and acknowledgment. This makes TCP void acknowledgments less frequent.

Figure 2 is a cross-layer view of HTTP traffic over TCP/IP, showing successive GET requests and using the classical TCP MSS of 1460 bytes[4] over a single TCP connection (in case of multiple connections, each one follows the same scheme). A GET request from a standard Web browser has a length of hundreds of bytes, while HTTP 1.1 responses are arbitrary sized. We can see that under these typical conditions, TCP has a particular 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 given 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 piggybacking is rare in this situation. Void TCP acknowledgments are numberred in Figure 2. Let ACKn be the n^{th} acknowledgment of the drawing. ACK1 is always sent without any data, because this policy is recommended in the TCP RFC during the three way handshake of connection establishment. Our observations on Internet traffics shows that HTTP responses are sometimes piggybacked, acknowledging their requests. That is why the last occurrence of ACK2 (who acknowledges the last segment of the request) is not always sent. When a client receives the end of a HTTP response and needs to request the server again, it may piggyback the next request, thus acknowledging the last respone. As the ACK2 one, the last occurrence of ACK3 is optional. As soon as a HTTP request or a response exceeds the TCP MSS, several occurrences of ACK2 and ACK3 are sent without any piggybacking. ACK4 is always sent alone because when receiving a TCP segment with a FIN flag, the client stack acknowledges it before deciding to close the connection. ACK5 is obviously always sent without any data, because it is the last connection packet.

This analysis shows that TCP rich behaviors are under-used when used for HTTP. Data are always sent in an unidirectional way and are quite rarely piggybacked.

3.2 Data overhead

Embedded systems network interfaces are often only half-duplex (*e.g.*, sensor with 802.15/ZigBee, smartcard with 7816-4/APDU, but also 803.3/Ethernet and 802.11/Wifi). For this reason, in our study, we do not take into account

[4]The maximal TCP MSS usable over Ethernet is 1460 B, because Ethernet MTU is 1500 B. That is why it is a classically used TCP MSS.

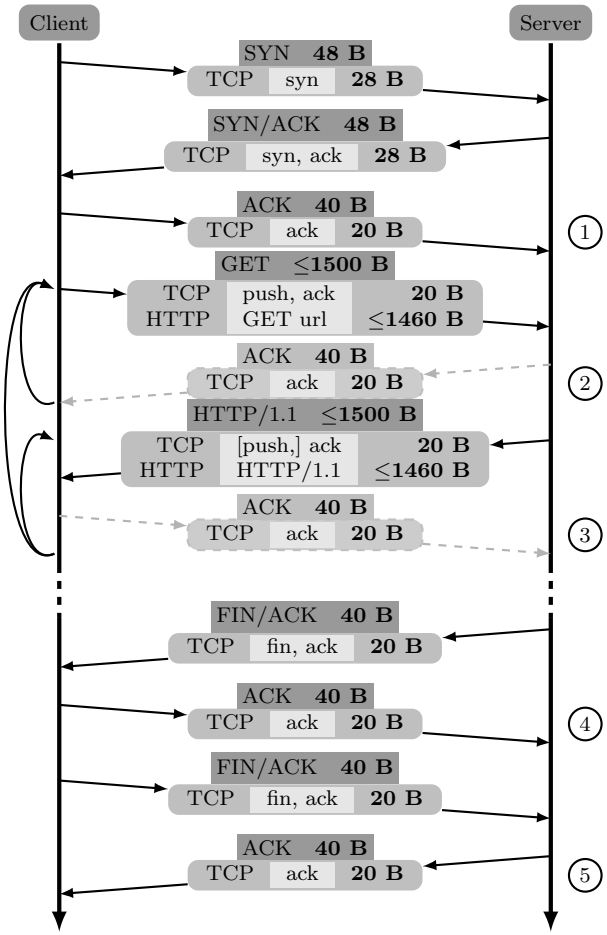


Figure 2: HTTP over a TCP connection. Packets sizes include IP headers of 20 bytes. Dashed arrows and boxes indicates optional packets.

packets direction. We also consider point-to-point communications, where latency are negligible[5].

Both TCP and IP use a header of at least 20 bytes for every packet. When a host is sending data, it regularly receives TCP acknowledgments. As mentioned in Section 3.1 when HTTP is used, TCP acknowledgments are mainly TCP void segments. Let α be the void acknowledgment reception frequency for every sent segment. The TCP/IP data overhead is:

$$\begin{aligned} &Data\ overhead = \\ &(IP\ header\ size + TCP\ header\ size \\ &+ \alpha \times TCP\ ACK\ size)/TCP\ MSS \end{aligned}$$

[5]For domotic applications or smartcards, this assertion is obvious. In case of multi-hop sensor networks, the latency is highly dependent on the routing algorithm, what is out of the scope of this paper.

3.3 Applicative throughput

In order to decrease the amount of TCP acknowledgments sent by TCP, a well-known algorithm called *delayed ACKs* has been proposed [3]. It is implemented by most of workstation operating systems TCP/IP stacks. It consists in acknowledging an incoming TCP segment only (i) when receiving a second TCP segment or (ii) after waiting for 200 ms.

Let β be the frequency of delayed by 200 ms TCP acknowledgment. If the sender sends data continuously, we have $\beta = 0$ and $\alpha = 0.5$. If it waits every segment to be acknowledged before sending the next segment, we have $\beta = 1$ and $\alpha = 1$. The applicative throughput is:

$$\begin{aligned}&\textit{Applicative throughput} = \\ &MSS/(\beta \times \textit{ACK delay} + \\ &(MTU + \alpha \times \textit{ACK size})/\textit{physical throughput})\end{aligned}$$

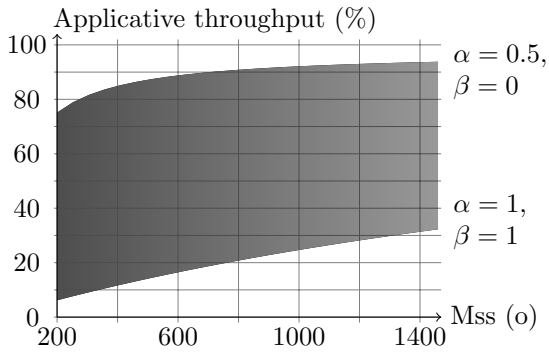


Figure 3: Maximal applicative throughput on 14.4 KB/s line

Figure 3 shows the maximal applicative throughput (percentage of the physicall throughput) that can be obtained when using HTTP over TCP/IP on a 14.4 KB/s line. It shows how much performances are degraded when $\alpha = 1$ and $\beta = 1$. In such situation, acknowledgments are always delayed, and packets are sent only every 200 ms. Whatever the physical maximal throughput, no more than 5 TCP segments are sent every second. If the TCP sender sends data continuously (thus having $\alpha = 0.5$ and $\beta = 0$), its maximal applicative throughput grows with the physical line throughput.

This figure also shows that the TCP MSS have an impact on applicative throughput. A big MSS should be used when it is possible (because TCP and IP headers have a constant size once the connection is established).

To allow the greatest applicative throughput for an embedded TCP/IP stack, that last one must be able to use any MSS and to have several in-flight (*i.e.*, sent but unacknowledged) TCP segments. Because of the TCP retransmission mechanism, sent segments have to be stored until they are acknowledged. This consumes a part of the limited RAM available in embedded targets.

4 Reference embedded Web servers study

In Section 2, we presented our two reference embedded Web servers: μIP server (with rich functionalities) and miniWeb (extremely memory lightweight). To evaluate their performances and to identifies their weakness, we ported these two servers to a same reference target, whose characteristics are given in Table 1. This section details these two servers. We identify their strengths and weaknesses and measure their memory consumptions. The servers performance is measured in Section 5.

Microcontroller	ARM7, 16.78 MHz
Bus	32 bits
IWRAM	32 KB
SRAM	256 KB
EEPROM	8 MB

Table 1: Reference target characteristics

4.1 μIP Web server analysis

4.1.1 Server presentation

μIP Web server is able to serve static files as well as dynamically generated contents. Files are pre-compiled then embedded with the server in the compilation phase. Dynamic contents generation is provided with JSP/ASP/PhP-like methodology: dedicated markers embedded in HTML files are substituted at runtime by the server.

μIP is implemented using protothreads [7], allowing low-cost (in term of memory) multi-threading.

4.1.2 Traffic analysis

As referred in Section 3.3, TCP retransmissions require to store every in-flight segment until it becomes acknowledged. For memory consumption reasons, μIP discards every packet just after having sent them. In case of a packet loss, the stack asks the application to re-generate the last packet before retransmitting it. This choice does not allow to have more than one in-flight packet, decreasing significantly the maximal application throughput (see Section 3.3).

μIP Web server is not able to handle HTTP persistent connections. This choice has been done for two reasons: (i) because it allows to free TCP connections after each HTTP response, realizing memory savings and (ii) because it reduces considerably HTTP management complexity (no need to wait for new connections and for data from an existing connection at the same time). With non-persistent connections, TCP connections have to be established and closed for each HTTP request, involving huge traffic overheads. This is particularly perceptible when sending small-sized data.

4.1.3 Memory consumption

μIP drivers interface need to read/write data from/into a global buffer, which size must be at least equals to the maximal MSS plus TCP and IP header size. To allow data reception while the TCP/IP stack is running, a second driver-level buffer of the same size is needed. Classical TCP MSS is of 1460 bytes. As shown in Section 3.3, the applicative maximal throughput of a TC/IP stack is highly dependent on the MSS used. As a consequence, when embedding μIP, a choice has to be made between memory-consumption and efficiency.

The port of μIP we did to our reference target (*c.f.* Table 1 allows us to precisely measure its Web server total memory-consumption (including server, TCP/IP stack and device drivers).

μIP memory footprint in persistent memory is of 16.6 KB (15.3 KB of code, 1.3 KB of read-only data).

Global variables need from 1 KB to 3.6 KB, depending on the maximal supported MSS (from 200 to 1460 bytes, the classical TCP MSS bounds). To know the memory space needed for local variables and procedure calls, we initialize the whole stack with a particular value. After executing μIP Web server, we scan the stack to know the maximal stack size used. We measured a maximal stack consumption of 204 bytes.

μIP presents several drawbacks: it is limited to one in-flight packet and it sees its memory consumption growing with the used MSS. We talk about μIP other benefits and drawbacks in the next sections.

4.2 MiniWeb analysis

4.2.1 Server presentation

MiniWeb is a functionally minimal embedded Web server, including its own dedicated TCP/IP stack. Like μIP server, it cannot handle HTTP persistent connections. It also does not support simultaneous TCP connections. When receiving a GET request, miniWeb does not read any byte of its contents but it selects a Web page to be served depending only on the TCP host port. MiniWeb is unable to send dynamically generated contents, making it unusable as a Web application sever.

We however choose miniWeb as one of our reference Web servers because its memory consumption is really small, and because of its interesting implementation choices.

4.2.2 Pre-calculations

As μIP Web server, miniWeb embeds Web pages thanks to a pre-compilation phase. More than a simple content-integration tool, the pre-compiler makes a lot of off-line pre-calculations. IP packets are fully pre-generated, including IP, TCP and HTTP headers. For each file to serve, a set of packets is ready to be sent in the proper order, including TCP connection establishment and closing. This allows huge simplifications in the TCP/IP stack, where the TCP state machine does not have to be taken into account anymore.

Optimizations are also proposed for TCP/IP checksums calculations, a critical aspect of TCP/IP stack in term of computation time [12]. While the IP checksum is computed on the IP header, the TCP checksum is computed on

a part of the IP header, on the TCP header, and on the whole TCP segment. TCP checksums are placed in the packet header, imposing to compute on every data of the packet before starting to send them (thus reading every data twice).

When packets are off-line pre-generated, Both TCP and IP checksums are calculated and included. Only a few parts of them are unknown at compile-time (IP destination address, TCP destination port, TCP sequence number, etc.), and completed at run-time by miniWeb.

This packets pre-calculations method however has a drawback: it imposes to choose a static MSS at compile-time. A TCP host must be able to handle any MSS size proposed by a client (during the TCP connection establishment, the smallest MSS proposed by the two hosts is chosen). This point forces miniWeb to always use the smallest TCP MSS authorized, which is of 200 bytes. As shown in Section 3.3, this involves a significant throughput degradation.

4.2.3 Retransmissions

Like μIP, miniWeb does not store unacknowledged sent packets. MiniWeb TCP/IP stack has the huge advantage to be specialized for HTTP. This allows to have no limitation but that the receiver window on in-flight packets: when a packet is lost, miniWeb can easily find and send it again thanks to its sequence number (because every packet is off-line pre-calculated, including TCP sequence numbers). In miniWeb, TCP retransmission involves no memory overhead nor any limitation on output in-flight packets.

4.2.4 Memory consumption

MiniWeb drivers interface allows to read and write data with a granularity of a single byte. Its device drivers can work without any buffer, or with a MSS-independent sized buffer, thus saving a lot of memory. When used without any buffer, incoming bytes are computed and discarded by miniWeb as soon as they are received.

This point helps to reduce miniWeb global variables usage to 88 bytes of memory. Only 268 bytes of stack are used by local variables and procedure calls. MiniWeb persistent memory footprint is of 7.1 KB (6.4 KB of code, 0.6 KB of read-only data). As we presented it, miniWeb is a memory lightweight Web server. We use it as our low bound reference, for its small memory usage as well as for its reduced functionalities.

5 Propositions and measurements

As shown in Section 1.2, an embedded Web server used for Web applications service has to be efficient when serving *large* static contents as well as *small* dynamic contents. In this section, we present and we evaluate new propositions to avoid μIP server and miniWeb weaknesses. We implemented all our propositions in our own embedded Web server, called *dynaWeb*. DynaWeb, as miniWeb, is a monolithic embedded Web server, using its own dedicated TCP/IP stack. Performances and ressources consumptions are compared be-

tween μIP server, miniWeb and dynaWeb, in various conditions. To make our experiments reproducible, we have made dynaWeb source code available [6].

5.1 Static contents service

5.1.1 Retransmissions management

TCP retransmission mechanism forces a TCP sender to be able to retransmit unacknowledged segments. However, storing every in-flight segments involves a huge memory overhead.

As said in Section 4, μIP does not store unacknowledged packets but it is limited to one in-flight packet. MiniWeb makes the same memory saving without any limitation on in-flight packets, because its TCP/IP stack is specialized for HTTP. DynaWeb uses the same strategy as miniWeb (because it TCP/IP stack is also dedicated to HTTP), having no need to store outgoing packets nor any limitation on its maximal in-flight packets number.

5.1.2 Problematic

It is has been shown [2] that checksum calculation is a critical part of TCP/IP stacks in terms of processing, because it involves every data to be accessed twice. Moreover, as mentioned in Section 1.2, Web applications often send large static contents, thus increasing checksum calculation cost.

Nothing is proposed in μIP to accelerate TCP checksum calculation. MiniWeb uses fully pre-calculated TCP/IP packets, drastically reducing checksum calculation time. Figure 4 shows μIP server and miniWeb maximal applicative throughput on a 14.4 KB/s line. MiniWeb is constrained in always using a MSS of 200 bytes, while μIP has no more than one in-flight packets, thus having $\alpha = 1$ and $\beta = 1$. Because of theirs respective choices, these Web servers implementations for small embedded devices use TCP/IP in an suboptimal way

5.1.3 Proposition

Our proposition consists in pre-calculating checksum by constant-sized chunks of data (let CS be this size). Packets are not fully pre-generated, allowing to handle any MSS proposed by a client. To be as efficient as possible, a common TCP/IP header is partially pre-generated (containing all common and fixed header fields), including partially-calculated checksums. HTTP header are also off-line generated and checksummed.

The pre-generated HTTP header and the file contents are placed contiguously in a persistent memory, as well as the files chunks checksums. This method involves a persistent memory overhead of:

$$chunks\ checksums\ size = 2 \times \left\lceil \frac{file\ size}{CS} \right\rceil$$

When file contents have to be send, the total packet checksum is calculated by summing successive chunks checksums. The output size is always a multiple of the chunks size:

$$maximal\ output\ size = CS \times \left\lceil \frac{MSS}{CS} \right\rceil$$

[6] DynaWeb source code: `http://www2.lifl.fr/~duquenno/Research/DynaWeb`

The chunks size have to be chosen big enough to reduce the computation time, but little enough to allow an efficient MSS adaptation. It is quite obvious that the total checksum computation time decreases logarithmically while the chunks size grows, allowing a great efficiency even when using relatively small chunks sizes.

Figure 4 shows how our server, dynaWeb, have a better maximal theoretical applicative throughput than μIP and miniWeb, because it can handle any MSS and have several in-flight packets.

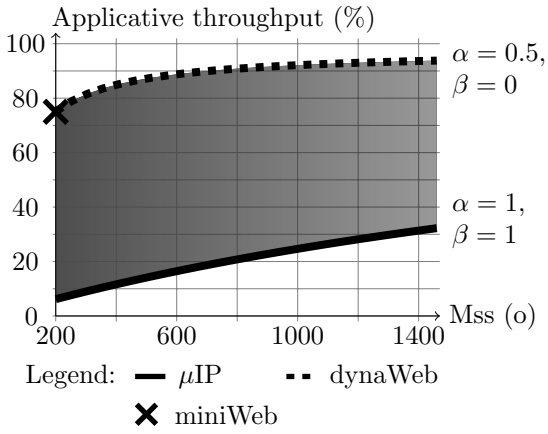


Figure 4: Maximal applicative throughput on a 14.4 KB/s line for μIP server, miniWeb and dynaWeb

5.1.4 Performances evaluation

Checksum pre-calculation performances We measured the time spent by dynaWeb (used on reference target presented in Table 1) in different phases when sending TCP data to evaluate the checksum calculation impact on performances. Measurement are done without checksums pre-calculation.

Table 2 shows the time spent in four phases: IP and TCP headers calculation and sending, TCP segment checksum calculation and TCP segment data sending. The size of the TCP segment is 1408 bytes (a multiple of 128 bytes, to allow comparison in same conditions when a chunks size up to 128 bytes is used). These measurements only show the pure processing time, independently of the device drivers and network constraints. Network driver functions calls have been substituted by void function calls, thus taking into account calls overheads in measurements[7].

Checksum calculation represents 30 % of the total computation time. The two most important phases consist in TCP segment checksum calculation and TCP segment data sending. Even without calling the device drivers, the data sending phase is quite heavy, because it needs to call the driver function (here, a

[7]In case of byte-oriented device drivers, procedure calls are done for each byte to send. Buffer-oriented drivers does not need these calls, making even more important checksum calculation time in ratio.

Processing phase	Time
IP header	0.3 ms
TCP header	0.4 ms
Segment checksum	3.82 ms
Data sending	8.05 ms
Total time	12.5 ms

Table 2: CPU Time spent in different dynaWeb sending phases, without any checksum pre-calculation

void function) for each data byte. The checksum calculation is done by accessing and summing every data by chunk of two bytes.

Table 3 shows the benefits of using our checksum pre-calculation method. For each chunks size (from 4 to 128 bytes), we measured the time needed for checksumming the TCP segment of 1408 bytes. These measurements are compared to our reference performances (see Table 2), obtained without any checksum optimization.

Chunk size	Memory cost	CPU time	Time saved	Total time saved
-	0 B	3.82 ms	0 %	0 %
4 B	704 B	1.48 ms	61 %	19 %
8 B	352 B	0.78 ms	80 %	24 %
16 B	176 B	0.39 ms	90 %	27 %
32 B	88 B	0.23 ms	94 %	29 %
64 B	44 B	0.16 ms	96 %	29 %
128 B	22 B	0.11 ms	97 %	29 %

Table 3: Checksum pre-calculation impact. *Memory cost* is the amount of persistent memory needed to store pre-calculated checksums. *Time saved* is relative to the unoptimized checksum processing time, while *total time saved* is relative to the unoptimized total sending processing time.

Checksum calculation time as well as persistent memory overhead decrease logarithmically when chunks sizes are growing. With a chunks size of 128 bytes, only 0.11 ms are spent for checksum calculation, saving 97 % of the checksum processing time. In this situation, total processing time spent in the sending phase is of 8.91 ms (saving 29 % of the total time without checksum pre-calculation).

Servers comparison We have ported μIP, miniWeb and dynaWeb to our reference target. We use its integrated serial line to its maximal throughput: 14.4 KB/s, using SLIP as link layer protocol.

For our experiments, we used a workstation using Windows XP as operating system, and Internet Explorer 6 as Web browser, the very commonly used configuration used by clients for Internet Web accesses [21, 22]. It is important to note that Windows TCP/IP stack implements the TCP delayed ACKs strategy. Because our workstation is directly connected to the serial line, the network physical latency is negligible.

To evaluate the 3 servers performances on large static files service, we measured the time they need to send a 55.9 KB file. Different MSS sizes are tested, from the smallest authorized (*i.e.*, 200 bytes) to 1460 bytes (the most common TCP MSS). Figure 5 shows our measurements for all our MSS and servers. DynaWeb uses pre-calculated checksums with a chunks size of 32 bytes.

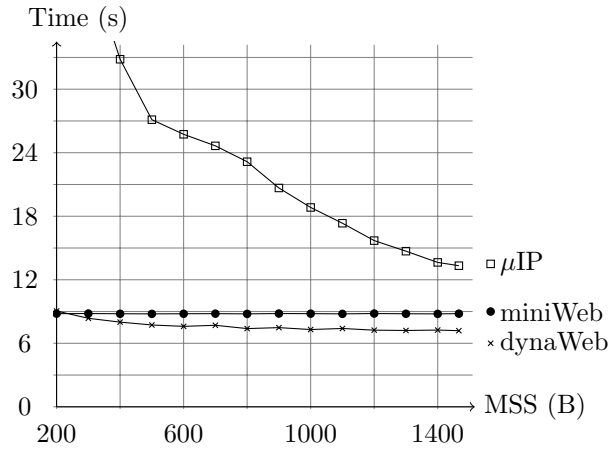


Figure 5: Time needed to send a 55.9 KB file on a 14.4 KB/s serial line for our three servers

The most obvious observation is that μIP server is significantly slower than the two other servers. This can be explained by its limitation to one in-flight packet: μIP cannot send more than 5 packets by second to a TCP/IP stack implementing the delayed ACKs strategy. When a small MSS is used, the total amount of packets to send is the bigger, and μIP needs around 60 seconds to send the whole file. With an MSS of 1460 bytes, the file is served in 13.3 seconds. It is important to keep in mind that in this situation, μIP has a bigger memory consumption (Section 4) because it uses two MSS-sized buffers.

MiniWeb is significantly faster than μIP, and its performances are fully independent on the MSS used. It serves the file in around 8.8 seconds.

DynaWeb has the greatest performances, it is able to serve the file in 7.2 seconds (18% faster than miniWeb). With the very smallest MSS (200 bytes), miniWeb is only 2% faster than dynaWeb (which needs 9 seconds to serve the file). That shows that the overhead of dynaWeb MSS adaptability is small.

Measurements presented in Figure 5 are in adequacy with theoretical applicative throughputs shown in Figure 4.

5.2 Dynamic contents service

Web applications often ask their Web server to dynamically generate and send data. We mentioned in Section 1.2 that an embedded Web server has to be efficient particularly when sending *small-sized* dynamic contents.

5.2.1 MiniWeb and μIP server approaches

MiniWeb is simple and functionally minimal. Actually, the URL requested by the a HTTP GET is not processed and the TCP port (default 80 for HTTP) is used to select the response (port 80 serves page1.html, port 81 replies page2.html ...). Nothing is available to serve dynamic content.

In another hand, μIP server, used on its general purpose embedded TCP/IP stack, is able to generate contents dynamically. It uses dedicated markers in HTML files, associated to generation functions. These functions have to be directly implemented in the server source code.

μIP applications must be developed using protothreads. This choice makes applications hard to write (local variables are not allowed, as well as switch-case control structures). Dynamic contents generation functions also suffer of this limitation.

For memory saving reasons, μIP Web server uses a very simple approach for sending generated contents: instead of building a whole HTTP response, it sends a first part of the response containing the HTTP return code, followed by a second packet containing the content-type, followed by the HTTP generated contents.

5.2.2 Our approach

Client OS	Web Server	Initial loading (ms)	Service time (ms)
Windows, IE 6	μIP server	1728	736
	dynaWeb	709	62
Linux, Firefox 2	μIP server	866	183
	dynaWeb	694	64

Table 4: Servers performances on the AJAX test application (MSS: 1460 B)

When a Web server has to return a response of a few bits, protocols constant overheads ratios becomes huge. We propose to reduce these overheads in two ways:

Checksums We pre-generate offline HTTP headers for dynamic contents, with their pre-computed checksums. At runtime, generated contents have to be checksummed, but IP, TCP and HTTP headers and their checksums are statically available.

TCP control We allow to use HTTP persistent connections, avoiding TCP connections establishment and closing each time we have to send small data. Embedded Web servers often do not use persistent connection for memory saving reasons (only one TCP connection is used at a given time). In dynaWeb, a TCP connection needs only 36 bytes of memory because data buffers and common informations (*e.g.*, TCP source port, IP source address, etc.) are shared by all connections.

Our contents generation model simply consists in URLs mapped to native functions instead of static contents. This approach is more adapted to small

contents service than the HTML markers approach. AJAX Web applications can easily integrate small HTTP contents into complete Web pages. The generation function simply has to write data into a global buffer, using a *write* procedure. Two situations are possible:

Small-sized data When generated data can be contained into the global buffer (which maximal size is the TCP MSS), we use HTTP persistent connection, including the content-length field in the header. In this case, generated contents transmission is very efficient.

Large-sized data When the global buffer is full, and the contents generation has not ended, the HTTP contents length is unknown. One can send the beginning of the HTTP response, without the content-length field. In this situation, a persistent connection cannot be used. The end of the response is identified by a TCP connection close. To enable eventual retransmissions, the stack is limited in this case to one in-flight packet, because a single global buffer is used. Each time an acknowledgment is received, the buffer is written again by the contents generation function, allowing a new transmission. This involves huge overheads, but large-sized generated contents are quite infrequent in AJAX Web applications.

In dynaWeb, dynamic contents generators are independent source code files, compiled separately and linked with the server before being embedded. A tool generates a table associating URLs to static (files data and chunks checksums) or dynamic (functions references) contents. A Web application is fully contained in a directory, including static files and content generation source codes.

5.2.3 Performances evaluation

We measured μIP and dynaWeb performances on a simple AJAX application. MiniWeb does not appear in these experiments, because of its incapacity to serve generated contents.

The Web application is a simple Web page sending periodically asynchronous HTTP requests to the server. The server is embedded into a portable console which characteristics have been presented in Table 1. A generator function is used to return the console buttons state. The application displays and update continuously the console buttons state, thanks to AJAX requests.

The application consists in a HTML file of 1.9 KB and JavaScript file of 1.7 KB. The generated contents containing buttons state is periodically served in two bytes.

We tested the two servers when accessed from different kind of clients. The first client configuration consists in a workstation using Windows XP as OS and Internet Explorer 6 as Web browser. The second client uses a Linux kernel and Mozilla Firefox 2 as Web browser. It is important to note that Linux TCP/IP stack does not use the TCP delayed ACKs strategy for the firsts exchanged segments of TCP connections. Table 4 shows the measured performances (using a 14.4 KB/s serial line).

The client OS does not significantly impact on dynaWeb performances, because it sends more than one in-flight packet. All the results are in favor of dynaWeb.

With a client using Windows, μIP is extremely slow, because of its in-flight packets limitation. It needs more than 700 ms to send the two generated bytes for two reasons: (i) it sends at least three TCP segments for each HTTP response and (ii) it uses non-persistent HTTP connections.

With a client using Linux TCP/IP stack (without delayed ACKs on these short connections), dynaWeb is three times faster than μIP for sending the two generated bytes. This is because dynaWeb uses persistent HTTP connections, it sends its response in only one TCP segment, and because its IP, TCP and HTTP headers are generated and checksummed offline.

5.2.4 Discussion on dynamic contents service

Embedded Web servers application often need to generate contents. Informations returned by this way may have different persistence properties, changing the optimal TCP retransmission management in case of a loss:

- The generation function has side effects. In case of retransmission, the TCP/IP stack must not re-generate the contents to send. The generation function must be called only once and its result has to be stored while data are not fully sent and acknowledged. A typical example of this situation is a function that updates client account values.

- The generation function is idempotent (deterministic and without side effects). In this case, the TCP/IP stack can equally re-generate the contents. Depending on its priority (memory saving or efficiency), the stack can choose by itself between two policies: (i) calling only once the generation function and storing its results or (ii) re-calling the function in case of retransmission. As an example, a function that encrypts a data transmitted in the request may be idempotent.

- The generation function has no side effects and generated contents have to be as recent as possible. In this case, when a retransmission is needed, the best choice consists in re-generating the contents (a significant amount of time can be spent between the first send and a retransmission need notification). A function that returns a sample from a sensor is a good example of such a situation.

The choice to cache or not the generated contents has an impact on processing time and memory consumption, and furthermore on the TCP traffic. Indeed, cases allowing to discard generated contents after having sent them permids to send large generated contents without waiting for incoming acknowledgments, thus having several in-flight packets (as explained in Section 5.2.2).

Proposing a new API or API extensions allowing the Web application programmer to give information about its generated contents persistence should be a big improvement for dynamic contents management in embedded Web servers. Indeed, in this context, choices between memory saving and efficiency have a crucial impact (more than in classical workstation context).

5.3 DynaWeb global analysis

5.3.1 Functionalities summary

DynaWeb uses a similar device drivers interface to miniWeb: it can handle incoming packets byte per byte, without any input buffer or with a MSS-independent sized buffer. Its implementation is monolithic, *i.e.*, including its own dedicated TCP/IP stack. Table 5 summarizes the three servers functionalities.

Functionality	μIP	miniWeb	dynaWeb
UDP	–	–	–
ICMP	√	–	–
IP options	–	–	–
IP reassembly	√	–	–
TCP options	√	–	–
TCP urgent data	√	–	–
TCP RTT estimation	√	–	–
POST requests	–	–	–
URL readding	√	–	√
TCP MSS adaptation	√	–	√
Multiple TCP conn.	√	–	√
Multiple in-flight packets	–	√	√
HTTP persistent conn.	–	–	√
Contents generation	√	–	√
Checksum pre-calculation	–	√	√

Table 5: μIP server, miniWeb and dynaWeb functionalities summary

Server	Volatile memory (B)			Persistent memory (B)		
	Globals	Stack	Total	Code	RO data	Total
μIP (MSS: 200 B)	1 k	204	**1.2 k**	15.3 k	1.3 k	**16.6 k**
μIP (MSS: 1460 B)	3.6 k	”	**3.8 k**	”	”	**16.6 k**
miniWeb (any MSS)	88	268	**356**	6.5 k	648	**7.1 k**
dynaWeb (any MSS)	152	92	**244**	8.9 k	304	**9.2 k**

Table 6: Servers minimal memory consumptions

μIP has a better support of TCP/IP than miniWeb and dynaWeb TCP/IP stacks. Indeed, the last two have been specialized for their specific usage. As examples, ICMP or TCP urgent data have no need to be supported by an embedded Web server.

IP reassembly has not been implemented in dynaWeb (nor miniWeb) because it is quite infrequent [18] and its management involves a big memory consumption. DynaWeb is unable to handle other TCP options than the MSS negotiation, and it does not estimate the connection RTT (Round Time Trip).

TCP RTT estimation allows to use a suitable retransmission timeout for a connection. DynaWeb uses an empirically-chosen constant timeout.

DynaWeb supports critical HTTP points. It is able to handle multiple connections, several MSS sizes, HTTP persistent connection and contents generation while managing multiple in-flight packets and pre-calculating its TCP and IP checksums. Such HTTP support allows dynaWeb to have the best performances, keeping its memory consumption extremely low (see Section 5.3.2).

The three servers does not support HTTP POST request. Including this support in dynaWeb should be intersting, allowing a user to easily upload new applications or large-sized data onto the server.

5.3.2 Memory consumption

As μIP and miniWeb, dynaWeb main priority consists in memory saving. Table 6 presents the minimal memory consumptions of the three embedded Web servers, including theirs TCP/IP stack and device drivers. MiniWeb and dynaWeb use an input buffer of only one byte. μIP uses its smaller input buffer size, 240 bytes, fixing the TCP MSS to 200 bytes. μIP and dynaWeb maximal connection number is set to one (note that miniWeb always suffer of this limitation).

μIP Web server needs more memory than the two monolithic Web servers, miniWeb and dynaWeb. Its minimal volatile memory consumption of 1.2 KB forces it to use a TCP MSS of 200 bytes, involving extremely poor performances (see Section 5.1.4). Its persistent memory footprint is also significantly higher than miniWeb and dynaWeb ones.

MiniWeb and dynaWeb have comparable memory consumptions: they need a few hundreds of bytes of volatile memory, and a few kilo-bytes of persistent memory.

μIP Web server presents the worst performances and the higher memory footprint. This is the illustration of 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.

It is important to remember that miniWeb is unable to send generated contents, to handle multiple TCP connection and to read HTTP requests contents. DynaWeb, with its very small memory footprint (less than 300 bytes of RAM), is able to serve dynamic Web applications as described in Section 1.2 (multiple and persistent TCP connections, contents generation, etc.).

6 Future works

In a near future, we would like to treat various topics related to embedded Web servers.

Large-sized dynamic contents service We would like to evaluate the costs and benefits of different strategies allowing large contents generation and service in dynaWeb. A strategy we think relevant has been described in Section 5.2.2.

Nevertheless this strategy has to be discussed and other solutions must also be evaluated.

Non point-to-point communications We would like to extend our work to remote communications. This situation involves new constraints (higher latency, more packet losses) and can be used for example in the case of sensor networks (allowing samples consultation from a Web browser).

TCP connection establishment Allowing our server to establish an (eventually secured) TCP connection to another host should also be relevant, for example for mashup applications (where served informations are merged from different sources).

Reverse AJAX integration A work on *reverse AJAX* with embedded constraints should also be interesting. This application model [17] allows the Web server to push data to the client instead of being polled, but it is very memory consuming (numerous TCP connections have to be stored simultaneously). Reverse AJAX dedicated APIs have already been proposed for classical (heavy) servers, and bring some scalability issues [13].

Dedicated API Finally, we would like to work on Web applications conception, providing, as an example, new APIs dedicated to embedded Web servers constraints. In Section 5.2.4, it is mentioned that informations about generated contents persistence should be exploited by an embedded Web server.

7 Conclusions

We presented an analysis of embedded Web servers performances for dynamic Web application service. We shown that this kind of application mainly serves two types of contents: (i) *large* static files and (ii) *small* generated contents. To provide an efficient Web application service, embedded Web servers have to be particularly efficient when sending these two kind of contents.

Our study identifies factors that have a big impact on embedded Web servers performances: correct TCP MSS handling, multiple in-flight packets management in front of delayed ACKs policies, checksum and headers pre-calculations, multiple and persistent TCP connections handling. Our measurements on existing embedded Web servers confirm by experimentation the importance of the factors.

We show that tightly coupled protocols implementation permits numerous optimizations, in term of memory footprint as well as in computation time and network efficiency. A monolithic implementation and cross-layer approach allows to consider new solutions. Our implementation, dynaWeb, is compared to other existing Web servers. DynaWeb shows by experiment that our proposals improve significantly Web server performances, keeping an extremely low memory consumption and optimal functionalities for dynamic Web application service.

References

[1] I. Agranat. Engineering web technologies for embedded applications. *Internet Computing, IEEE*, 2(3):40–45, May-June 1998.

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

[3] R. Braden. Rfc 1122: Requirements for internet hosts - communication layers, 1989.

[4] M. Domingues. A simple architecture for embedded web servers. *ICCA'03*, 2003.

[5] 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.

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

[7] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM Press.

[8] A. Dunkels, T. Voigt, and J. Alonso. Making tcp/ip viable for wireless sensor networks. In *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN 2004), work-in-progress session*, Berlin, Germany, Jan. 2004.

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

[10] G.-j. Han, H. Zhao, J.-d. Wang, T. Lin, and J.-y. Wang. Webit: a minimum and efficient internet server for non-pc devices. In *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE*, volume 5, pages 2928–2931 vol.5, 2003.

[11] H.-T. Ju, M.-J. Choi, and J. W. Hong. An efficient and lightweight embedded web server for web-based network element management. *Int. J. Netw. Manag.*, 10(5):261–275, 2000.

[12] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in tcp/ip. In *SIGCOMM '93: Conference proceedings on Communications architectures, protocols and applications*, pages 259–268, New York, NY, USA, 1993. ACM.

[13] E. B. A. Mesbah and A. van Deursen. A comparison of push and pull techniques for ajax. In S. uang and M. D. Penta, editors, *Proceedings of the 9th IEEE International Symposium on Web Site Evolution (WSE)*, pages 15–22. IEEE Computer Society, 2007.

[14] J. Postel. Rfc 791: Internet protocol, Sept. 1981.

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

[16] J. Riihijarvi, P. Mahonen, M. Saaranen, J. Roivainen, and J.-P. Soininen. Providing network connectivity for small appliances: a functionally minimized embedded web server. *Communications Magazine, IEEE*, 39(10):74–79, Oct. 2001.

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

[18] C. Shannon, D. Moore, and K. C. Claffy. Beyond folklore: observations on fragmented traffic. *IEEE/ACM Trans. Netw.*, 10(6):709–720, December 2002.

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

[20] R. Sridharan, R. Sridhar, and S. Mishra. Poster: A robust header compression technique for wireless ad hoc networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 7(3):23–24, 2003.

[21] W3Schools. World wide web browsers statistics. `http://www.w3schools.com/browsers/browsers_os.asp`.

[22] W3Schools. World wide web clients os statistics. `http://www.w3schools.com/browsers/browsers_stats.asp`.

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803