I. J. Computer Network and Information Security, **2015, 5, 1-8**
Published Online April 2015 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijcnis.2015.05.01

Embedded Real-Time HTTP Server

Radosław Czarnecki
Division of Computer Science, Cracow University of Technology, Poland
Email: czarneck@pk.edu.pl

Stanislaw Deniziak
Department of Computer Science, Kielce University of Technology, Poland
Email: s.deniziak@computer.org

Abstract—This paper presents the architecture of embedded real-time web server. Unlike existing web servers, in our approach, requests are processed not in the "first in first out" order but according to their deadlines and the expected server load. For this purpose the Least Laxity First scheduling method is used. First, requests with imposed hard real-time constraints are served. Then requests enclosed by soft deadlines are processed. Finally, request without time requirements are served in the order they arrived. We also present real-time extensions to the Hypertext Transfer Protocol. We propose headers that enable defining hard and soft deadlines, as well as responses containing time information, that are being sent to the client application. The experimental results showed that in case of real-time applications our server misses significantly fewer requests, due to time out, then existing solutions. The presented server may be very useful for implementing real-time services supported by embedded systems, e.g. in future real-time "Internet of things" applications.

Index Terms—Web server, embedded system, real-time system, HTTP, Internet of things, Sensing as a service.

I. Introduction

More and more embedded systems are connected to Internet. The emerging concepts, like Internet of Things (IoT) [1], Machine to Machine communication (M2M) [2], Sensing as a Sevice (S^{2}aaS) [3], wireless sensor networks [4] etc., stimulate the development of web-enabled devices. Such devices can communicate with web-applications using HTTP protocol [5], a lot of them are controlled by web browsers. For this purpose an embedded web server (EWS) should be built in. EWS usually is a lightweight application implementing only main methods defined by the HTTP protocol. It contains simple web pages, with forms used for the interaction. EWS are used for controlling network printers, wireless routers, network cameras and other devices or sensors. It is expected that in a few years almost each product may be identified and traced on Internet using wireless communication methods. Moreover, a lot of them will supply sensing data to web applications.

Existing web-based sensing applications do not consider time requirements. Although the increasing number of sensing services will cause that sensors with built in EWS will face real-time conditions, which should be satisfied to provide appropriate and required level of Quality of Service (QoS). Recently it was revealed that future Internet of Things or Service Oriented Architectures should address the real-time aspects. Some web applications have time-critical demand, especially in domains like environmental monitoring, transportation. The IoT will comprise billions of intelligent communicating "things" or Internet Connected Objects (ICOs) that will have sensing, actuating and data processing capabilities. Each ICO will have one or more embedded sensors that will capture potentially enormous amounts of data. To enable processing a large number of requests, such ICO should take into consideration real-time constraints.

Real-time applications expect responses from sensors or external services in predictable time periods. Unfortunately HTTP does not support real-time constraints. When the response to the request did not be received during the expected time it is not possible to determine if it was caused by the EWS overload, network faults, server failure or heavy network traffic. Moreover, in existing HTTP servers it is not possible either to control the order of processing of incoming requests or to predict the time of processing. In real-time environment EWS should first, process requests according to their deadlines, second, if it is not able to process request in the expected time, EWS should send timeout message to a client application. Server that meets above requirements will significantly improve QoS in real-time IoT systems, especially when an ICO supports sensing data for a large number of real-time web applications and when getting these data requires time-consuming computations.

In this work we present the architecture of an embedded real-time HTTP server. To enable imposing the real-time requirements for HTTP requests, we will define the real-time extensions of the HTTP protocol. The server schedules all requests according to their priority, which is based on real-time requirements, and an expected processing time. The main appliance of our server would be embedded systems supporting sensing data in real time. According to our best knowledge there are not similar solutions for embedded systems.

The rest of this paper is organized as follows. In the next section related works are presented. In section 3 we present real-time extensions to the HTTP protocol.

Section 4 describes the architecture of the real-time HTTP server. The experimental results showing the advantages of applying our server in real-time web applications are given in section 5. Finally, in section 6, we end with some conclusions and show further work to develop in this field.

II. Related work

In [6] the problem of real-time requirements in the Web of Things (WoT) applications was discussed. Authors observed that a lot of WoT systems interact with embedded devices and expect real-time data, thus development of WoT application that satisfy real-time requirements is one of the main challenges. Although some technologies for real-time communication (e.g. RTP/RTSP [7], XMPP [8]) or real-time interaction (e.g. Comet [9]) were developed, but still more developments and standards are required for real-time WoT and IoT systems.

Real-time web applications are often considered in the context of cloud computing. Current work concerning real-time cloud computing (RTCC) mainly concentrates on 2 domains: adopting existing web technologies to this new paradigm and developing software architectures for real-time applications. Recent studies have been performed on the allocation of resources for real-time tasks. Aymerich *et al.* [10] developed an infrastructure for a real-time financial system based on cloud computing technologies. Liu *et al.* [11] showed how to schedule real-time tasks with different utility functions. The real-time tasks are scheduled non-preemptively with the objective to maximize the total utility by using the time utility function (TUF). Tsai *et al.* [12] discuss a real-time database partitioning on cloud infrastructures. Kim *et al.* [13] investigate power-aware provisioning of resources for real-time cloud services. In their work the real-time constraint is specified in a Service Level Agreement (SLA) between customers and cloud providers. SLAs specify the negotiated agreements, including QoS. In such cloud models the service provider is responsible for the allocation resources. Their work examines power management while allocation of resources should meet the SLA. None of the above studies consider a cost-efficient selection, from a set of different types of resources available in clouds, for real-time tasks. Kumar *et al.* [14] developed an algorithm of resource allocation for applications with real-time tasks. They propose an EDF-greedy scheme that considers temporal overlapping to allocate resources efficiently. The methods of cost-aware synthesis of real-time cloud applications for IoT are presented in [15] [16] [17].

There are a lot of implementations of embedded web servers. Appweb [18] is a compact, multi-threaded server that supports in-memory modules for the ESP, Ejscript and PHP frameworks. Fusion Embedded HTTP Server [19] supports only GET and POST methods. It may process multiple concurrent requests and main advantage is very small memory footprint. The Barracuda Web Server [20] is an industrial-strength, small embeddable web server engine that is optimized for compact, deeply embedded devices. Smews [21] is a prototype of very efficient and very small web server for WoT systems.

There are no known web servers that consider real-time constraints. The Chloe [22] which is called the realtime web server, deals with „real-time web", i.e. solutions that enable browsers to receive information as soon as it is published by its authors. The real-time web is fundamentally different from real-time computing since there is no knowing when, or if, a response will be received.

III. Real-time HTTP

HTTP is a stateless protocol for communication between a client and a server [Fie99]. An HTTP session consists of a sequence of request-response transactions. The client application sends a request, which may correspond to one of the following methods: GET, HEAD, POST, PUT, DELETE, OPTIONS, TRACE and CONNECT. As a response the server application sends back a message containing a status line and a requested resource. Not all methods are required to be implemented in the server application. Embedded systems usually use lightweight web servers where the most important is a small memory footprint, minimal CPU utilization and reliability.

Although some extensions for specification of time requirements were proposed [23], the HTTP protocol does not consider real-time requirements. Requests are processed in the FIFO order, regardless of the expected processing time. Thus it is not possible to predict when the client will receive the response, as well as it is not possible to define time constraints associated with the requests. The server latency depends on the number of requests, that must be processed, and on the processing time.

In order to enable specification of real-time constraints, we propose the real-time extensions to the HTTP protocol (RT-HTTP). All methods defined in the HTTP/1.1 are available in the RT-HTTP, moreover, three new headers (Hard Deadline, Soft Deadline and Remaining Time) as well as four new responses (120 Server Timeout, 220 Constraint Satisfied, 420 Wrong Deadline, 520 Deadlines Not Supported) are added.

A. Header "Hard Deadline"

Header "Hard Deadline" defines the hard time constraint, i.e. maximal time for processing the request by the server. This header may be specified only in requests. Hard constraint must not be violated, otherwise the server should return the response 120 (Server Timeout). When the request will be processed in time, then the server sends the 220 (Constraint Satisfied) response to the client.

The "Hard Deadline" request header is defined in RT-HTTP as follows:

Hard-Deadline := **Hard Deadline:** SP *time [***ms***]* CRLF

where: SP is a whitespace character and CRLF denotes

the end of line, *time* means a deadline value given in seconds (default) or milliseconds.
Assume that the IoT application traces bus positions in a public transportation system. Each bus is equipped with GPS and EWS supporting the current bus position. To get up-to-date bus position, information should be provided timely. Below, a sample POST request containing hard deadline, is shown:

POST /task/GetPosition HTTP/1.1
Accept-Language: pl-PL
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0)
Content-Type: text/plain
Accept-Encoding: text/html
Host: 198.51.100.0:80
Connection: Keep-Alive
Cache-Control: no-cache
Hard Deadline: 5

In the above example *GetPosition* is a servlet that has to send a response during 5 seconds after receiving the request. First, the server tries to schedule the *GetPosition* in such a way, that the servlet will finish its execution before the deadline. If it will be possible then the servlet will send the response 220 as a result of this request. A sample response may have the following form (the message body contains information about the current and next positions):

HTTP/1.1 220 Constraint Satisfied
Date: So, 22 lis 2014 20:01:05 CET
Content-Length: 87
Content-Encoding: aslam
Connection: close
Content-Type: text/html; charset=UTF-8
Server: HunterServer
<html> <body> current: Cracow, Warszawska: 45, change: Cracow, Szlak: 12</body> </html>

If it will not be possible to find the feasible schedule, then the server will send the response 120, then the request is discarded. A sample response may be as follows:

HTTP/1.1 120 Server Timeout
Date: So, 22 lis 2014 20:21:55 CET
Content-Length: 42
Content-Encoding: aslam
Connection: close
Content-Type: text/html; charset=UTF-8
Server: HunterServer
<html> <body>Server timeout</body> </html>

B. Header "Soft Deadline"

Header “Soft Deadline” also may be used only in requests. It defines the maximal time for processing the request, this deadline may be violated, but this results in degraded quality of service. For the same requests both soft and hard deadlines may be given. Server returns response 220 even when the soft deadline is violated. The “Soft Deadline” request header is defined as follows:

Soft-Deadline := **Soft Deadline:** SP *time[***ms***]* CRLF

Assume that the most precise results will be obtained if the response will be sent not later than after 2s. Hence, the request referring to the *GetPosition* additionally specifies a soft deadline. The request may have the following form:

POST /task/ GetPosition HTTP/1.1
Accept-Language: pl-PL
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0)
Content-Type: text/plain
Accept-Encoding: text/html
Host: 198.51.100.0:80
Connection: Keep-Alive
Cache-Control: no-cache
Soft Deadline: 2
Hard Deadline: 5

If it will be not possible to schedule *GetPosition* before the hard deadline then the response 120 will be sent. Otherwise, the response 220 will be sent, even when the soft deadline will be violated. The servlet may take into consideration this delay to produce appropriate results, e.g. it may modify the next position taking into consideration this delay:

HTTP/1.1 220 Constraint Satisfied
Date: So, 22 lis 2014 20:01:05 CET
Content-Length: 87
Content-Encoding: aslam
Connection: close
Content-Type: text/html; charset=UTF-8
Server: HunterServer
<html> <body> current: Cracow, Warszawska: 45, change: Cracow, Szlak: 18 </body> </html>

C. Header "Remaining Time"

Header “Remaining Time” is specified in responses that are sent as a result of processing real-time requests. It specifies the remaining time that may be used for sending the response to the client. This information may be useful for proxies or gateways. The “Remaining Time” response header is defined as follows:

Remaining-Time := **Remaining Time:** SP *time[***ms***]* CRLF

Fig. 1 presents the processing of deadlines by a proxy, a gateway and a server. First, the request R1 with deadline 5000 ms, is sent, we assume that the proxy processes this request during 200 ms, thus the deadline is modified. Next, the request is processed by the gateway, it will take 300 ms. After processing the request by the server that takes 4000 ms, the response 220 is sent to the server. Times of processing the response by the gateway and the proxy are equal 200 ms. Hence, the deadline for processing the request is not violated. In the second case, the request R2 is processed in time by the proxy, the gateway and the server, but the proxy is not able to process the response and the deadline was exceeded by 100 ms, therefore the response code was changed to 120.

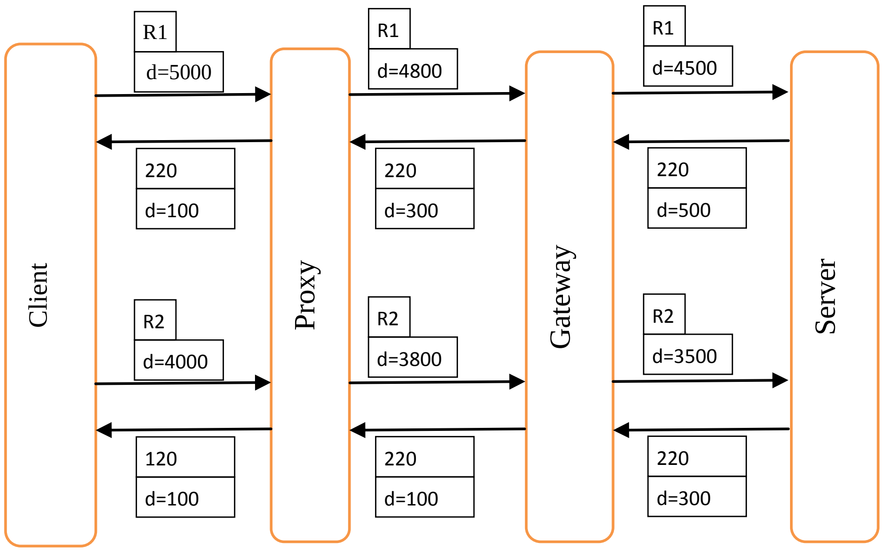


Fig. 1. Reduction of deadlines by intermediaries

Assume that the proxy is used for communication between a server and a client. To take into consideration delays caused by processing requests by the proxy, the server should use the “Remaining Time” header. Assume that the processing of a request was finished 100 ms before the hard deadline, then the server should send the following response:

HTTP/1.1 220 Constraint Satisfied
Date: So, 22 lis 2014 20:31:12 CET
Content-Length: 57
Content-Encoding: aslam
Connection: close
Content-Type: text/html; charset=UTF-8
Server: HunterServer
Remaining Time: 100 ms
<html> <body> current: Cracow, Warszawska: 45, change: Cracow, Szlak: 12</body> </html>

If the proxy is not able to process the response during 100 ms, then it should change the response to 120. A sample response may be the following:

HTTP/1.1 120 Server Timeout
Date: So, 22 lis 2014 20:31:54 CET
Content-Length: 40
Content-Encoding: aslam
Connection: close
Content-Type: text/html; charset=UTF-8
Server: HunterServer
<html> <body>Proxy timeout</body> </html>

Response 420 is sent when the value of the deadline is not valid, e.g. is a negative number or it is shorter than the time of processing the request by a server.

Response 520 is sent by servers which do not support real-time requests. The server may attach the proper message body, but it is not guaranteed that real-time constraints are satisfied.

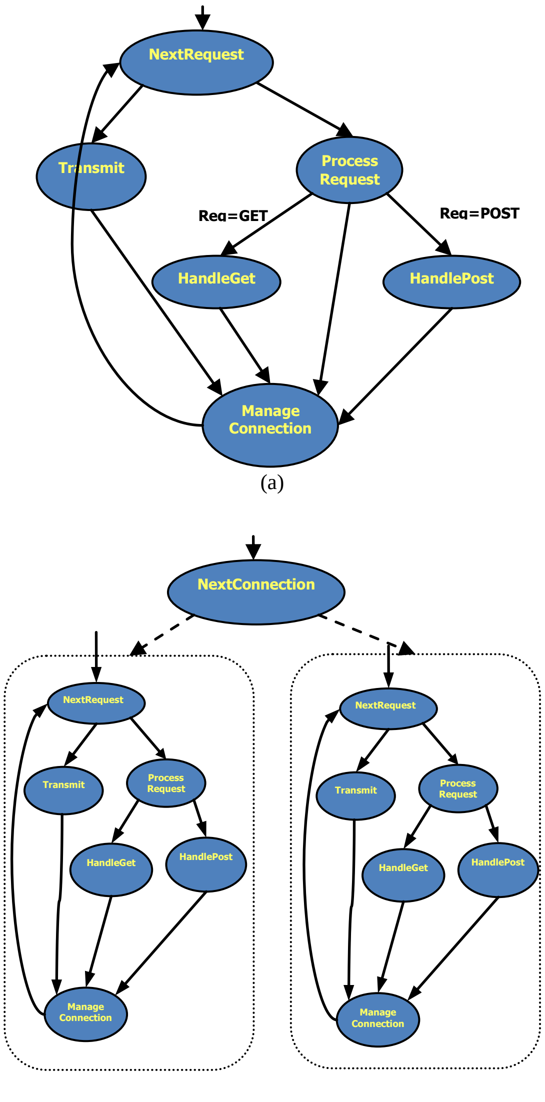
IV. Architecture of the Real-Time Embedded Web Server

A. *Single-thread and Multi-thread Architectures*

Architectures of the EWS and traditional web servers are similar. In both cases a server may be single or multi-threaded. But the multithreading enables simultaneous processing of multiple requests. Fig. 2 presents the architecture of single- and multiple-threaded web servers. The specification of the embedded HTTP server in the form of a task graph consists of the following tasks:

- **NextRequest** is the task that waits for the incoming requests appearing in HTTP socket. All received requests are stored in two queues, (described in the next subsection), where they are scheduled and then they are passed to task *ProcessReqest*. Task *NextRequest* is also responsible for scheduling the responses. The following responses are passed to task *Transmit*.
- **ProcessReqest** is the task that is activated whenever next request is ready for processing. It gets the next request, removes it from the queue, and processes it. Requests GET and POST are passes to tasks *HandleGet* or *HandlePost* for futher processing.
- **HandleGet** processes GET methods. Usually this task is implemented as servlet. The result is passed to task *ManageConnection* for completion of the final response.

- **HandlePost** processes POST methods. It is very similar to the *HandleGet* task.
- **ManageConnection** is the task which prepares the response for clients and defines the status of connection. In the case where the connection is timed out, the deadline is exceeded or if an error occurred while processing the command, buffers are reset and the connection is closed.
- **Transmit** is the process that transmits responses as HTML messages.



(b)

Fig. 2 Architecture of a single-thread (a) and multiple-thread (b) web server

Single-thread server (Fig. 2a) processes all requests sequentially. Next request can not be served until the server will finish processing the previous one. The efficiency of single-thread server may be increased by applying a pipelined or parallel architecture. In the multi-threaded server (Fig. 2b) for each client another thread is created (process NextConnection). Each thread processes requests sent by another client. When the connection is closed, then the corresponding thread is destroyed.

Regardless of the server architecture all requests are processed in the FIFO order. Therefore, in the case of a large number of simultaneous requests some clients may wait a long time for the response. When EWS supports services for IoT systems, the web application expects the response during the required time period. In some cases the time may be critical due to short period of the data validity e.g. when data contains the position of the mobile system. While in other cases this period may be longer. Thus, more suitable would be the scheduling of requests according to their deadlines.

One of the most important requirements for the EWS is a small memory footprint and low memory requirements. Therefore, single-threaded EWS, or at least server with limited number of threads, is more appropriate. In case of heavy duty EWSs multi-core embedded processors may be used to increase the sever throughput.

B. *Request Scheduling*

We assume that the times required to perform tasks, corresponding to processing the requests, are known. Therefore, scheduling method based on Least Laxity First (LLF) [24] may be applied. Our Real-Time Embedded Web Server (RTEWS) accepts requests defined by the extended HTTP protocol described in p. III.

First, the algorithm schedules requests containing real-time requirements using LLF method. Other requests are served meanwhile i.e. when there are no real-time requests. FIFO scheduling is used for this purpose. Scheduling is performed by the NextRequest process (Fig. 2). Since, requests may feed the server continuously, the process modifies makespan after receiving each request. All requests are scheduled in two queues: LLFQ and FIFOQ. Process ProcessRequest gets the next request from the LLFQ, or if this queue is empty, then from the FIFOQ, and sends it to the appropriate task for processing.

```
// algorithm starts each time when a new request $R_n$
//appears in time t.
Schedule($R_{n,}$ t)
  if (deadline($R_n$)!=0) then
     laxity($R_n$)= $t_{max}$($R_n$)-$t_{proc}$($R_n$);
     add $R_n$ to LLFQ in the order of ascending laxity
       taking R with hard deadlines first;
     for each $R_i$ with hard deadline in LLFQ do
          find processor P with min($t_{finish}$(P));
          if ($t_{finish}$(P)+ $t_{proc}$($R_i$)- $t_{arr}$($R_i$)<= $t_{max}$($R_i$)) then
              $t_{start}$(P,$R_i$)= $t_{finish}$(P);
              $t_{finish}$(P)= $t_{finish}$(P)+ $t_{proc}$($R_i$);
          else
              if ($R_i$ has hard deadline)
                send response 120;
                remove $R_i$ from LLFQ;
              else //soft deadline
                $t_{start}$(P,$R_i$)= $t_{finish}$(P);
                $t_{finish}$(P)= $t_{finish}$(P)+ $t_{proc}$($R_i$);
  else add $R_i$ to FIFOQ;
```

Fig. 3 Request scheduling algorithm

The draft of the scheduling algorithm is shown on Figure 3. R_n means the next request that arrived at time *t*,

t_{max}(R$_i$) is the deadline specified for request R_i, $t_{proc}(R_i)$ is the estimated time of processing the request R_i, $t_{finish}(P)$ is the finish time of all tasks currently assigned to processor P, $t_{arr}(R_i)$ is the time when request R_i arrived, $t_{start}(P,R_i)$ denotes the time when the processor P will start to process request R_i.

Since during processing the next real-time requests may arrive, the scheduler modifies schedule according to the LLF method. For each task the *laxity*= t_{max}-t_{proc} is computed. All tasks are scheduled according to the *laxity* (task with the lowest laxity is scheduled first). If for the given task it is not possible to find the feasible schedule, then the server sends the response 120 and the request is canceled.

Requests containing soft deadlines are scheduled in the second pass. The scheduler modifies the LLFQ by adding these requests. First it tries to find the schedule that satisfies soft deadline, if it is not possible, then the schedule such that the time exceed is minimal is chosen.

C. Example

The scheduling of requests in the RTEWS architecture will be illustrated with an example. Assume that the server may serve four types of requests, processing of each request is performed by another task (e.g. servlets). Average execution times of all tasks are shown in Table 1. It is assumed that clients send requests using standard methods POST and GET. Real-time requests will be denoted as POST$_{RT}$ and GET$_{RT}$. Let the scenario of arriving requests will be as shown in Table 2. The following columns contain: time of the request, request identifier, type of request, deadline t_{max} (for RT), constraint type (hard or soft), the task that is to be launched, the latency (t_{max}-t_{proc}).

Table 1. Average Execution Time Of Tasks By RTEWS

Task Name	Average execution time [ms]
T1	700
T2	1200
T3	1000
T4	300

The makespan is shown in Figure 4. Arrows below the Gantt chart denote the events, corresponding to requests which arrived at RTEWS. Arrows on top of the chart denote points in time where scheduling of newly arrived requests is performed. *Sch(Rx, Ry)* means the order in which requests *Rx*, *Ry* will be processed. We do not consider preemptive scheduling, hence newly arrived requests are scheduled after finishing the currently executed tasks, even if they have the higher priority then the request being processed.

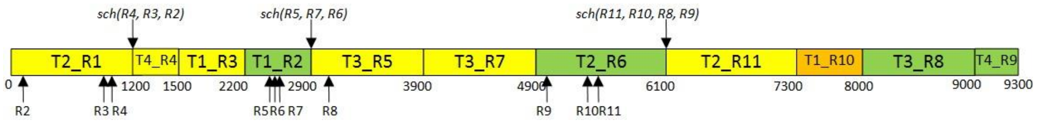


Fig. 4. The schedule of processing the requests in RTEWS

Real-time requests with imposed hard deadlines have the highest priority, thus they are scheduled first. If the deadline will not be satisfied, such request is not scheduled and it is missed. In the example, if request R10 had hard deadline, it would be missed, because it is not possible to find the proper schedule, in the best case it will exceed the deadline by 200ms. But actually, R10 has a soft deadline, thus it will be scheduled after request R11, which has a hard deadline. Requests that have no deadlines specified, have to wait for the execution of all handlers serving real-time requests. Therefore, the order of processing the requests is as follows. The R1 request has been received as first, at this time the server does not process any other requests, so the task T2 was launched immediately as the handler of this request. Next, requests R2, R3 and R4 arrive during the execution of T2, thus they are scheduled after finishing the T1. The order is R4, R3 and R2, because R4 has more stringent time constraint, i.e. t_{req}(R4) + laxity(R4) < t_{req}(R3) + laxity(R3) and R2 has no time constraints. The task serving request R2 is scheduled to execute after the processing of R3 and R4 will be finished, unless other real-time events will appear during this time. During the executing of T1, requests R5, R6, R7 arrived, they were scheduled according to their priorities. Request R8 arrived during

Table 2. Parameters of Requests Sent from the Client to the RT Server

Request arrive time [ms]	Request ID	Request type	Deadline [ms]	Constraint type	Task	Latency [ms]
0	R1	POST$_{RT}$	2000	hard	T2	800
100	R2	POST	-	-	T1	-
1100	R3	POST$_{RT}$	3000	hard	T1	2300
1120	R4	GET$_{RT}$	2500	hard	T4	2200
2500	R5	GET$_{RT}$	3000	hard	T3	2400
2600	R6	GET	-	-	T2	-
2610	R7	POST$_{RT}$	3000	hard	T3	2400
3000	R8	POST	-	-	T3	-
5000	R9	POST	-	-	T4	-
5300	R10	POST$_{RT}$	1300	soft	T1	600
5500	R11	GET$_{RT}$	2000	hard	T2	800

execution of task T3, but since this is no real-time request, there is no necessity for rescheduling after finishing T3, thus request R8 is simply added to the FIFOQ (after R6). Requests R8 and R9 were scheduled at the end, in order of coming, because they do not have a time constraint in the header, and R11 and R10 had to be executed first to satisfy their deadlines. Finally, our server processes all requests in the order that guarantees satisfying all hard real-time requests. It should be noticed that in case of FIFO scheduling, deadlines would not be satisfied for requests R7 and R11.

V. Experimental results

In order to demonstrate the practical benefits of using RTEWS for processing real-time requests, we have developed a lightweight server. Only methods GET and POST and headers defining hard deadlines were available. RTEWS was implemented in C++, in the MicroC/OS-II environment running on the Nios II processor built in Altera Cyclone II FPGA. Next, we performed experiments showing the effectiveness of request processing. The server was tested with increasing number of incoming requests per second. The test was performed in two passes. First, all requests were precessed in the FIFO order, like in existing web servers. During the second pass, our scheduling strategy was applied. The number of concurrent requests has been increased from 10 to 3000. For each case the number of canceled real-time requests were analyzed and counted. The results are presented in Figure 5. The x-axis shows the number of incoming requests and the y-axis the percentage of RT requests canceled by the server, because they do not meet deadlines. Server executed a few different tasks as handlers of the requests. The average execution time of these tasks was in the range from 100 to 3000 ms.The deadlines for RT requests were not higher than 3500 ms.

It may be observed that the number of missed real-time requests, obtained in the HTTP server was a few times larger than in the RTEWS. The number of missed RT requests in RTEWS did not exceed 5%, even for the big number of concurrent requests. For HTTP server the RT requests did not meet the time requirements in 40% of such requests. For high number of concurrent requests this percentage increased up to 80-90%.

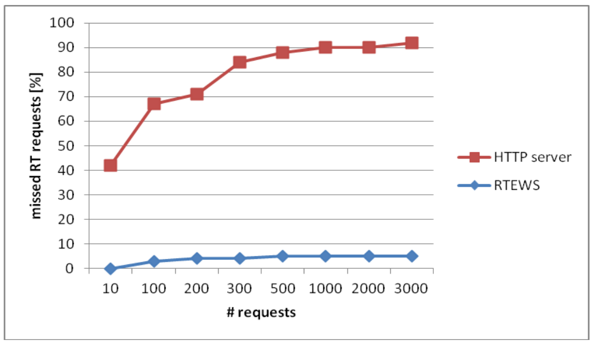


Fig. 5. The number of missed real-time requests.

VI. Conclusions

In this paper the architecture of embedded real-time web server was proposed. The server schedules HTTP requests taking into consideration hard and soft deadlines as well as the expected server load. Specifications of deadlines are possible by using the proposed extensions to the HTTP protocol. LLF scheduling of request handlers guarantees finding the proper schedule (if exists) and the high quality of service. If it is not possible to meet the deadline, the server cancels the request and sends the appropriate response to the client.

Experimental results showed that the number of missed real-time requests in the proposed server is a few times smaller than using the existing HTTP servers. Since in our approach requests that cannot be served in expected time period are canceled, thus the overloading or falling over of the server is avoided.

The server may be very useful in IoT applications that use embedded systems according to the Sensing as a Service business model. We believe that in the near future more and more embedded systems equipped with different sensors and located in different places will support real-time services for different web applications.

References

[1] Atzoria, L., Iera, A., Morabito, G.: The Internet of Things: A survey. Computer Networks, vol. 54, no.15, pp. 2787-2805 (2010).

[2] Hὂller, V. Tsiatsis, C. Mulligan, S. Karnouskos, S. Avesand, D. Boyle: From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence. Elsevier, 2014.

[3] Xiang Sheng; Jian Tang; Xuejie Xiao; Guoliang Xue, "Sensing as a Service: Challenges, Solutions and Future Directions", *IEEE Sensors Journal*, vol.13, no.10, pp.3733-3741, Oct. 2013.

[4] Dargie, W. and Poellabauer, C., "Fundamentals of wireless sensor networks: theory and practice", John Wiley and Sons, 2010.

[5] R. Fielding, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, "Hypertext Transfer Protocol - HTTP/1.1", IETF, RFC 2616, 1999.

[6] Guinard, D., Trifa, V., Mattern, F., Wilde, E.: From the Internet of Things to the Web of Things: Resource Oriented Architecture and Best Practices. In: Uckelmann, Dieter; Harrison, Mark; Michahelles, Florian (Eds.) Architecting the Internet of Things, Springer-Verlag Berlin Heidelberg (2011).

[7] H. Schulzrinne, A. Rao, R. Lanphier, *Real Time Streaming Protocol (RTSP)*, IETF, RFC 2326,1998

[8] Peter Saint-Andre, Kevin Smith, Remko TronconPeter Saint-Andre, Kevin Smith, Remko Troncon, XMPP: The Definitive Guide, O'Reilly Media, 2009.

[9] Crane, Dave; McCarthy, Phil: Comet and Reverse Ajax: The Next-Generation Ajax 2.0. Apress, (2008).

[10] F. M. Aymerich, G. Fenu, S. Surcis, "A real time financial system based on grid and cloud computing" *ACM symposium on Applied Computing*, March 2009, New York, pp 1219–1220.

[11] S. Liu, G. Quan, S. Ren, "On-Line Scheduling of Real-Time Services for Cloud Computing" *World Congress on Services*, July 2010, Miami, pp 459–464.

[12] W. Tsai, Q. Shao, X. Sun, J. Elston, “Real-Time Service-Oriented Cloud Computing” *World Congress on Services*, July 2010, Miami, pp 473–478.

[13] K. H. Kim, A. Beloglazov, R. Buyya, “Power-aware provisioning of cloud resources for realtime services” *International Workshop on Middleware for Grids, Clouds and e-Science*, 2009, New York.

[14] K. Kumar, J. Feng, Y. Nimmagadda, Y. Lu, “Resource Allocation for Real-Time Tasks using Cloud Computing” *International Conference on Computer Communications and Networks (ICCCN)*, July 2011, pp. 1-7.

[15] S. Bąk, R.Czarnecki, S. Deniziak, "Synthesis of Real-Time Applications for Internet of Things", LNCS vol. 7719, pp. 37-51, 2013.

[16] S. Bąk, R. Czarnecki, S. Deniziak "Synthesis of real-time cloud applications for Internet of things" *Turkish Journal of Electrical Engineering and Computer Sciences*, to be published, http://dx.doi.org/10.3906/elk-1302-178.

[17] Deniziak, S.; Bak, S., "Synthesis of real time distributed applications for cloud computing", Proc of the IEEE *Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp.743-752, 2014.

[18] Embedthis Software, Appweb Architecture, http://appwebserver.org/products/appweb/doc/ref/architecture.html, 2014.

[19] Unicoi Systems, Fusion Embedded™ HTTP, http://www.unicoi.com/product_briefs/http.pdf, 2013.

[20] Real Time Logic, Barracuda Web Server, https://realtimelogic.com/products/barracuda-web-server/, 2014.

[21] Duquennoy, S.; Grimaud, G.; Vandewalle, J.-J., "Smews: Smart and Mobile Embedded Web Server", *International Conference on Complex, Intelligent and Software Intensive Systems, 2009. CISIS '09*, pp.571-576, 2009.

[22] Trotter Cashion, Introducing Chloe - The Realtime Web Server, http://www.trottercashion.com/2011/06/13/introducing-chloe.html, 2011.

[23] S. Loreto, M. Thomson, G. Wilkins, “Hypertext Transfer Protocol (HTTP) Timeouts”, IETF, draft-loreto-http-timeout-00, 2010.

[24] G. C. Buttazzo *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers, 1997.

Authors’ Profiles

Radosław Czarnecki is Assistant Professor in Division of Computer Science in Faculty of Electrical and Computer Engineering, Cracow University of Technology, Poland. In 2002 he received MSc in Electrical Engineering from Cracow University of Technology, and in 2008 PhD degree in Computer Science from University of Zielona Góra.

The main profile of his research focuses on methodologies of designing a real-time Internet of Things systems. Previous researches were directed at synthesis methods for hardware-software distributed systems and reconfigurable embedded systems. He regular published scientific papers in embedded systems and synthesis methods of such systems in journals and conference papers and presented them on national and international meetings to promote the research. He is the author of two papers on the synthesis of real-time applications for Internet of Things in IEEE journals.

Stanisław Deniziak is Professor of Computer Science in Department of Computer Science, Kielce University of Technology, Poland. He received MSc in Computer Science from Warsaw University of Technology, and PhD degree from Gdańsk University of Technology. In 2006 he received DSc in Computer Science from Warsaw University of Technology. Now, he is Vice Dean for Research and Promotion of Faculty of Electrical Engineering, Automatics and Computer Science, Kielce University of Technology.

He has published 78 research papers in various international and national journals and conferences. He is active reviewer end editorial member of 7 international journals such Journal of Systems and Software, Computing, Microprocessors and Microsystems, International Journal of Applied Mathematics and Computer Science, The Open Cybernetics & Systemic Journal, International Journal of the Physical Sciences, Annales UMCS - Sectio A Informatica. He has reviewed research papers of many international conferences like: IEEE Design Automation Conference, International Conference of Computational Methods in Sciences and Engineering etc.

Prof. Deniziak is IEEE and IEEE Computer Society Member.

How to cite this paper: Radosław Czarnecki, Stanislaw Deniziak,"Embedded Real-Time HTTP Server", IJCNIS, vol.7, no.5, pp.1-8, 2015.DOI: 10.5815/ijcnis.2015.05.01