# Consistency and scalability in event notification for embedded Web applications

Simon Duquennoy, Gilles Grimaud, Jean-Jacques Vandewalle

# Consistency and scalability in event notification for embedded Web applications

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

*A new way to interact with small devices consists in embedding tiny Web servers, allowing the devices to serve fully-fledged Web applications. When the device needs to keep its users up-to-date of its internal state, the Web application has to use an event publication solution. Several works have recently been conducted in order to evaluate the trade-offs of various Web-based event notification solutions. In this paper, we propose to evaluate the feasibility of event notification in embedded Web applications. We conduct a large set of experiments in order to compare various push and pull based approaches for embedded systems. We show that a push-based approach can be very efficient in most situations, both in terms of client consistency and of scalability.*

## 1. Introduction

A new way to implement embedded systems software consists in embedding Web servers in devices. Such servers can be used in sensors, home automation or routers, making these devices accessible from any computer, PDA or smart phone. Furthermore, Web application development is well-known and widespread. The global interconnection of devices based on Web technologies is called the *Web of Things* [26, 15, 14, 11].

Web technologies are based on a request/response model. HTTP allows clients to retrieve resources from servers, but disallows servers to push data to the clients. However, there are many use cases where servers would like to notify clients for event, *e.g.*, auction Web site, stock ticker, news portal, forums, chat-rooms, etc. Usually, such behavior is implemented in AJAX applications by polling the server at a given interval. A new model named *Comet* [22] allows servers to push data over HTTP. Works have been conducted in order to evaluate the best strategies for Web-based event-notification [18, 1].

Intuitively pull-based approaches are simple to handle on the server-side, because it is stateless. Push-based approaches are not trivial to design, and require more resources on the server-side, which has to keep and manage information about the clients listening to events. Web push is often considered as a luxury for clients, with huge cost on the server-side [4, 18, 2].

Many Web of Things use cases also require event-notification: sensors, routers, and home automation systems would like to trigger alerts or to notify some changes on their environment or internal state. In this context, where the server runs on a tiny device, the choice between push and pull-based approach is very important. It has a great impact on the scalability of the application, on the energy consumption of the system, on its hardware requirements, and on the reactivity and consistency obtained on the clients-side (some applications are very critical and require excellent reactivity).

We focus on the approach where each device runs a whole HTTP/TCP/IP stack rather than on solutions where the management of HTTP, TCP and IP are delegated to a gateway located between the client and the targeted device. This model is a user-centric architecture, where the embedded Web servers are organized around the client. This approach has been discussed in [10]. We showed in a previous work [11] that it is possible to serve efficiently interactive AJAX [13] applications over TCP/IP from devices with a CPU cadenced at a few MHz, with a few hundred of bytes of RAM and a few kilo-bytes of EEPROM.

In this paper, we discuss and evaluate the costs of the embeddability of server push for embedded Web applications. We compare push and pull based approaches in the context of the Web of Things, in order to highlight the trade-offs of each methodology. We base this analysis on the works of Bozdag *et al.* [1]: they evaluate the pros and cons of push and pull based event notification for usual (non embedded) AJAX applications.

This paper is organized as follows: Section 2 presents a state of the art of embedded Web servers and Web-based event notification. In Section 3, we present the challenges for embedding Comet in tiny Web servers. We conduct experiments in order to find the trade-offs of push and pull based approaches for the Web of Things in Section 4. We finally conclude in Section 5.

## 2. State of the art

In this section, we present a state of the art of embedded Web servers and Web-based event notification.

### 2.1. Embedded Web servers

Several works [7, 17, 21] have shown that it is possible to embed Web servers in tiny devices. Proposed solutions such as iPic [24], WebIt [16] and Miniweb [9] are stand-alone Web servers, with no underlaying operating system, but thought as the operating system itself. They do not rely on usual general-purpose networking interfaces like Berkeley sockets, but implement their own dedicated TCP/IP stack. That allows to design a cross-layer architecture instead of usual layered architectures, making possible many optimizations, thus saving memory, code size and energy. Their memory footprint is around one or two hundreds bytes of RAM and a few kilo-bytes of EEPROM.

Other works have been done on a more generic context, focusing on TCP/IP stack support for embedded systems: TinyTCP [3], mIP [23] and uIP [8]. It is possible to run a Web server on such stacks, but this forbids cross-layer optimizations for efficient Web contents service over TCP. The memory footprint of this kind of TCP/IP stack is of a few kilo-bytes of RAM and dozens of kilo-bytes of EEPROM.

As far as we know, no work has been done about the embeddability of push-based Web applications on tiny devices. In fact, many previous works on embedded Web servers mainly focus on the service of static Web pages, which are pre-processed and statically embedded in the server at compile time.

In [11], we propose new solutions for efficient embedded Web applications support with very low memory requirements. We propose solutions based on off-line computations and cross-layer optimizations, where TCP is specialized for supporting an HTTP server. Our prototype, named *Smews*, is publicly available[1] and has been ported to various targets such as sensors and smart cards. It is able to serve efficiently fully-fledged Web applications made of static and dynamic contents, including push-based notification.

### 2.2. Web based event notification

HTTP was initially designed for retrieving resources on the Internet, so it is based on a simple request/response model. Today, Web servers manage dynamic applications, and sometimes need to keep their clients up-to date, in adequacy with the server state.

The simpler approach for keeping clients up-to date in a Web context consists in polling the server with an empirically chosen time interval. That can be done in dynamic AJAX [13] Web applications. Small intervals improve the client-side coherence while big intervals improve scalability by saving network and server resources. A common solution for this problem is based on an adaptive Time To Refresh (TTR) [25, 4], trying to calculate a polling interval that fits with event publications. This approach is only efficient when the event publication tends to be constant.

Netscape introduced in 1996 a solution for pushing data from Web servers [20], based on HTTP streaming. A long-lived HTTP connection (initiated by the client) was used in order to send a streamed content. When a client notification is needed, a new part of the content is sent by the server, and the connection remains open. The browser has to be still waiting for the end of the HTTP response, which possibly never occurs. It receives notifications by chunks.

The usage of this solution in AJAX Web applications is known under the name of *Comet* [22]. A protocol draft (based on JSON and on a publish/subscribe model) named Bayeux [6] has been proposed for Comet support. The Cometd project [5] provides Bayeux implementations for various Web servers, such as Jetty [19]. Google's DWR [12] also provides a Comet support, without relying on Bayeux.

Bozdag *et al.* propose a study of push and pull approaches for AJAX applications [18, 1]. They provide key metrics for performance analysis of Web-based event notification. They also conducted a large set of experiments in order to compare existing approaches (push vs. pull) and implementations (Cometd vs. DWR). The conclusion of this work is that Comet provides great reactivity, client coherence and low traffic overheads, while polling provides a better scalability in term of server-side CPU usage.

Both push and pull-based solutions have their own advantages. That is why some works [4, 2] propose adaptive solutions where the choice between push or pull is done at runtime, depending on the current server congestion and on the client requirements.

[1] Smews source code available at: `http://smews.gforge.inria.fr/`

## 3. On Comet for embedded Web servers

In this section, we describe the challenges of Comet support on embedded Web servers.

### 3.1. Why supporting Comet?

Embedded devices such as sensors or home automation systems may offer various services, such requiring various interaction models with the client accessing them. We propose a classification of the different interactions schemes that can be needed by such systems:

**On-demand information exchange:** the client needs to send/receive data to/from the device (*e.g.*, getting/sending applicative information, driving or managing the device).

**Event triggering:** the server needs to notify the client that something happened (*e.g.*, the environment has changed, a precessing has ended).

**Data sampling:** the client needs to collect continuously data from the device (*e.g.*, monitoring of the environment).

Since the Web of Things aims at designing embedded software using the Web applicative model, it has to provide solutions for handling each of these interaction schemes. The original REST model of HTTP (based on a request-response scheme) is well suited for on-demand information exchange (using GET/POST to retrieve/send data).

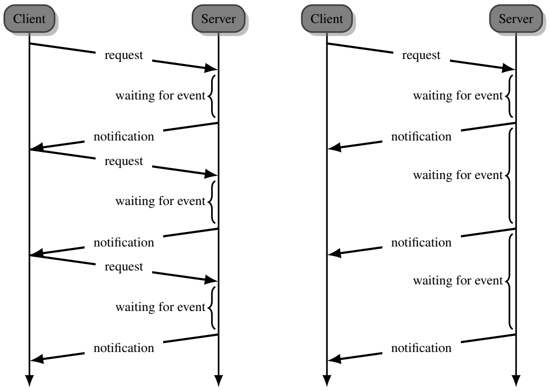
For supporting triggering or sampling, the device needs to push information to the client, which cannot be be done with the REST model in Web applications. That is why the support of Comet (*i.e.*, server push in Web applications) is a key point in the design of the future Web of Things.

### 3.2. Long polling vs. streaming

In a general context, event-notification can be implemented easily by opening a connection from the server to the client, sending a data and closing the connection. In a Web context, a server can not establish a connection to the client, because (i) if the client is in a local network and uses address translation (NAT), it is not publicly visible and (ii) the request/response model of HTTP is always used in Web applications.

Comet implementations mainly propose two ways to notify clients on a Web context: long polling and streaming.

**Long polling** Each time a client needs to register to an event, it sends an HTTP request to the server. The server idles, sends the HTTP response when needed. The client connects again if it needs a new notification;



(a) Long polling approach (b) Streaming approach

**Figure 1. Comparison of long polling and streaming**

**Streaming** When a client needs to listen for an event, it sends an HTTP request to the server. When event occurs, the server sends notifications in a *chunk-encoded* HTTP response, without ending it. The client is still listening. Next notification will be the following of this potentially never-ending HTTP response.

Figure 1 shows the behavior of long polling and streaming approaches when notifying a client several times. With long polling, a single notification is involved per client request, while in streaming, the client receives multiples notifications per request. When the client needs to be notified multiples times, streaming generates a lighter traffic.

The Bayeux protocol provides a support for both long polling and streaming, but Cometd, the most common implementation, only provides a support for long polling. Google's DWR supports both approaches.

### 3.3. Server-side support of Comet

On usual Web servers, Comet is considered as heavy because it requires to manage many clients simultaneously, compared to pull-based approaches where connections are idle as soon as requests have been served.

Usual Web application containers associate server-side code to URLs (in Java for Servlets containers), and call this code for each incoming HTTP request.

Most Web servers (*e.g.*, Apache, IIS, ...) allocate one thread/process per connected client, because they need one socket per connection, and because sockets are the most often managed via blocking routines. URLs handlers (*e.g.*, Servlets) have to generate an output and to return. In Comet, this code has to enter a passive wait for an undetermined duration. As a consequence, for each client, a server based on this approach has an idle thread and TCP connection, thus wasting a lot of resources on the server.

New frameworks such as Cometd and DWR have been designed including a native support for Comet. Instead of using one blocking thread/process per client sockets, they use a common blocking routine that allows to listen on multiple sockets (based on the *select* POSIX system call). They provide special routines to URLs handlers (*e.g.*, Servlets) allowing them to wait for events without idling a thread (as an example, Jetty uses *continuations*). When an event occurs, all URLs handlers listening for it are awaken and their associated clients receive the notification. With such strategy, no more resource is wasted for threads.

### 3.4. Breaking sockets, saving memory

Web servers for tiny embedded systems can work with no underlaying operating system. They use their own dedicated TCP implementation instead of usual Berkeley sockets. The most often, they are event-driven systems (they schedule their task with no thread). At a given time, only one packet is managed, either in input or in output.

In Smews, the structure used to store connections contains information about IP, TCP and HTTP. Smews is event driven and uses of only a few global buffers, shared by all the connections. Each connection only require almost 30 bytes of RAM. This allows to handle easily a large number of clients even in very constrained hardware.

Intuitively, event-driven architectures fit well with event notification. Instead of managing threads waiting for events, a simple event pool is used. This point makes possible Comet support in tiny Web servers. In Smews, when an event is generated in order to notify a set of clients, the HTTP response is built only once. It is placed in a buffer that will be used to send the notification to every listening clients in separate TCP segments (multi-cast is not supported by standard TCP). In usual OS architectures, the data is generated once per client and replicated on each socket.

## 4. Life size experiments

In this section, we put to the test various Web-based event notification methods on an embedded Web server.

### 4.1. Goals of the experiments

The objective of the experiments is to know the trade-offs of various Comet and polling implementation for embedded Web applications. One of our aims is to know if the results of Bozdag *et al.* work [1] can be applied to tiny Web servers. We also evaluate the usability of push-based approach in embedded devices. The steps for achieving these goals are:

1. identifying the parameters of the various experiments, in order to benchmark different approaches on various context;
2. identifying relevant metrics in order to evaluate the trade-offs of each solution;
3. creating a Web application and implementing it for all push and pull-based approaches;
4. simulating clients connections in order to run the experiments;
5. analyzing the results of the experiments.

### 4.2. Experiments description

We describe here the environment of the experiments we conducted in order to compare various notification approaches in the context of the Web of Things.

#### 4.2.1 Settings

We start from Bozdag *et al.* work, which describes a protocol for evaluating event-notification solutions. So use the same variables for the settings of our experiments.

**Number of concurrent users** This variable allows to evaluate the scalability of each approach. Even in the context of Web of Things, scalability may be required: several clients may be each listening on several events of a same device. Because of embedded devices constraints and applicative needs, we choose lower interval than Bozdag *et al.*: [1;256], instead of [100;10000].

**Publish interval** This is the frequency of event publication by the server. We used the same values than Bozdag *et al.*: 1, 5, 15, 30 and 50 seconds. We added a sixth configuration based on a random choice between 1 and 50 seconds, making the publication no more regular.

**Pull interval** When a pull approach is used, we used various intervals. We use the same values than Bozdag *et al.*: 1, 5, 15, 30 and 50 seconds. It may be interesting to try a strategy based on adaptive TTR, but since we measure asymptotic performance, it will provide the same result as a static interval equals to the publication interval.

**Application mode** For this variable, Bozdag *et al.* used three modes: polling, Cometd, DWR. Since Cometd and DWR can not be executed in tiny devices with no operating systems, we use our own Comet implementations. Our three modes are: polling, long polling, and streaming. Bozdag *et al.* did not use any streaming implementation.

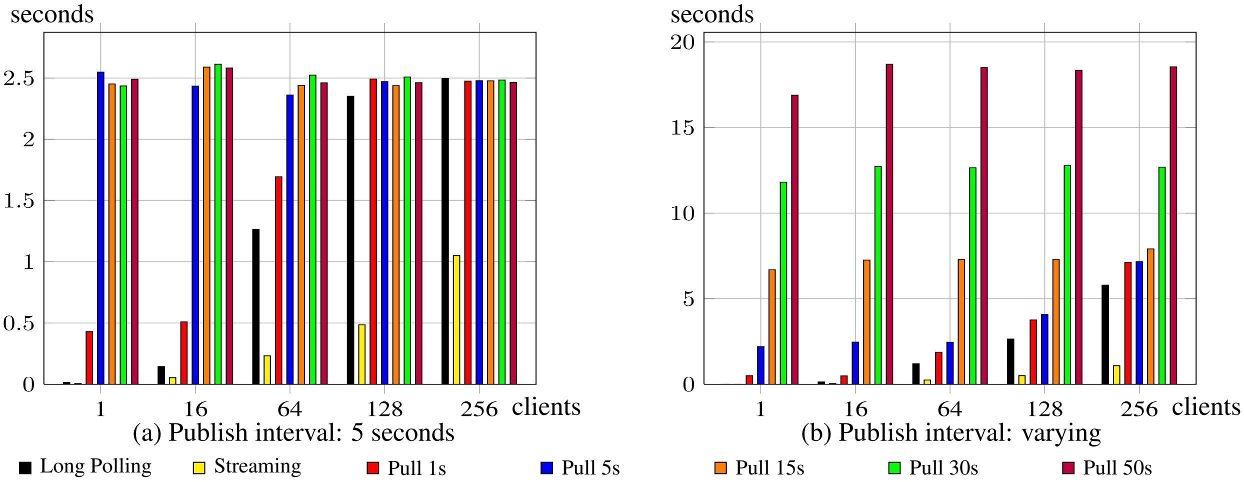


**Figure 2. Mean Publish Trip time for a publish interval of 5s (a) or varying between 1 and 50s (b)**

### 4.2.2 Experiments configuration

We designed Web applications with event notification for the experiments. They are executed by our embedded Web server prototype, Smews (this is the only available embedded Web server that supports Comet). The device we choose for our experiments is a sensor named WSN430, with the following hardware configuration: 16 bits MSP430 CPU at 8 MHz, 10 kB of RAM, 48 kB of EEPROM and a serial line at 14400 B/s as communication interface. With its smallest configuration, Smews only requires 200 bytes of RAM and 8 kB of EEPROM.

The sample application is very simple: the server gets a new value at a given interval. Each time, it notifies every listening clients. The application supports polling, long polling and streaming. During an experiment, a given number of clients connect to the server in order to receive notifications. All the clients use the same notification method.

The four variables described in Section 4.2.1 can be combined in 210 experiment settings. Each experiment has a duration between 5 and 10 minutes, and has been run 10 times. The results presented here are the mean of these 10 iterations, after the removal of the 2 lower and higher sampled values.

We used a script written in *Python* in order simulate the clients, allowing to run the experiments in the various settings described in Section 4.2.1. All the clients are simulated on the same machine and access the sensor using the same serial line.

The experiments we conduce aim at being as general as possible. The usage of a serial line, which is full-duplex and collision-free, allows the experiments to be independent from specific physical layer properties. In future works, we plan to analyze the impact of lossy links such as ZigBee, Wifi, or other shared lines.

## 4.3. Experimental results

In order to evaluate the trade-offs of each approach, we identified a set of relevant metrics and extracted them from our experiments. These metrics are inspired from Bozdag *et al.* works. In the following sections, we present and synthesize our benchmarks results (only a relevant subset of all the experiments conduced are shown).

### 4.3.1 Mean Publish Trip time (MPT)

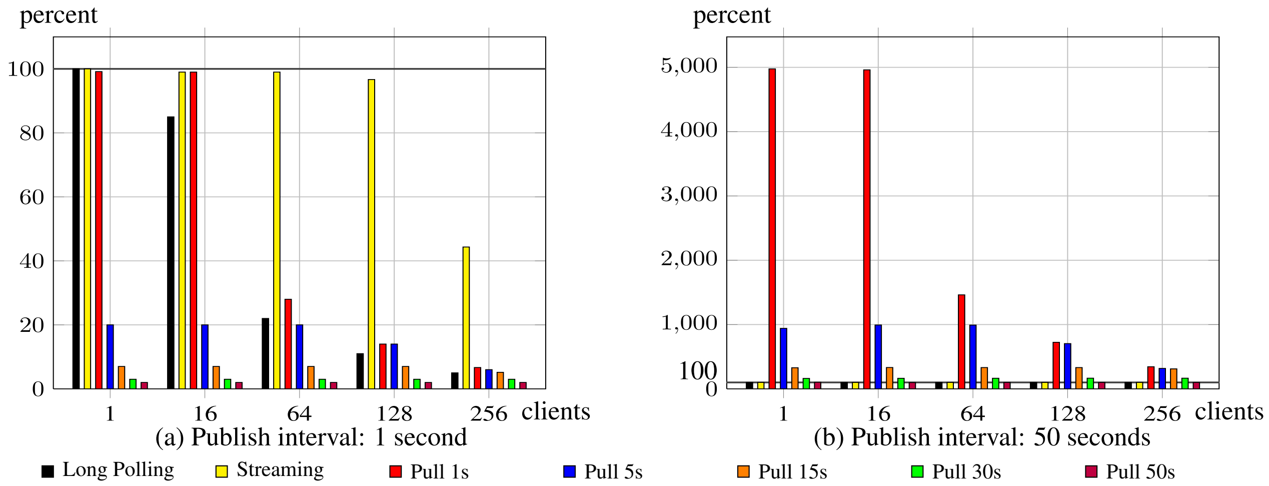
The publish trip-time is the time elapsed between the creation of a data by the server and its reception by the client. It shows how long it takes for the client to be updated when an event occurs. In their study, Bozdag *et al.* showed that push-based approaches provide a lower Trip Time, so a better consistency.

Figure 2 shows the MPT we measured from our experiments, with a publication interval of 5 seconds or varying between 1 and 50 seconds.

With a few clients (less than 64), both streaming and long polling provide excellent trip times in comparison with polling. This is because with Comet, the server pushes data to all the clients as soon as an event occurs.

Long polling generates more traffic than streaming because it forces the client to send a HTTP request (around 600 bytes) between each notification (see Section 3.2). With more than 64 clients, long polling does not provide a significantly shorter trip time than polling, because clients registrations saturate the traffic.

Streaming provides the shortest trip times, because it generates a lightweight traffic. With a growing number of clients, it makes a gap with both long polling and polling approaches. Streaming approach provides low trip time with



(a) Publish interval: 1 second

(b) Publish interval: 50 seconds

**Figure 3. Received Publish Messages for a publish interval of 1s (a) or 50s (b)**

no scalability issue.

With a publish interval of 5 seconds, the traffic is saturated in most configurations. Logically, in such case, the mean trip time is almost half the publication interval. In this context and with 256 clients, the streaming approach allows to reach a mean trip time of only 1 second.

#### 4.3.2 Server Performance (SP)

The server performance shows the CPU usage of the server software. In Bozdag *et al.* experiments, it was used as the indicator of the server charge. The authors concluded that SP grows faster with Comet solutions, possibly causing scalability issues. In our experimental configuration, the performance bottleneck is the network capacity rather than the CPU, so we did not measure SP in our experiments.

#### 4.3.3 Received Publish Messages (RPM)

The RPM is the mean amount of messages received by the clients. When presented as a percent of the published events, it shows the possible traffic overhead.

Figure 3 shows the RPM for a publish interval of 1 second and of 50 seconds. With Comet (long polling as well as streaming), the RPM never exceeds 100% because each published data is received only once by the clients.

With polling, the lower is the polling interval, the higher is the RPM. With 16 or less clients, a polling interval of 1 second and a publication interval of 50 seconds, 5000% of the publications are received; in other words, 98% percent of requests were unnecessary. We also notice that when using polling with a high rate, the RPM decreases with higher number of clients. This occurs when the serial line of the sensor is saturated with the traffic.

#### 4.3.4 Received Unique Publish Messages (RUPM)

The RUPM is the mean amount of unique publications received by clients. It is given as a percent of the published events, and shows if clients miss any items.

Figure 4 shows the RUPM for a publish interval of 1 second or of 30 seconds. It shows that with a too big polling interval, the clients may miss many notifications. With many clients, the network bandwidth is saturated, also involving misses. The long polling approach provides a RUPM that is comparable (in fact, a bit worse) to the polling approach with an interval equals to the publication rate. This is because long polling requires one HTTP requests per notification, as well as polling.

With the streaming method, misses are quite rare. The RUPM is close to 100% even with 128 clients for a publication interval of 1 second. The only situation in which we obtained a RUPM significantly lower than 100% with streaming was with a publication interval of 1 second and more than 128 clients.

Logically, with slow publication rate, clients misses become more rare.

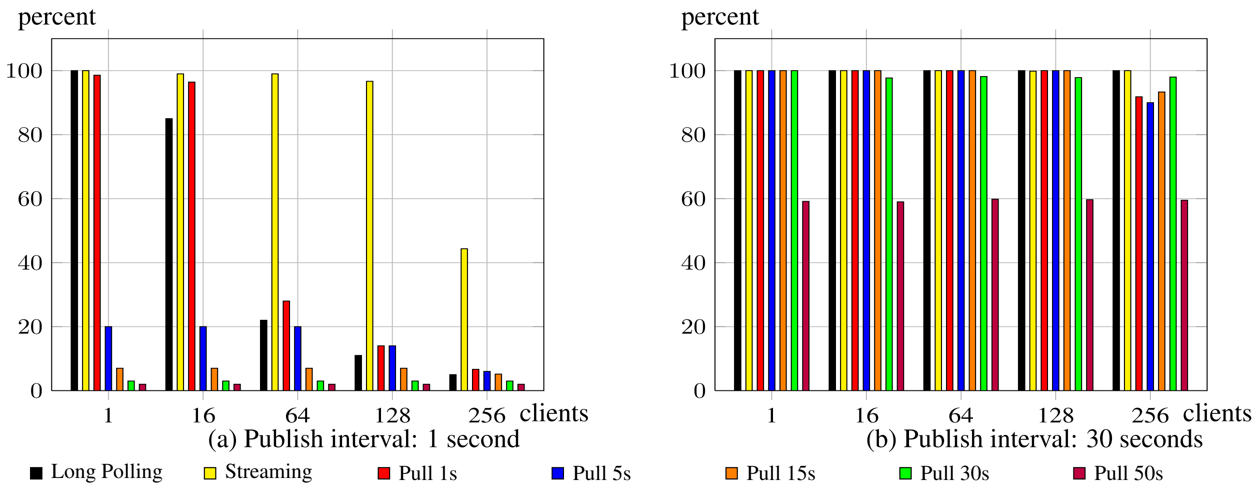
#### 4.3.5 Received Message Percentage (RMP)

This variable was used by Bozdag *et al.* to have an indicator on packet losses. In our configuration, the network bottleneck was the serial line of the sensor, where losses never occur, so we did not measure the RMP.

#### 4.3.6 Network Traffic in Packets (NTP)

The NTP allows to evaluate the network usage of a strategy.

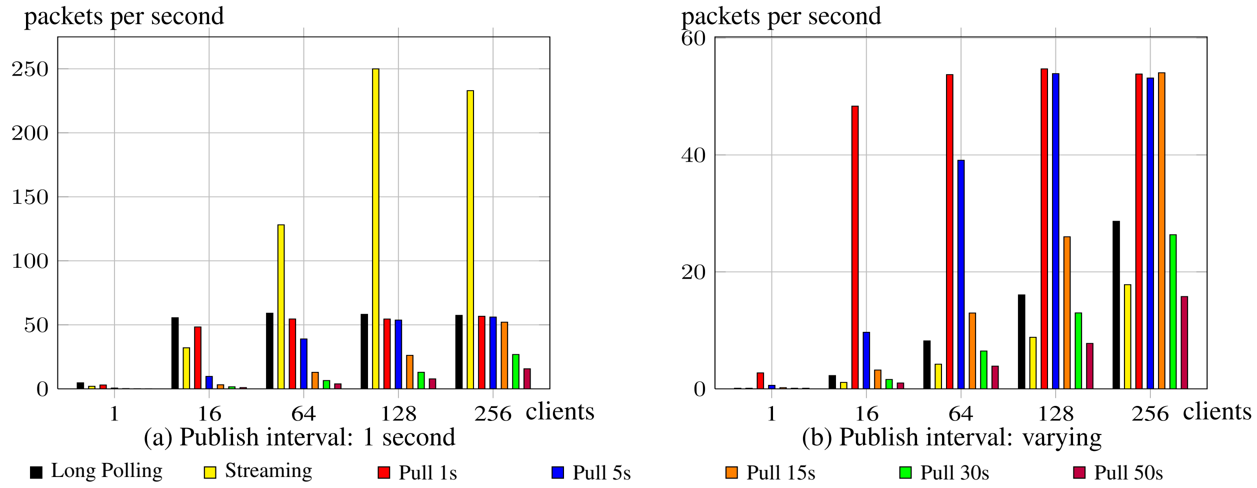
Figure 5 shows the NTP for a publish interval of 1 second or varying between 1 and 50 seconds. For polling and long



(a) Publish interval: 1 second

(b) Publish interval: 30 seconds

**Figure 4. Received Unique Publish Messages for a publish interval of 1s (a) or 30s (b)**



(a) Publish interval: 1 second

(b) Publish interval: varying

**Figure 5. Network Traffic in Packets for a publish interval of 1s (a) or varying between 1 and 50s (b)**

polling approaches, the network comes to saturation from almost 50 packets per second. This highlights the situations where the bottleneck is the traffic (with many clients and/or short polling intervals).

With streaming, NTP grows up to 250 packets/second with a publication every second and 128 or more clients. This is because with streaming, the traffic is made of a lot of small TCP segments, allowing to send more packets with the same bandwidth limitations. In fact, this point helps the streaming approach in providing good performances.

#### 4.3.7 Network Traffic in bytes (NTB)

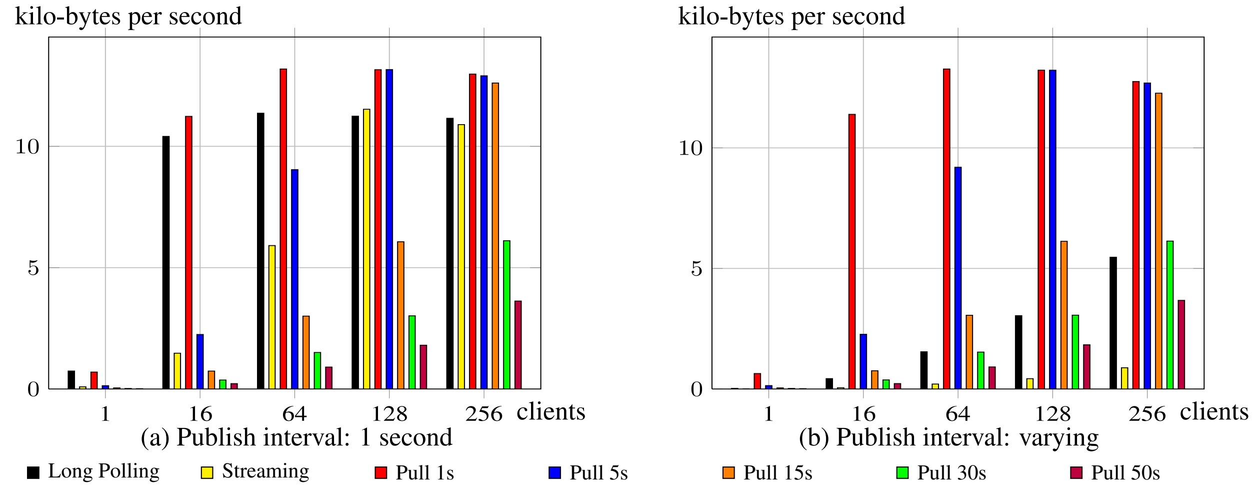
This metric was not used by Bozdag *et al.*, but our analysis of the NTP shows that it is not the best way to synthesize network congestions. That is why we also evaluate the network traffic in bytes per seconds.

Figure 5 shows the NTP for a publish interval of 1 second or varying between 1 and 50 seconds. It provides a precise overview of the network usage. It shows that long polling generates a heavy traffic; this is because it involves requests between each publication. With a large polling interval, polling allows a very low network usage (but provides less data coherence).

Streaming provides a lower traffic usage than long polling or polling with a short interval.

#### 4.3.8 Mean Coherent Time (MCT)

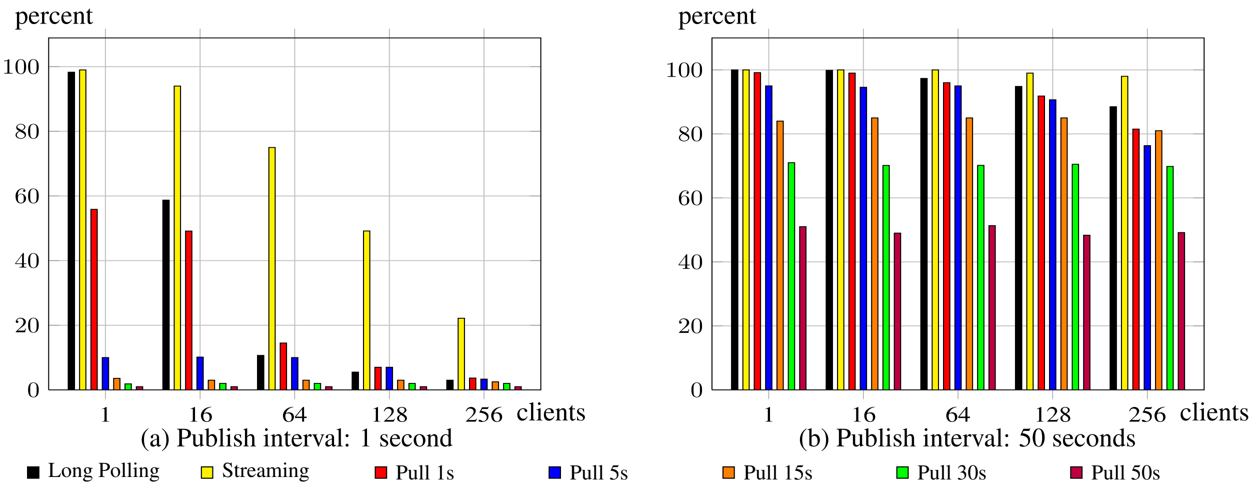
Finally, we introduce a new metric in order to evaluate the client-side data coherence. Bozdag *et al.* made use of the MPT to evaluate it, but this metric is perfectible because it



(a) Publish interval: 1 second

(b) Publish interval: varying

**Figure 6. Network Traffic in bytes for a publish interval of 1s (a) or varying between 1 and 50s (b)**



(a) Publish interval: 1 second

(b) Publish interval: 50 seconds

**Figure 7. Mean Coherent Time for a publish interval of 1s (a) and 50s (b)**

does not take into account client misses. A low trip time can be obtained even with many misses. MCT represents the amount of time spent by the clients with up-to-date data. It provides a synthesis of all the metrics.

Figure 7(a) and 7(b) show the MCT for a publication interval of respectively 1 and 50 seconds.

With a publication interval of 1 second, streaming makes a gap with long polling and polling with any frequency. With 128 clients, it reaches 50%, compared to results between 1% and 7% for polling and long polling.

With a publication interval of 50 seconds, MCT values are higher than with an interval of 1 second. Note that when polling exactly at the publication interval and when no congestion occur, the mean coherent time is of 50%. This is due to the lack of synchronization between the server and the clients. Some clients send their requests and retrieve a data that has just been published by the server, while other clients get their notifications a few seconds before their expiration.

## 4.4. Results analysis

We propose a summary of the results we presented in Section 4.3. We highlight the trade-offs of various event notification approaches in various contexts.

### 4.4.1 Impact of the network

The network capacity plays an important role in the quality of Web-based notification. In our experiments, the server is accessed via a serial line at 14400 B/s. Such a small throughput is common in embedded devices network links, which networking hardware often has no DMA and is unable to manage packets queues. With such simple hardware, which is very different from computer network interfaces, the network capacity is limited by the CPU speed.

The most efficient strategy in term of traffic usage is polling with big interval. In fact, with polling, the traffic generated by each client is very predictable and easy to control. The polling interval can be chosen depending on the number of clients and the network capacity. This is the main advantage of polling approaches.

The traffic generated by Comet approaches is not depending on a client interval but on the server publication interval. Long polling is sensibly heavier than streaming in practice, because it produces a lot of HTTP requests, which are of typically around 600 bytes, while notifications often contain only a few bytes of payload. Streaming produces mainly very small packets, involving low bandwidth usage.

### 4.4.2 Data coherence

In their paper [1], Bozdag *et al.* showed that Comet provides the best coherence. They only used solutions based on long polling, because streaming is still rarely implemented. In the context of embedded Web servers, our results show that long polling does not provide a great data coherence. The MCT obtained with long polling or with polling at the right interval are similar. Long polling allows to reach low trip time, but it involves traffic overhead when compared to polling, because of the cost of client registrations.

By implementing a support for HTTP streaming in our prototype, we were able to compare streaming to long polling and to polling. Our results show that streaming makes a gap with other solutions for all of our 210 benchmark settings. This is because it generates a small amount of requests, uses small packets and is very easy to manage on the server-side since it makes incoming requests very rare. Once the clients are registered, they are served very efficiently. This approach is also the most scalable.

### 4.4.3 Summary

We showed that Web servers embedded in very constrained devices are able to run Comet Web applications. Furthermore, thanks to very small dedicated connections, they are able to handle a large number clients when compared to the amount of volatile memory available.

Bozdag *et al.* obtained excellent performances with long polling in the context of powerful Web servers in [1]. We show that these results can not be applied directly on constrained Web servers, because tiny devices suffer of their slow network interfaces.

By proposing a streaming support in our prototype, we showed that Comet is a great solution for event notification in embedded Web applications. In fact, the streaming approach provides the best results in term of publish trip time, received events, non-redundancy of messages, data coherence and scalability.

In practice, long polling is never better than streaming, but is still interesting is some cases where a client needs to be notified only once of an event.

Polling may be interesting for event-notification only when a great number of clients are needed without requiring a high data coherence: by using a very large interval, the traffic required per client can decrease to very low values.

We identified in Section 3.1 three interactions schemes for Web applications. We summarize here how each of them should be handled in a Web of Things context:

**On-demand information exchange:** this is the basic scheme for Web interactions, for which Comet is not needed. Usual request-response interactions (like with polling) are well suited for it.

**Event triggering:** event triggering should be implemented with long polling, because this allows a client to listen only once for an event.

**Data sampling:** data sampling should be implemented with streaming, since this is the most efficient approach for continuous samples retrieval by a client, as shown by our experiments.

## 5. Conclusions and perspectives

The support of Comet in embedded devices is a key point in the design of the future Web of Things.

Server push solutions are known as reactive but not scalable. Bozdag *et al.* showed [1] that push-based approaches in Web applications are efficient in terms of reactivity, consistency and traffic. The only benefit of polling is the server-side resources usages, allowing a better scalability.

Based on this work, we conducted similar experiments in the context of the Web of Things. We first showed that by using an fully integrated Web server (with its own communication stack, with no OS), Comet can be supported in very constrained devices (256 simultaneous clients supported in only 10 kB of RAM).

The results of our benchmarks showed that Comet provides heterogeneous performances depending on the way it is implemented. Streaming makes a gap in term of performance with both polling and long polling approaches. As a big difference with Bozdag *et al.* results, streaming provides the best scalability.

The event-driven model used by embedded Web servers, coupled with their dedicated TCP/IP stack, allows to support efficiently event-notification. In such model, the whole system is event-driven, making push approaches easier to support. In fact, the well-known scalability issues of push-based approaches seem to be due to traditional OS constraints, which break the native event-driven model of the hardware.

In future works, we plan extend the scope of this study to contexts with link losses and collisions. We also would like to analyze the energy consumption of the devices when notifying events in order to study duty cycle management.

## References

[1] E. Bozdag, A. Mesbah, and A. van Deursen. Performance testing of data delivery techniques for ajax applications. *Journal of Web Engineering (JWE)*, 2009. To appear.

[2] E. Bozdag and A. van Deursen. An adaptive push/pull algorithm for ajax applications. In *Third International Workshop on Adaptation and Evolution in Web Systems Engineering (AEWSE'08)*, pages 95–100, July 2008.

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

[4] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: disseminating dynamic web data. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 265–274, New York, NY, USA, 2001. ACM.

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

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

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

[8] 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.

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

[10] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. Smews: Smart and mobile embedded web server. In *3rd International Workshop on Intelligent, Mobile and Internet Services in Ubiquitous Computing (IMIS'09)*, Fukuoka, Japan, March 2009.

[11] 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.

[12] DWR. Direct web remoting, 2007. `http://directwebremoting.org/dwr/reverse-ajax`.

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

[14] 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.

[15] 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.

[16] 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.

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

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

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

[20] Netscape. An exploration of dynamic documents, 1996.

[21] 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.

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

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

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

[25] R. Srinivasan, C. Liang, and K. Ramamritham. Maintaining temporal coherency of virtual data warehouses. *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 60–70, Dec 1998.

[26] 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.