Serving Embedded Content via Web Applications: Model, Design and Experimentation

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

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

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

ABSTRACT

Embedded systems such as smart cards or sensors are now widespread, but are often closed systems, only accessed via dedicated terminals. A new trend consists in embedding Web servers in small devices, making both access and application development easier. In this paper, we propose a TCP performance model in the context of embedded Web servers, and we introduce a taxonomy of the contents possibly served by Web applications. The main idea of this paper is to adapt the communication stack behavior to application contents properties. We propose a strategies set fitting with each type of content. The model allows to evaluate the benefits of our strategies in terms of time and memory charge. By implementing a real use case on a smart card, we measure the benefits of our proposals and validate our model. Our prototype, called *Smews*, makes a gap with state of the art solutions both in terms of performance and memory charge.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Network communications*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Network operating systems*; C.2.6 [**Computer-Communication Networks**]: Internetworking

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Modeling, Embedded Web server, Applicative model

1. INTRODUCTION

Embedded systems around us such as smart cards or sensors are mainly closed systems, only able to communicate with dedicated terminals. A new trend called *Web of Things* [30, 14, 13] consists in embedding Web servers in highly constrained devices, allowing the service of dynamic Web applications. The device should embed a TCP/IP stack with the Web server in order to be as autonomous and accessible as possible. Such device can be accessed from any computer (or smart phone, PDA...) simply using a standard Web browser, thus avoiding any client software development and deployment. This allows new usages, where personal devices and Internet services can easily interact. Furthermore, the development of Web applications is already widespread and well-known by companies and engineers. This makes more accessible the programming of tiny devices.

Smart cards are a great example of widespread devices: around 5 billions of devices have been released in 2008 [1] (in comparison with 0.15 billion laptops [2]). Smart Card Web Server [22] is an industrial standard that allows the usage of Web servers in smart cards. With such an embedded Web server, one can access and manage its SIM card via a Web application thanks to the Web browser of its smart phone. By using Web mashup [20, 14, 13], private information stored on the SIM card (*e.g.*, contacts book, last operations) may be merged with information that are only available on the Internet (*e.g.*, telephone company special operations or various on-line service).

Providing a Web access to widespread devices is a great challenge for which we provide solutions in this paper. When working with highly-constrained devices (typical smart cards are based on an 8 bit CPU at a few MHz with a cryptographic co-processor, have around 1 kB of RAM and 16 kB of addressable EEPROM for code), usual solutions based on a Web server over an operating system (*e.g.*, Apache/Linux, IIS/Windows) are almost impracticable. Using a Web server thought as the operating system itself is more suitable. This also makes possible dedicated cross-layer optimizations. The fact that such Web servers are only accessed by a few clients also contributes in making their usage possible.

The main idea of this paper is to fit the communication stack behavior with the properties of the Web contents being served. To achieve this objective, we describe a taxonomy of Web applicative contents, and we propose a set of strategies for each type of content. We evaluate the relevance of our strategies (i) by proposing a dedicated traffic model and metric and (ii) by conducing experiments on a smart card.

[1]`http://www.researchandmarkets.com/`
[2]`http://www.idctracker.com/`

The paper is organized as follows: Section 2 presents a state of the arts of works about TCP/IP and Web servers for embedded systems. In Section 3, we introduce a dedicated TCP performance model and a metric for evaluating the cost of requests service. In Section 4, we propose a taxonomy of embedded Web contents, followed by a set of strategies for each kind of content. The strategies are designed and analyzed in relation with our model. In, Section 5, we experiment our propositions on smart cards, thus validating our model and strategies. We finally conclude in Section 6.

2. STATE OF THE ART

This section presents a state of the art of works related to TCP/IP stack and Web server design for embedded devices.

2.1 TCP/IP stacks for embedded systems

It has been shown with TinyTCP [4] and mIP [27] that by implementing a subset of TCP/IP RFCs, communication with common TCP/IP nodes remains possible.

Adam Dunkels proposed two embedded TCP/IP stacks [6]. LwIP is fully RCS-compliant and uses a layered protocol implementation (including IP, TCP, UDP and ICMP). uIP implements a subset of TCP/IP with a monolithic approach and it uses less memory than lwIP.

Because of the small size of uIP we selected this embedded TCP/IP stack as our first reference for the experiments conduced in Section 5 (uIP source code is provided including a small size Web server).

uIP is a tiny event-driven TCP/IP stack, which implementation is based on protothreads [8]. Protothreads provide a blocking abstraction with extremely low memory requirements (2 or 3 bytes per protothread), without runtime stack allocation.

The Web server of uIP is able to serve static files as well as dynamic contents. Files are pre-processed 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.

2.2 Embedded Web servers

Numerous works [1, 16, 26, 28, 15, 5, 19, 25] prove that it is possible to embed a Web server into constrained devices. Such Web servers are not executed in user-space over on operating system: they are designed as the operating system by itself. They use their own dedicated TCP/IP stack instead of usual OS abstractions (*e.g.*, Berkeley sockets). As a consequence, they can be optimized at every software level of the system.

Some of these servers provide a simple way to serve dynamic contents. Most of them make use of a dedicated tool in order to pre-process the Web contents to be served. These tools are build on a same scheme: from static files (HTML, CSS, Javascript, pictures, etc) and dynamic contents generators (*c* source code or enriched HTML), they produce *c* source files. These files are then compiled and linked with the Web server engine. The resulting binary file can directly be run by the device.

The proof-of-concept Miniweb server [7] particularly drew 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 Web server.

When Miniweb receives a packet, it computes incoming bytes and discards them as one goes along. Its device drivers can work with a buffer which size is independent of the TCP MSS[3], or with no buffer. This helps Miniweb to reduce its memory consumption to only a few hundreds of bytes.

In Miniweb, IP packets are entirely pre-generated, including IP, TCP and HTTP headers and checksums. For each file to serve, a set of packets is ready to be sent in the proper order, including TCP handshakes. This allows huge simplifications in the TCP/IP stack, where the TCP state machine does not have to be implemented.

Because of its extremely low memory consumption and its interesting design choices, we choose this server as our second reference for the experiments conduced in Section 5. By using Miniweb and uIP as reference implementations, we will be able to compare respectively a monolithic HTTP/TCP/IP server with a Web server running over a TCP/IP stack.

3. A MODEL FOR LOCALLY ACCESSED AND CONSTRAINED WEB SERVERS

In this section, we present a model of TCP performance dedicated to embedded Web servers accessed on personal area networks. We first propose an analysis of the traffic embedded Web servers have to handle. Then, we describe a fine-grained model for TCP performance in relation with this analysis.

3.1 Preliminary remarks

We present here an analysis of HTTP, TCP and IP protocols when interacting for serving Web applications.

3.1.1 HTTP: a request-response model over TCP

HTTP is based on a request/response scheme where each request uses its own TCP connection (involving a three-ways handshake for opening and a four-ways handshake for closing, thus wasting 5 RTT[4]). In order to allow better interaction with TCP, HTTP 1.1 promotes the usage of *keep-alive* connections, allowing a Web browser to use the same connection for consecutive HTTP requests.

3.1.2 Impact of the TCP delayed acknowledgments

TCP delayed acknowledgments [3] is a policy used to reduce the traffic caused by void acknowledgments (with no payload). It is implemented by most of the desktop OS TCP/IP stacks (*e.g.*, Windows and MacOS). A TCP host implementing TCP delayed ACKs only acknowledges a segment (i) 200 ms after having received it or (ii) when a second segment is received.

Many embedded TCP/IP stacks do not support more than one in-flight TCP segment[5], because this allows very lightweight and simple TCP implementations. As mentioned in [6, 25], with such a strategy, the 200ms delay will always be triggered, bounding the maximal sending rate to 5 packets per seconds.

[3]MSS stands for TCP Maximal Segment Size
[4]RTT stands for Round Trip Time
[5]In-flight segments are segments which have been sent but have not yet been acknowledged

3.1.3 Typical Web applications traffic

The widely used AJAX[12] methodology allows to design highly interactive applications with an efficient task repartition between the client and the server. In a previous work [9], we characterized the traffic produced by some widely used Internet AJAX applications.

During a first phase, the client (*i.e.*, the browser) collects several static files, containing style sheets (CSS), contents (XHTML), and applicative code (JavaScript). Then, it executes the applicative code downloaded in the first phase, and interacts with the server by sending asynchronous requests, allowing the Web application to run in the client browser. We observed that static files are mainly bigger (average size about 8 kB) than dynamic content (average size about 0.6 kB).

3.2 A dedicated TCP/IP performance model

Great care has been given to TCP modeling in the literature. The objective is mainly to characterize the sending rate of a bulk TCP flow as a function of packet loss and RTT [2, 18, 21, 23, 24]. The complexity of these models is due to the congestion control mechanism (Tahoe, Reno, etc. [29, 11]) and to the network congestion modeling. The most precise results have been reached in [2], in which the authors describe a rich model for network congestion.

3.2.1 Model presentation

We propose a TCP/IP performance model adapted to embedded devices access through a local network. The granularity of our model is the byte, and, instead of approximating asymptotic sending rates (where data of an infinite size are sent), it estimates the time needed by a host to send a content of a bounded size.

In this context, we assume that the server will not cause any congestion, because it does not generate heavy traffics in comparison with the local network capacity. The output window is limited by the server-side memory, because each TCP in-flight segment has to be kept in memory. In practice, the advertised window announced by Windows and Linux is respectively around 8 kB and 5 kB, which is greater than the available RAM in many tiny devices.

We describe the case where the delayed acknowledgements policy is used (the most common situation, because both Windows and MacOS make use of this policy). We suppose the link layer to be half-duplex, what is the true for most of the protocols supported by embedded devices (*e.g.*, USB, ZigBee, APDU, Wifi, Ethernet).

3.2.2 Model definition

Our model defines $ST(cs, wnd, n_{max})$ where ST is the sending time as function of cs (the size of the content to send), wnd (the maximal number of in-flight bytes) and n_{max} (the maximal number of in-flight segments). We note BW the maximal link bandwidth between the two hosts, while the round trip time is noted RTT.

We note $t(n, s)$ the time needed to send s bytes in n TCP/IP packets. Both TCP and IP headers are of 20 bytes (when used without option, the most common case):

$$t(s,n) = \frac{40 \times n + s}{BW} \tag{1}$$

We note T the time spent for sending cs bytes in n TCP segments and for receiving every TCP acknowledgment. Because of the delayed ACKs policy, the number of void acknowledgements received is $\lceil \frac{n}{2} \rceil$. If the number of segments is odd, the acknowledgement for the last segment will be delayed and sent 200 ms after the reception of the segment:

$$T(s,n) = \begin{cases} t(s,n) + t\left(0, \lceil \frac{n}{2} \rceil\right) + RTT & \text{if } n \text{ is even} \\ t(s,n) + t\left(0, \lceil \frac{n}{2} \rceil\right) + RTT + 0.2 & \text{otherwise} \end{cases} \tag{2}$$

When a host sends a content of cs bytes, it first sends d times wnd bytes. wnd is the maximal number of sent but unacknowledged bytes (equals to the sum of in-flight segments sizes). The value of d is:

$$d = \left\lfloor \frac{cs}{wnd} \right\rfloor \tag{3}$$

These $d \times wnd$ bytes are sent in n packets. We suppose that, in order to avoid odd number of in-flight packets (for good interaction with delayed ACKs), the TCP stack chooses an even value for n, when possible. We define a function called $toEven$ in order to simplify our formulas:

$$toEven(a) = \begin{cases} a & \text{if } n \text{ is even} \\ a+1 & \text{otherwise} \end{cases} \tag{4}$$

Let n be the number of TCP segments used to fill each window. MSS is the TCP Maximum Segment Size.

$$n_1 = min\left(n_{max}, toEven\left(\left\lceil \frac{wnd}{MSS} \right\rceil\right)\right) \tag{5}$$

Ones $d \times wnd$ bytes have been sent, r bytes remain to send:

$$r = cs \mod wnd \tag{6}$$

The r remaining bytes are sent in n_2 TCP segments:

$$n_2 = min\left(n_{max}, toEven\left(\left\lceil \frac{r}{MSS} \right\rceil\right)\right) \tag{7}$$

Finally, the sending time ST can be calculated as:

$$ST(cs, wnd, n_{max}) = d \times T(wnd, n_1) + T(r, n_2) \tag{8}$$

Figure 1 synthesizes ST by drawing the time needed to send a content of a given size. Each curve represents a different value for n_{max}. The MSS is set to 1460 bytes[6], the bandwidth is set to 100 kB/s, and the RTT to 50 ms.

The sending time evolves by step, because of the discretization involved by the notion of packets. The curves highlight that poor performances are obtained when having $n_{max} = 1$. Because of the delayed ACKs, all odd values for n_{max} provide very bad performance. For even values, the higher n_{max} is, the faster is the TCP sender.

3.3 Summary and analysis

Most of the software designed for embedded devices use event-driven approaches, thus fitting with highly constrained hardware. Indeed, threaded models waste many memory. Event-driven approaches are extremely 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 stateful (notions of connection, sequence numbers, acknowledgments). We synthesize this analysis by highlighting the three most important properties for the TCP stack of embedded Web servers:

[6]1460 bytes is the most common TCP MSS, because the MTU of Ethernet is 1500 bytes

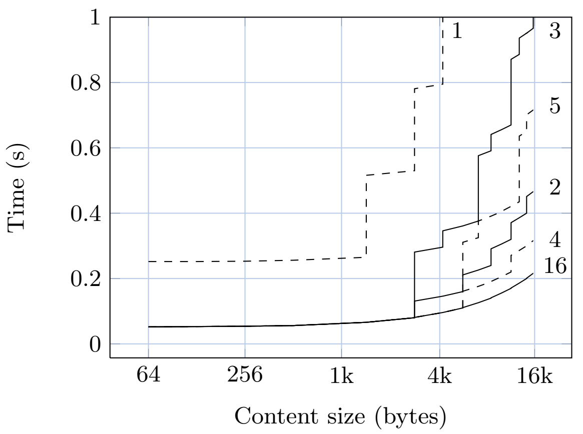

Figure 1: Sending time as a function of the content size. Each curve represents a different maximal number of in-flight segments.

Support of multiple in-flight packets Handling multiple in-flight TCP segments is important because the delayed ACKs policy is widespread. It is a stateful property that involves to keep several segments in memory, thus wasting many RAM.

HTTP persistent-connections handling HTTP persistent connections provide better performance, but require to manage multiple TCP connections simultaneously. It is a stateful mechanism which implementation is expensive in term of RAM and code size.

Large TCP segments management The support of large TCP segments improves the sending rate (as showed by our model), but requires to manage packets, what is hard to do in memory constrained systems. This is a stateless property.

As many embedded TCP/IP stacks, uIP is limited to one in-flight segment. In Miniweb, all the packets are entirely built off-line. It does not store sent packets in RAM, so it handles several in-flight segments. This forces Miniweb to use a constant MSS, which value has to be as small as authorized by TCP (200 bytes) for TCP compliance reasons (when a connection is penned, the MSS is chosen as the smallest value proposed by the hosts). Finally, both uIP's Web server and Miniweb do not handle persistent connections for memory saving reasons.

4. DEDICATING THE COMMUNICATION STACK BEHAVIOR TO APPLICATIVE PROPERTIES

In this section, we show how the communication stack behavior can be dynamically adapted to applicative content properties. We first describe a taxonomy of the contents needed by Web applications. Then we introduce a new metric in order to evaluate the cost involved on a server when serving a request. Based on this metric, we propose communication stack strategies dedicated to each kind of contents. We finally describe a software architecture able to use the strategies we proposed depending on the properties of the Web content being served.

4.1 A taxonomy of Web contents

In order to design efficient solutions for serving Web contents, we introduce a taxonomy of the different kinds of contents embedded Web applications may produce. We identified two classification criterions for contents a Web server may produce. The first criterion (noted A) of our taxonomy is the content size. It is based on two sets:

A1 - Small output Contents that are smaller than the TCP output window (noted *wnd* in Section 3).

A2 - Large output Contents that are larger than *wnd*.

The second criterion (noted B) is the output persistence. We identified five sets for this criterion:

B1 - Pre-defined static contents These contents are fixed off-line. Static files embedded with the server are a good example of this set. Typically, tiny Web servers are only able to serve B1 contents.

B2 - Runtime static contents These contents are fixed only once at runtime, even if sent several times. A good example of such content is a SIM card that manages contacts information: each time a contact is added, new static contents are created.

B3 - Persistent contents generators These contents are generated depending on the current device state (hardware state or software variables). A second call on its generator function will produce a different output or have side effects. Examples of this set are: configuration function, update of a client session, etc.

B4 - Volatile contents generators These contents must be generated as late as possible before they are received by the client. Functions used to send a value sampled from a sensor or a current device state are good example of this set.

B5 - Idempotent contents generators Idempotent functions are deterministic and have no side effect (inspired from Untrusted Deterministic Functions [17]). Given a context, they can be called as many times as needed, always returning the same result with no context modification. Functions that make a calculation on an input are an example of this set.

4.2 The memory charge: a metric for request service cost

In order to evaluate various strategies for Web contents service, we introduce a new metric, called *memory charge*, which aim is to synthesize the cost involved on a server when serving a request.

We take the example of a popular embedded TCP/IP stack, uIP[6]. It is limited to a single in-flight segment which size is bounded by the negotiated MSS.

When serving a static content with the Web server of uIP, the memory consumption is null. In case of a packet loss, the data can be reaccessed from its persistent memory and sent again. The time needed by uIP to send a content of *cs* bytes is (see Section 3.2):

$$T_s = ST(cs, MSS, 1) \tag{9}$$

When serving a dynamic content with the Web server of uIP, all in-flight segments are kept in RAM. During a first phase (a), sent segments have the size of the MSS. During a second phase (b), the remaining part of the data is sent ($cs \mod MSS$ bytes). The total time needed to serve a content of cs bytes is:

$$T_{d_a} = ST(cs - cs \mod MSS, MSS, 1) \quad (10)$$
$$T_{d_b} = ST(cs \mod MSS, MSS, 1) \quad (11)$$
$$T_d = T_{d_a} + T_{d_b} = ST(cs, MSS, 1) \quad (12)$$

These two examples show that the sending time is not a sufficient metric to evaluate a strategy. The amount of memory required while the transmission process carries out is also a critical criterion, but it is harder to evaluate.

We introduce the notion of *memory charge* in order to synthesize information about the memory usage over the time. The memory charge for a given service is the integration of the consumed memory during the sending time. It is expressed in *bytes.seconds* (B.s). It represents the server-side charge involved by a request on a given Web content. With a small memory charge, an embedded Web server is able to run in more constrained (and cheaper) devices, and is more scalable (it will probably not be accessed by thousands of clients, but scalability issues may appear with only a few connections).

When serving a static content with the Web server of uIP, the memory charge (noted Cs) is null because sent segments are never kept into memory. In the second example, the memory charge is:

$$C_d = T_{d_a} \times MSS + T_{d_b} \times (cs \mod MSS) \quad (13)$$

4.3 Handling the contents taxonomy

Here, we show how to serve Web contents efficiently with a cross layer perspective, from the Web application to HTTP, TCP and IP. The goal is to improve the server performance and to reduce its resources consumption. Depending on each set of our taxonomy, we identified five different strategies. The associations between the nature of contents and the strategy we propose are presented in Table 1. We describe here all the strategies.

	B1	B2	B3	B4	B5
A1	S0	S0	S1	S2	S1,S2
A2	S0	S0	S3	S4	S3,S4

Table 1: Strategies applied for each kind of content of our taxonomy

S0 strategy: "pre-processed contents".

This strategy is applied to static contents (B1, B2) embedded in persistent memory (or a cheap read only memory). Their TCP checksums are pre-calculated only once. Data are directly accessed and sent from memory. The current part of the file to be served is retrieved from the current TCP sequence number. In case of packet losses, data to retransmit can be easily re-accessed thanks to the sequence number of the lost segment.

This strategy allows to have no limitation on the number of in-flight packets. The memory charge is null with this strategy. The service time is:

$$T_0 = ST(cs, \infty, \infty) \quad (14)$$

Let α be $\lceil \frac{cs}{MSS} \rceil$. The speed ratio between the uIP approach and this strategy, thanks to trivial calculations, can be approximated by:

$$\frac{T_s}{T_0} \approx \frac{80 + MSS + BW \times (RTT + 0.2)}{60 + MSS + BW \times \frac{RTT}{\alpha}} \quad (15)$$

Because cs, MSS, BW and RTT are positive integers, this ratio is always greater than 1, and grows with α, *i.e.*, when the content size is growing or when the MSS is decreasing. When serving a content of 8 kB (the average content size in the AJAX loading phase, measured in Section 3.1.3) with the reference network settings used in Section 3 (bandwidth of 10000 B/s, RTT of 50 ms, MSS of 1460 bytes), this speed ratio is of 11.2.

S1 strategy: "wait-for-ACK buffer".

This strategy is used for small persistent contents generators (A1/B3). The server engine calls the generator function (written by the application developer), which uses a *outc* routine to send data. It stores data to be sent into a buffer and calculates TCP checksums on-the-fly. When the generator ends its execution, the whole HTTP response is sent and kept in the buffer until it has been acknowledged, thus allowing possible retransmissions.

The usage of a common shared buffer is possible, allowing to serve simultaneously multiple A1/B3 contents while the amount of memory available is sufficient. If not, HTTP requests are suspended until acknowledgments are received.

With S1, the time to send the data is:

$$T_1 = ST(cs, cs, \infty) \quad (16)$$

The speed ratio between the uIP approach and S1:

$$\frac{T_d}{T_1} = \frac{80 + cs + BW \times (RTT + 0.2)}{120 + cs + BW \times RTT} \quad (17)$$

This ratio is greater than one as soon as we have $40 < BW \times 0.2$, *i.e.*, when the bandwidth is greater than 200 bytes/s. When serving a content of 600 bytes (the average content size in the AJAX running phase) with our reference network settings, this speed ratio is of 4.5.

Like the uIP approach, S1 consumes cs bytes during the service. Its memory charge is:

$$C_1 = cs \times T1 \quad (18)$$

Note that the ratio between C_d and C_1 is equal to the ratio between T_d and T_1.

S2 strategy: "volatile buffer".

This strategy is used for small volatile generators (A1/B4). As for S1, the generator function written by the application developer uses *outc* to generate data. When the function ends, the HTTP response is sent and discarded from memory. In case of TCP retransmissions, the engine regenerates the contents by calling again the generator function in order to resend it. This strategy is as fast as S1, so it is faster than uIP approach and its memory charge is null.

S3 strategy: "stateful generators".

This strategy is used for large persistent generators (A2/B3). The generator function (written using the blocking *outc* routine) has to be executed only once per HTTP request. Once the buffer is full (or filled with more than one

MSS), the blocking routine gives back the hand to the TCP stack, which sends one segment (or two, in order to avoid odd in-flight segments number) and continues working normally. The generator function is awaken by the engine as soon as possible, and continues its execution.

Let M be the memory allocated for one S3 service. The total time needed with S3 is:

$$T_3 = ST(cs, M, \infty) \tag{19}$$

The uIP approach uses a single buffer which size if equals to the MSS. To allow a fair comparison between uIP and S3, we set $M = MSS$. The speed ratio between uIP and S3 can be approximated by:

$$\frac{T_d}{T_3} \approx \frac{80 + MSS + BW \times (RTT + 0.2)}{120 + MSS + BW \times RTT} \tag{20}$$

As for the S1 strategy, the performance ratio between S3 and uIP is greater than 1 as soon as the bandwidth is greater than 200 bytes/s. When serving a content of 8 kB with our reference network settings, this speed ratio is of 4.1.

Thanks to trivial calculations, the ratio between C_d and C_3 can be approximated by the ratio between T_d and T_3. S3 allows a lower memory charge than uIP.

A short analysis shows that T_3 is not proportional to M: it grows slower than this last one. By using a greater value for M, the service is faster, but the memory charge degrades. With a physical bandwidth of 10 kB/s and a RTT of 50 ms, this strategy is as fast as uIP when its buffer is of only 300 bytes (compared to the 1460 bytes used by the uIP), making its memory charge sensibly smaller.

S4 strategy: "stateless generators".

This strategy is used for large volatile contents generators (A2/B4). In such situation, the generator function takes as parameters the current relative sequence number (index of the first byte to generate) and a maximal amount of data to generate. It is able to generate any part of the response at any time.

This approach fits well with volatile functions because they can be called and recalled several times with any parameters. Thereby, the connection can have many in-flight packets during the service, whatever the amount of memory available.

This strategy is as fast as S0 (it has no other limitation on in-flight segments than the output window), so it is sensibly faster than uIP, but, unlike it, its memory charge is null.

Note that when using either S2 or S4 strategy, resent segments may be different from original ones. This behavior does not fits exactly original TCP semantics, but it does not spawn any undesirable effect on TCP communications in practice.

4.3.1 Idempotent generators

Idempotent contents (B5) let the TCP stack choose the most efficient behavior in case of retransmission. Data can be re-generated when needed (S4), or stored in a buffer while unacknowledged (S3).

In most cases, S4 is more efficient, both in terms of memory charge and of service time. However, in some particular cases, where the generator makes heavy processing (*e.g.*, cryptographic encryption), applying S3 will be more interesting in terms of time, memory charge, and, as a consequence, of energy. As an example, the choice between these two strategies could be based on the measured computation time of the generator routine.

4.3.2 HTTP content length

When serving static (B1 or B2) or small contents (A1), the engine knows the HTTP *contents-length* field of the response when it sends the first segment. In S3 and S4, it must use another way to identify the end of the response: it can use HTTP/1.1 chunks encoding [10] instead of closing the connection to notify the end of the HTTP response.

4.4 Supporting the service strategies

We propose a software architecture able to handle the strategies set described in Section 4.3. As shown in Figure 2, it is made of two parts: a kernel engine and a set of Web applications, which are preprocessed and compiled together before being embedded in a device.

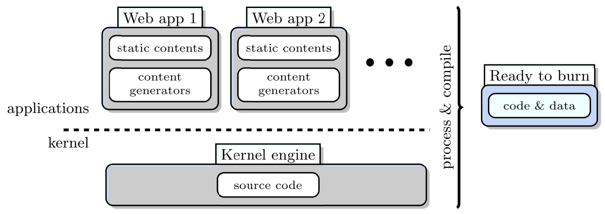

Figure 2: The Web server software architecture

The engine is fully event-driven and uses a cross-layer design in which HTTP and TCP/IP stack are tightly coupled. Applications are made of static and dynamic content. Dynamic content generators are written using an dedicated API allowing the engine to be aware of applicative content properties, and to apply our service strategies.

4.4.1 Kernel engine description

The granularity of the event-driven engine is IP packets. In the kernel, IP, TCP and HTTP are processed at the same level. Several simultaneous connections can be handled, but never more than one packet is processed at a time. Coupled with our strategies, this engine makes only use of a few (and reasonably sized) shared buffers. The input buffer can be of any size (possibly smaller than the packet sizes) if incoming data are processed rapidly enough.

The way we design TCP connections is very different from the well-known Berkeley sockets. The data structure used to store connections states does not include any buffer, making it really small (around 40 bytes). This makes possible to handle the three key points we identified in Section 3.3:

Support of multiple in-flight packets.

Our strategies allow to have no limitation on in-flight packets for most of the contents it serves (static pages, volatile and idempotent generators). In other situations, the number of in-flight segments is limited by the available memory.

HTTP persistent-connections handling.

Because our connection data structure is quite small, it is possible to support multiple simultaneous connections with low memory costs. This enables HTTP persistent connections support.

Large TCP MSS handling.

In input as output, our engine and strategies allow to handle large-sized packets even with small-sized buffers. This makes easy large MSS handling.

4.4.2 An API for contents generators

As described in Section 4.3, most of our strategies make use of an routine called *outc*, without caring for the current TCP stack state. Depending on the persistence of the generator, calls to *outc* involve a different behavior in the kernel engine.

Persistent generators implementation.

Both S1 and S3 strategies require a blocking *outc* abstraction, because they are used for persistent functions, which have to be executed only once. We propose to use protothreads [8], because it is a lightweight blocking abstraction for event-driven architectures. We call PT_OUTC the protothreaded *outc* routine.

```
Data: $i$ a static integer
input: $samples$ a table to send, $nb\_samples$ its size
$PT\_BEGIN$
$preprocess\_samples(samples, 0, nb\_samples)$
for $i$=0 to $nb\_samples$ do
    $PT\_OUTC(samples[i])$
end
$flush\_samples(samples, 0, nb\_samples)$
$PT\_END$
```

Algorithm 1: Stateful generator implemented with protothreads

Algorithm 1 is an example of persistent generator function. It is written for a sensor. It pre-processes a set of sampled values, and sends them as an HTTP response. Because it has side effects, it will be handled by either S1 or S3 strategy, depending on the amount of samples. PT_OUTC calls blocks until data are effectively sent and acknowledged.

Volatile generators implementation.

As S1 and S3, S2 and S4 strategies use a *outc* routine. Unlike persistent generators, volatile ones are executed in a given context (a sequence index and the size of the output to generate) that is provided by the TCP stack. Such a function can be written by an expert developer with no blocking abstraction.

```
Data: $start, end, i$ local integers
input: $samples$ a table to send, $nb\_samples$ its size
input: $sequence\_index$ the index of the first byte to
       send
input: $buffer\_length$ the current available buffer size
$start \leftarrow sequence\_index$
$end \leftarrow min(start + buffer\_length, nb\_samples)$
for $i$=$start$ to $end$ do
    $outc(samples[i])$
end
```

Algorithm 2: Volatile generator implemented by an "expert" developer

The generator function presented in Algorithm 2 is an example of volatile generator. It sends a set of samples, without any side effect. This function is stateless and can be called several times with any parameters (to serve or retransmit any subset of the samples). Depending on the amount of samples, it will be handled by either S2 or S4 strategy.

We propose an alternative to this "best effort" approach, using an extension of protothreads. As mentioned by their authors [8], a *state object* can be used with protothreads for local variables storage. By holding multiple *continuations* and state objects per generator, the engine is able to re-run any part of generators code at any time. In fact, one couple continuation/state object is kept for each in-flight segments, allowing segments reconstruction when needed, while keeping a drastically low memory usage. In a more general context, this approach of multiple couples continuation/state objects per protothread is an interesting way to implement rerun points.

```
Data: $start, end, i$ local integers
input: $samples$ a table to send, $nb\_samples$ its size
input: $state$ the current state
$PT\_BEGIN$
for $state.i$=0 to $nb\_samples$ do
    $PT\_OUTC\_CTX(samples[state.i], state)$
end
$PT\_END$
```

Algorithm 3: Volatile generator implemented with our extended protothreads

Algorithm 3 has the same behavior as Algorithm 2, but is written thanks to our extended protothreads. It has no explicit indexes calculation, so it is simpler to write. Here, the state object only contains the iterator.

5. EXPERIMENTS ON A REAL USE-CASE

In order to put to the test our proposals, we implemented a prototype, called *Smews*. It is built in adequacy with the software architecture described in Section 4.4.

Thanks to our prototype, we compare our proposals to the two reference embedded Web servers (uIP's one and Miniweb) when serving the same application.

To make our experiments reproducible, we made Smews source code available[7]. The source code is provided with the application example used in this paper (see Section 5.1). We choose to illustrate a complete example here, rather than micro-benchmarks.

5.1 Application description

We take the example described in Section 1. The Web server is embedded in a smart card and runs a Web application able to manage a personal contacts book.

The embedded 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;

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

cb_extract idempotent generator used to retrieve the list of all contacts;

cb_get idempotent generator that returns information about a given contact (specified by URLs arguments);

cb_add persistent generator that adds a new contacts and returns a status code.

We implemented a simplified version of the application for uIP, because it does not handle the advanced typing of Web contents. Regarding Miniweb, it is only able to serve the static files of the application.

This application can be entirely supported by Smews, because it handles static and dynamic contents, as well as URL arguments. The nature of dynamic contents (persistent, volatile or idempotent) is provided by the Web application designer, via XML annotations in the *c* source code of dynamic content generators. It is took into account by the Smews engine at runtime.

It is important to note that in the current implementation of Smews, the choice of the strategy for serving idempotent content is not done dynamically at runtime. In our experiments, both *cb_extract* and *cb_get* are handled as volatile content (in-flight segments are not kept in memory).

In practice, it would be necessary to perform dynamic memory allocation for an efficient support of the S3 strategy. In our experiments, the buffer for serving persistent content has a fixed size (32 bytes for our experiments).

In other words, Smews is able to serve dynamic persistent content of any size (multiple small segments are sent), but in case of multiple concurrent requests, the output will slow down because a single buffer is shared by all the connections.

In most situations, this is not an issue because large generated content are rare (see Section 3.1.3). The most often, they simply contain some server-side informations (like *cb_extract*), and are rarely persistent, so they do not monopolize the buffer for long durations.

5.2 Experiments configuration

For our experiments, we use a workstation using Windows XP as operating system, and Internet Explorer 6 as Web browser, the most common configuration used by clients for World Wide Web accesses[8]. It is important to remember that Windows TCP/IP stack implements the TCP delayed ACKs strategy.

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

We presented in [9] an analysis of the memory footprint of the three servers on the Funcard. uIP and its Web server require around 3 kB of RAM and 12 kB of EEPROM. Smews only needs 200 bytes of RAM and 8 kB of EEPROM. Miniweb only used 100 bytes of RAM and 4 kB of EEPROM (its functionalities are quite simple: it is unable to serve dynamic content, to handle multiple connections or a variable MSS, and does not parse HTTP requests).

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

5.3 Experimental results

We compare the performance and memory charge of the three embedded Web servers on the contacts book application. The procedure of our benchmark is in two steps. The first step is the initial connection of the Web browser to the smart card. The four static files are served during this phase, and the list of contacts is retrieved, thanks to a request to *cb_extract*. During the second phase, we put to the test small contents generators: contacts are added *cb_add*, and contacts informations are retrieved via *cb_get*.

5.3.1 Performance

Table 2 shows the time spent by the three Web servers for each request on the Web application. The measurements are made using *tcpdump*, and compared to theoretical values computed by our model. Both measurements and theoretical values take into account the request and response transmission, including HTTP headers. As an example, uIP's Web server sends each HTTP header in two TCP segments before the content. This specific behavior is also taken into account in our calculations.

uIP's Web server and Miniweb do not handle HTTP persistent connections. To allow fair comparisons with Smews, connection establishment and closure times are only included in the global service time of the whole page. That is why for uIP and Miniweb, the time measured to send the whole Web page is sensibly greater than the sum of individual requests times.

First, our measurements highlight the performance gap between uIP and Miniweb/Smews (8.2 s for the whole page, against 2.6 or 1.8 s using Smews). This is mainly due to the limitation of uIP to a single in-flight packet. Miniweb is faster (10 %) than Smews only for the two smallest static files. This is because (i) its HTTP response contains a simplified header and (ii) it starts sending its response before having received the entire request[9]. In other cases, Smews is a faster than Miniweb (40 % for the whole page), and supports entirely the Web application (Miniweb is only able to serve the static files).

Secondly, these measurements show that our model is precise (maximal error is 11 %). On both Miniweb and Smews, the value computed by the model is always under-estimated. This is simply because it does not take into account unpredictable execution overheads in both server and client sides. Furthermore, even in our local network configuration, the TCP stack of the client has other applications and traffics to manage than those of the experiments.

Concerning uIP, the measurements are lower than the model estimations. This is because of the the particular TCP delayed ACKs implementation in Windows: in fact, the 200 ms delay is not triggered when a packet is fully received (as in our model), but when its first byte has been received.

5.3.2 Memory charge

Thanks to the traffic dumps we made and to the access to the servers source codes, we have been able to estimate the memory charge involved by each request on dynamic contents (for the three servers, static contents involve a null memory charge). Table 3 synthesizes these measurements, and compares them to our model.

[9]Miniweb does not decode HTTP requests. It simply affects

Content	uIP			Miniweb			Smews			Speed factor	
	model	measure	diff.	model	measure	diff.	model	measure	diff.	$\frac{uIP}{Smews}$	$\frac{Miniweb}{Smews}$
index.html 752 B	0.77	0.70	10 %	0.12	0.14	-11 %	0.14	0.16	-11 %	× 4.5	× 0.9
style.css 826 B	0.78	0.70	11 %	0.14	0.14	-4 %	0.15	0.16	-9 %	× 4.4	× 0.9
script.js 3613 B	1.48	1.36	9 %	0.47	0.50	-5 %	0.43	0.44	-1 %	× 3.1	× 1.1
logo.png 5805 B	1.91	1.76	9 %	0.73	0.78	-6 %	0.66	0.66	0 %	× 2.7	× 1.2
cb_extract 1915 B	1.10	1.01	4 %	–	–	–	0.26	0.29	-11 %	× 3.5	–
cb_get 28 B	0.70	0.68	2 %	–	–	–	0.06	0.06	0 %	× 10.6	–
cb_add 2 B	0.70	0.67	4 %	–	–	–	0.06	0.06	0 %	× 10.8	–
whole page	–	8.2	–	–	2.6	–	–	1.8	–	× 4.6	× 1.4

Table 2: Measured performances compared to the model (in seconds) The time needed to serve each content of the application has been measured for each Web server. The error done by the model is given as a percentage of the measurement. The speed ratio between Smews and the two reference Web servers are given.

Content	uIP		Smews	
	model	measure	model	measure
cb_extract 1915 B	666	603	0	0
cb_get 28 B	29	30	0	0
cb_add 2 B	25	24	1.3	1.5

Table 3: Measured memory charges compared to the model (in *bytes.seconds*)

This shows that our model is precise not only for performance evaluation, but also for memory charge. Here, the maximal error is of 11 %. We also notice that Smews allows drastically low memory charges: idempotent contents (*cb_extract* and *cb_get*) involve a null charge (remember that in is this example, they are served as volatile contents), while *cb_add* sends its response (of 2 bytes) with a charge of 1.5 B.s, against 24 B.s for uIP. This shows that the strategies we proposed in Section 4.3 are efficient in terms of memory as well as of performance. When serving large generated contents, uIP has a big memory charge.

These experiments prove the efficiency of the architecture and strategies we proposed. They also show that the fine-grained model proposed in Section 3 is very precise, with a maximal observed error of 11 %. Our prototype, Smews, makes a gap with the state of the art embedded TCP/IP stack and Web server both in terms of speed and memory charge. Furthermore, Smews provides a support for fully-fledged embedded Web application: it handles simultaneous HTTP persistent connections, uses a fine description of its contents generators (via our taxonomy), decodes URLs arguments, and more.

6. CONCLUSIONS

Most embedded devices are closed systems only able to communicate with dedicated terminals. These *ad hoc* solutions allow to reach the best compromises between cost and performance, but do not allow a generalized interaction between users, various devices and networks as the Internet. By working on embedded Web servers solutions, we showed that it was possible to use an already widespread applicative model in embedded devices, while keeping low hardware requirements and cost.

We established and validated a traffic model for small systems serving data over TCP. We proposed to measure the cost of each request service by integrating the server memory consumption over the sending time. The service strategies we propose for this model allow to serve any kind of Web content efficiently and with a low memory charge. By caring simultaneously for the transport layer and applicative contents, these strategies make possible the usage of a high level applicative model in the most constrained devices. Our prototype, Smews, confirms the efficiency of our strategies in this context in comparison with state of the art solutions.

As a future work, we will focus on the usage of Web servers on sensor networks. In such context, multi-hop communications and high loss-rates invalidate a part of the initial hypothesis for our model, and introduce new open issues. We will also study the impact of secure connections support on our service strategies.

7. REFERENCES

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

[2] E. Altman, K. Avrachenkov, and C. Barakat. A stochastic model of tcp/ip with stationary random losses. *IEEE/ACM Trans. Netw.*, 13(2):356–369, 2005.

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

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

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

[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] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. The web of things: interconnecting devices with high usability and performance. In *6th International Conference on Embedded Software and Systems (ICESS'09)*, HangZhou, Zhejiang, China, May 2009.

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

a TCP listening port for each of its Web pages.

[11] S. Floyd. Rfc 2582: The newreno modification to tcp's fast recovery algorithm, 1999.

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

[13] D. Guinard and V. Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences)*, Madrid, Spain, Apr. 2009.

[14] D. Guinard, V. Trifa, T. Pham, and O. Liechti. Towards physical mashups in the web of things. In *Proceedings of INSS 2009 (IEEE Sixth International Conference on Networked Sensing Systems)*, Pittsburgh, USA, June 2009.

[15] 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.

[16] 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.

[17] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proc. of SOSP '97*, pages 52–65, New York, NY, USA, 1997. ACM Press.

[18] T. V. Lakshman and U. Madhow. The performance of tcp/ip for networks with high bandwidth-delay products and random loss. *IEEE/ACM Trans. Netw.*, 5(3):336–350, 1997.

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

[20] X. Liu, Y. Hui, W. Sun, and H. Liang. Towards service composition based on mashup. volume 0, pages 332–339, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[21] M. Mathis, J. Semke, and J. Mahdavi. The macroscopic behavior of the tcp congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3):67–82, 1997.

[22] OMA. Smartcard-web-server, 2008.

[23] J. Padhye, V. Firoiu, and D. Towsley. A stochastic model of tcp reno congestion avoidence and control. Technical report, Amherst, MA, USA, 1999.

[24] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose. Modeling tcp reno performance: a simple model and its empirical validation. *IEEE/ACM Trans. Netw.*, 8(2):133–145, 2000.

[25] N. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao. Tiny web services: design and implementation of interoperable and evolvable sensor networks. In T. F. Abdelzaher, M. Martonosi, and A. Wolisz, editors, *SenSys*, pages 253–266. ACM, 2008.

[26] 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.

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

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

[29] W. Stevens. Rfc 2001: Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, 1997.

[30] V. Stirbu. Towards a restful plug and play experience in the web of things. In *ICSC '08: Proceedings of the 2008 IEEE International Conference on Semantic Computing*, pages 512–517, Washington, DC, USA, 2008. IEEE Computer Society.