2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops

An Evaluation of the Performance of DPWS on Embedded Devices in a Body Area Network

Robert Hilbrich
Fraunhofer Institute for Computer Architecture
and Software Technology (FIRST)
Berlin, Germany
robert.hilbrich@first.fraunhofer.de

***Abstract*—In view of the aging society, intelligent devices pervading everyday life are faced with important challenges, such as the ease of use and the ease of configuration. The whole potential of using a body area network with several sensors to monitor vital functions of a human body can only be tapped, if the sensors used are highly specialized and tightly integrated to collaborate in a decentralized way and exhibit true plug-and-play behavior.**

Although the aspect of interconnecting vital sensors close to the human body bears a variety of technical challenges in itself, the development of the necessary abstraction layer in software to hide the heterogeneity of the highly specialized sensor boards is confronted with even higher challenges as these devices are often equipped with very limited resources to reduce power consumption. However, this abstraction layer is a necessary prerequisite to facilitate the development of software for a body area network with the previously mentioned characteristics.

This paper presents the results of a study conducted to evaluate the performance and overhead of using web services on embedded devices to implement an abstraction layer for a body area network. In several experiments, two different implementations of the Devices Profile for Web Services (DPWS) were evaluated: the Microsoft .NET Micro Framework and the open-source DPWS-plugin from the Web Services for Devices initiative. These were used to measure absolute latencies and the "web service overhead" in the communication between three different types of resource-constraint devices.

***Keywords*-Embedded SOA, Devices Profile for Web Services (DPWS), performance evaluation, body area network.**

I. INTRODUCTION

In light of an aging society, new technologies which allow elderly people to be under continuous medical observation while being in their home environment will become increasingly important. We assume that a body area network which incorporates several vital sensors and gateways nodes will play a key role in this development. Therefore, we conducted a study to evaluate the feasibility of a body area network dealing with the following requirements.

- Hardware heterogeneity of sensor nodes should be hidden to facilitate the development of a smart software layer.
- The body area network should support a dynamic discovery of nodes and a dynamic establishment of a workflow.

With regard to the first requirement, we chose a web service middleware approach based on the Devices Profile for Web Services (DPWS) [1]. DPWS is a subset of web service standards. It aims to simplify the development of lightweight web services on embedded device. A simple interaction to dynamically establish a workflow may then take place in the following way:

1) Discovery of a device with a desired "type" in the environment
2) Retrieval of metadata information, such as a list of active services and public interface specifications, from the device previously discovered
3) Usage of a previously discovered device

This three-step interaction pattern represents the level of flexibility for dynamic device interactions in a decentralized environment that can be achieved with DPWS today and is well suited for our concept of plug-and-play behavior in a body area network.

Problem Description

Generally speaking, using a middleware on embedded devices is often the result of a trade-off between reduced development effort and increased processing effort on the device. To support a sensible decision whether to use DPWS for body area network we

- analyzed the performance impact that is to be expected with this approach and
- analyzed performance bottle-necks with regard to further optimization potentials.

With this paper, we want to give an estimate on the performance that is achievable with this approach today by using low power off-the-shelf hardware components and generic web service toolkits that are freely available.

Related Work

R. Döring's work on the performance measurements of XML-RPC communications in a client-server environment [2] inspired the distinction between lightly and heavily structured parameters in this paper. H. Bohn et al. [3] announced the first DPWS-compliant protocol stack and presented results obtained in several SIRENA-project related demonstrators. E. Zeeb et al. [4] introduced a derived DPWS-toolkit based on C and gSOAP and discussed the relevance of web services for embedded devices with regard to industrial automation. They also described a potential "shift in the industrial landscape" of highly heterogeneous devices. S. Prüter et al. [5] used

978-0-7695-4019-1/10 $26.00 © 2010 IEEE
DOI 10.1109/WAINA.2010.93

IEEE computer society

the same DPWS-toolkit and analyzed its real-time capabilities on a single device type with regard to varying packet sizes. We use the same DPWS-framework and extend the previous work by a more detailed and profound performance analysis in addition to the comparison with another commercial DPWS-implementation.

II. Technical Background

To measure latencies in the communication between DPWS-enabled devices with regard to the pattern previously presented, we chose two different DPWS toolkits and three different embedded devices.

DPWS Toolkits

The following implementations of the DPWS protocol stack were used in our experiments: the Microsoft® .NET Micro Framework [6] and the WS4D DPWS-implementation for gSOAP [7]. The former is an efficient runtime environment based on the software platform .NET for embedded systems. It allows the development of applications in C#. The latter was initially created by the Web Services for Devices initiative (WS4D) and is published under an open source license. The DPWS functionality is implemented as a plug-in for gSOAP and allows the development in C and C++.

Embedded Devices and Network Setup

The performance of DPWS was evaluated on the following hardware platforms:

- **HiCO.ARM9** - a processor board with an AT91RM9200 @ 180 MHz, 64 MB memory and 100Base-T Ethernet
- **Tahoe Development Platform** - a processor board with an ARM920T @ 100 MHz, 8 MB memory and 10Base-T Ethernet
- **FOX Board LX832** - a processor board with an Axis ETRAX 100LX @ 100 MHz, 32 MB memory and 100Base-T Ethernet

In addition to these processor boards, two regular laptops - which we do not consider to be an embedded system - were also used to reasonably estimate the potential performance gain of faster processors or larger memory configurations.

- **Laptop-1GHz** - a Thinkpad T61 running at 1000 MHz with 2048 MB memory and 100Base-T Ethernet.
- **Laptop-2GHz** -a Thinkpad T61 running at 2000 MHz with 2048 MB memory and 1000Base-T Ethernet.

Due to several bugs in the firmware of the HiCO.ARM9, the only platform that allowed the analysis of the Microsoft® .NET Micro Framework was the Tahoe processor board. All other devices were tested with the Linux-based DPWS implementation from the WS4D initiative.

Although the reference scenario demands for a wireless connectivity, we decided to use a fixed *wired* connection to minimize the effects of interferences in the ISM band. Furthermore, all devices were directly connected to each other with a cross-over patch cable, so that all network transfers are single hop. Solely during the DNS tests, where we used the corporate network to reach a DNS server.

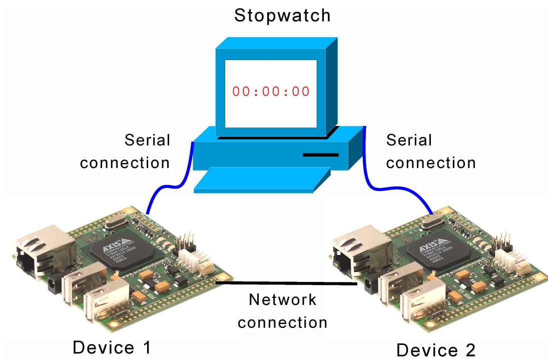


Figure 1. Measuring latencies in the communication between embedded distributed devices with our stopwatch-framework.

These hardware platforms for evaluation differ quite substantially in processor architecture (ARM9, ETRAX, x86), processing speed (100 MHz up to 2000 MHz) and communication interfaces (10 Mbit/s up to 1000 Mbit/s). Still, these were the only devices available to us in our experiments which were supported by at least one DPWS stack.

Test Environment and Latency Measurement

Analysis of the communication latencies required a fine grained time measurement in the communication of a distributed system with low overhead. Common approaches, such as clock synchronization and time-stamping of round-trip network packets, were neglected in favor of using a single "stopwatch".

Therefore, we developed a "stopwatch-framework" running on a separate computer - the "stopwatch". Both processor boards were connected to the stopwatch with a serial link (see Fig. 1). These serial links were then used to transport a "start" and a "stop" signal from the board to the stopwatch to trigger the time measurement.

The stopwatch computer is a regular workstation featuring an Intel® Pentium™ 4 processor running at 2 GHz with Microsoft® Windows XP™. The stopwatch-framework uses a special hardware-based high-resolution timer, so that it is - *theoretically* - accurate within 280 ns.

Albeit the accuracy of the internal timer, the time measurements were still noticeably affected by random measurement errors as a result of various scheduling effects in the operating system. To estimate the impact of these errors, we tested the accuracy of the stopwatch empirically with prior-known latencies.

Therefore two serial cables were connected to a single processor board which emitted a "start" signal over one cable, waited a pre-set time by using the `usleep(int s)` method and finally emitted a "stop" signal on the other cable. The expected time measurement (= `int s`) was compared to the actual measurement from the stopwatch. This was repeated 100 times for each time length within a range from 5 ms to 120 ms. It allowed an estimation of the maximum error $\hat{f}_s$ according to:

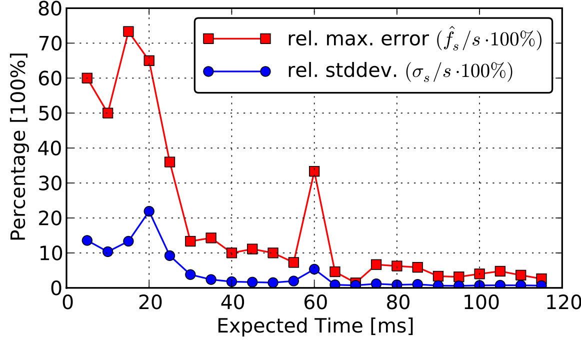


Figure 2. Assessment of the error in time measurement. The majority of the measurements longer than 60 ms can be trusted to be accurate within $\pm 1\%$.

$$\hat{f}_s = \max_{1 \leq i \leq n} \{d_i^s : d_i^s = |s - t_i^s|\}$$

$\hat{f}_s$ denotes the maximum error for the expected time length s. t_i^s represents the actual measurement from the stopwatch for the expected time measurement s and the iteration i. n denotes the total number of iterations - in our case $n = 100$.

Figure 2 contains the results expressed as a relative maximum error and a relative standard deviation. Based on these results, we estimate the accuracy of the stopwatch setup to be at least $\pm 10\%$ for times longer than 65 ms. With regard to the relative standard deviation we can assume that the majority of measurements beyond 65 ms reach an accuracy of less than $\pm 1\%$. As a consequence we used repetitions of test runs to guarantee results longer than 65 ms so that the accuracy condition is met.

III. Experiments and Results

In this section, we present the results from four different experiments. The first three experiments contain absolute latencies and the last experiment deals with the relative performance impact of DPWS.

A. Time to reach operational readiness

We define "operational readiness" as a specific state, which a device reaches when all necessary pieces of information have been acquired at runtime so that a remote service could be invoked. Several steps have to be accomplished to reach this state after boot-up:

1) "Probe" the network environment for the desired device (mandatory)
2) "Resolve" this device for a transport address (optional)
3) Do a "DNS query" of the transport address to get the network address (optional)
4) "Exchange metadata" to obtain the transport address of the hosted service (mandatory)

Step 2 and 3 are optional as the information may already be contained in the reply to the initial "probe" message. This is not standardized and depends on the DPWS implementation.

Table I
Time required to reach operational readiness (in ms).

Device	Probe	Resolve	DNS	Get	Min	Max
FOX LX832	1047	44	11	52	1099	1154
HiCO.ARM9	1022	15	7	18	1040	1062
Tahoe	1891	1663	125	681	2572	4360
Laptop-1GHz	1002	12	7	2	1004	1023
Laptop-2GHz	1001	6	6	1	1002	1014

We distinguish minimum delays (only mandatory steps) and maximum delays (all steps) which are required to reach the state of operational readiness. By using the previously described test environment we measured the time needed to accomplish each individual step.

Every measurement has been repeated 100 times. The MTU setting of the communication interfaces was set to the default value, which is 1500 bytes on Linux, but unknown for the .NET Micro Framework. The caching of address information was manually disabled in the source code of the WS4D toolkit. For the measurement of the DNS resolve latency, we used an internal DNS server, so that the measurements for this step are affected by other users, thus reflecting a real-world usage scenario.

Table I contains the results, which show the expected negative correlation between the amount of available resources and the observed delays.

The minimum and maximum delays on the Linux platform are largely dominated by the processing of the "probe" messages. This is particularly interesting, because the "probe" messages are on the one hand similar to the "resolve" messages as they are transmitted via IP multicast, but on the other hand the delay of the "resolve" messages takes about 1000 ms less. This may be a result of the multicast group membership implementation in the Linux kernel which implements an additional delay of 1000 ms (= 1 s) when joining a multicast group for the first time.

B. Service invocation

After the state of operational readiness has been reached and analyzed, we focused on the latency between the invocation on the consumer's device and the resulting indication on the provider's device (= "one-way call latency"). It is mainly influenced by the amount of "data" that is required to be processed and transmitted between the provider and the consumer.

In the case of DPWS, "data" mainly encompasses the call-parameters. On the consumer side, they are serialized in XML then transferred to the provider where they are deserialized and finally processed.

Inspired by the approach presented by R. Döring [2], we distinguish between heavily structured and lightly structured parameters to denote different ratios between XML metadata and "real" data in their serialized form. We used a set of integer parameters as a representative of the heavily structured parameters and a single string parameter with a variable length

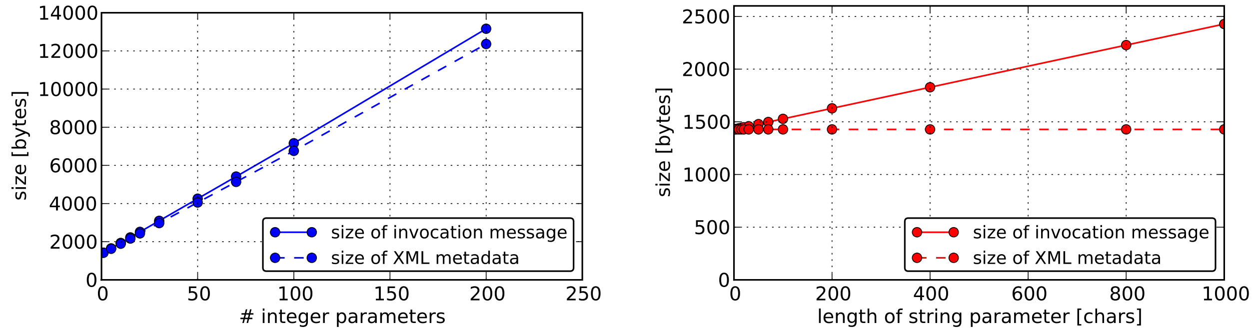


Figure 3. The size of heavily (left) and lightly structured parameters (right) in a service invocation message. The solid line shows the total size of the message, while the dashed line indicates the size for XML metadata, such as tags and namespace definitions. This shows that the structure of an XML message is directly related to its size and processing effort.

as a representative of the lightly structured parameters. The two parameter types represent "extremes" and their results can be interpreted as lower and upper bounds for call latencies.

The different ratios between XML metadata and "real" data can be seen in Fig. 3, which contains a comparison of the "invocation" messages when using either parameter type. The dotted line in Fig. 3 represents the overhead resulting from the XML serialization. For heavily structured parameters the size of the XML metadata increases with the number of parameters. With lightly structured parameters the size of the overhead is almost constant.

The results for a service invocation with heavily and lightly structured parameters are depicted in Fig. 4 and 5. Both confirm the expected negative correlation between available resources of the hardware platform and the latencies encountered in a service invocation.

The latencies for service calls using both parameter types cannot be directly related to each other. However, they can be compared on the basis of the amount of data transported within the parameters. All devices feature a 32-bit architecture on which a single integer value requires four bytes. Therefore, each integer parameter in the service invocation is able to transport four bytes of "raw" data. Assuming that each character of a string can be used to transport one byte of "raw" data, latencies between calls with lightly and heavily structured parameters can be compared with each other as shown in Fig. 6.

The distances between measurements depicted with the same color in Fig. 6 show the effect of the processing overhead of the XML metadata on the resulting latencies. It reflects the additional effort needed to process the invocation request.

These results show that the design of a web service interface has a significant implication on the performance achieved with an embedded SOA approach. They also show that optimizing a set of web services - especially for embedded devices - may benefit from a redesign of the interface and the usage of parameters with less XML metadata overhead to process and transmit.

C. *Effect of a reduced MTU*

Messages between devices are transfered over the network as payload in IP packets. An upper bound on the size of an IP packet is given by the maximum transfer unit (MTU). If the size of an IP packet exceeds the MTU setting, its content is split into several packets ("fragmentation"). Handling fragmented IP packets requires more packets to be transmitted and processed so it adds to the total latency.

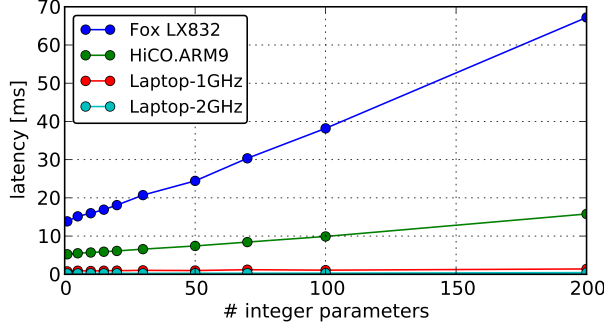
Many promising network technologies for body area networks, such as Bluetooth low energy (MTU between 675 bytes and 48 bytes) or 6LoWPan (MTU approx. 150 bytes), feature a reduced MTU setting. These are optimized for smaller packets and low energy consumption. This optimization increases the probability of fragmentation for larger datasets, such as XML data.

To obtain an estimate on the latency of a service invocation on low power networks, we measured the call latencies with different MTU settings. Each service invocation was repeated 100 times for each MTU setting on each board. The Tahoe board did not offer an API to configure the MTU, so that it was not included here. The results are depicted in Figure 7.

The impact of the MTU setting becomes significant with fewer resources available and more data transmitted. MTU settings larger than approx. 512 bytes appear to have little effect on the latency, while MTU setting below 200 bytes noticeably increase the latency in the communication. These results indicate that the verbosity of XML is not suited for low power networks with low MTU settings as it significantly affects the performance. In addition to the MTU-related effects, differences in bandwidth and medium access protocols of wireless networks will affect the total latency even further.

D. *DPWS's share of the total latency*

In our last experiment, we studied the overhead of the DPWS layer relative to "pure" network communication. Therefore we distinguished only between two simplified layers: the DPWS layer and the network layer below. The former mainly contains the XML-marshaling capabilities and the



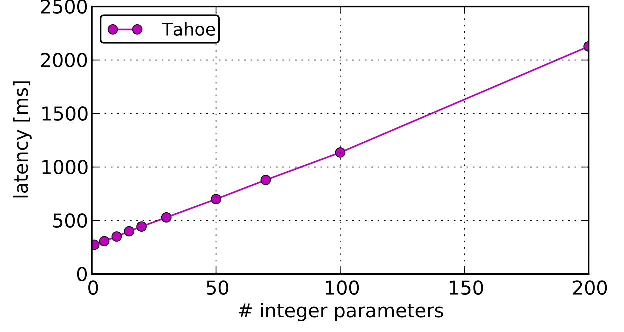
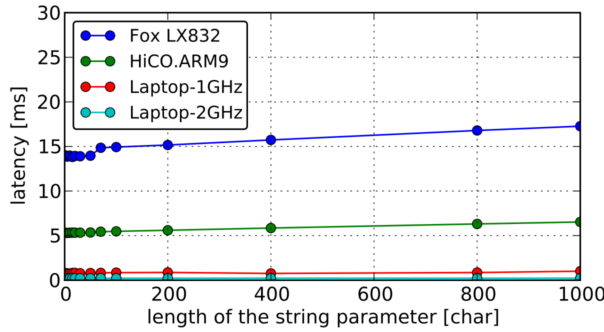


Figure 4. Absolute latencies of a service invocation with a varying number of heavily structured parameters (one-way call). This represents the time from the service invocation on the consumer side until this request is indicated on the provider side.



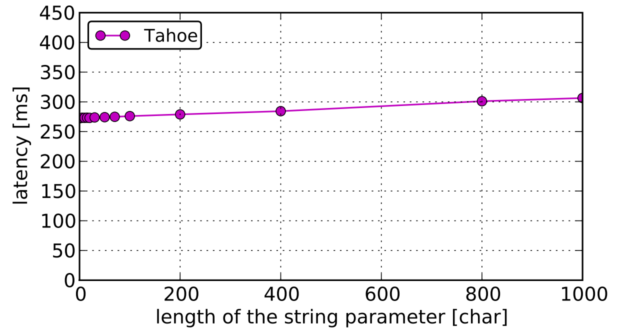
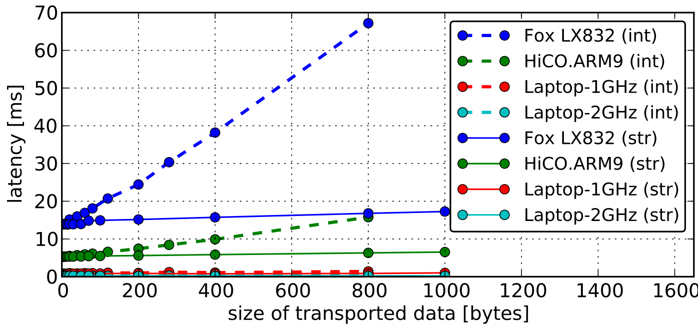


Figure 5. Absolute latencies of a service invocation (one-way call) with lightly structured parameters represented by a single string parameter with a variable length.



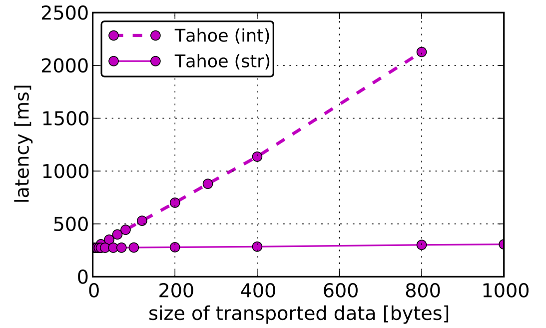


Figure 6. Comparing the latencies of a service invocation with heavily (dashed line) and lightly structured parameters (solid line) on the basis of the transported data. The gap between line of the same color indicates the latency which is induced by the additional processing effort as a result of the inherent structure of the parameters.

latter comprises of the functions to transfer data between devices over the network ("sockets").

In our experiments, we determined the size of each invocation message and measured the time needed to transfer an equally sized byte array by using only socket communication ($= t_{net}$). With the results from section III-B ($= t_{total}$), we were calculated the share of DPWS layer of the total latency according to:

$$c_{DPWS} = (t_{total} - t_{net})/t_{total} \cdot 100\%.$$

The results are depicted in Figure 8 and show that the majority of the latency is generated within the DPWS layer. Even the Tahoe board with a very slow 10 Mbit connection exhibits a c_{DPWS} of approx. 90%.

Surprisingly, the Fox LX832 with a slow 100 MHz proces-

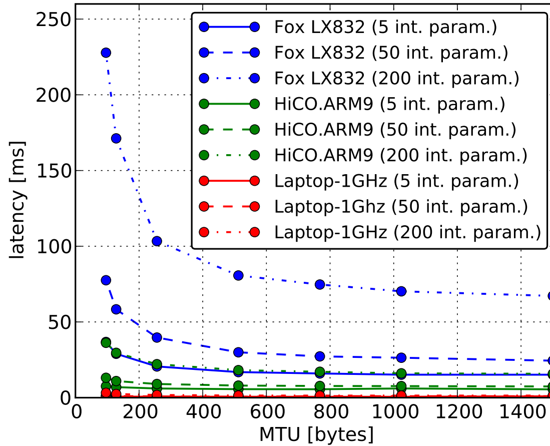


Figure 7. Effects of different MTU settings on the latencies in a service invocation message exchange

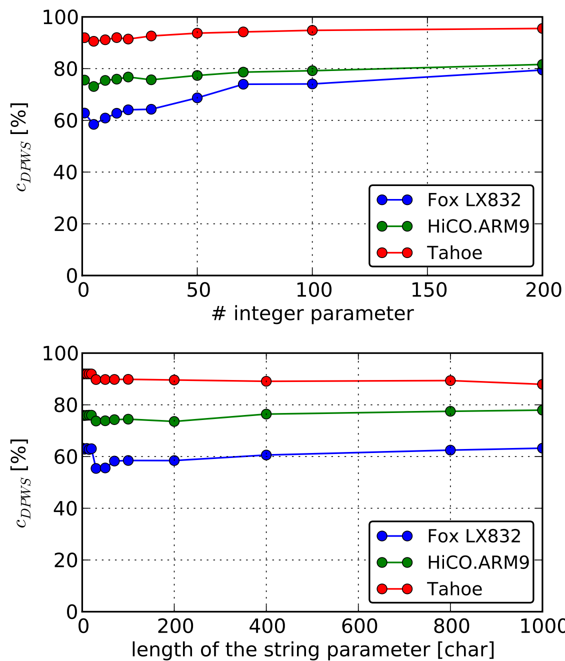


Figure 8. The share of DPWS of the total latency when invoking a web service with heavily (top) and lightly (bottom) structured parameters.

sor spends relatively less time in the DPWS layer compared to the HiCO.ARM9 board with a 180 MHz processor, although both comprise a 100 Mbit connection.

IV. Conclusion

We have evaluated the applicability of DPWS as a middleware on several embedded devices and have shown that its support for dynamic discoveries and workflows is beneficial for a plug-and-play behavior of nodes in a body area network. Flexibility and hardware abstraction are on the other hand intrinsically tied to increasing processing costs and prolonged latencies in the communication - especially on embedded devices with limited resources.

First, we described our approach to measure latencies in the communication by using a central "stopwatch". Then we presented the results gained from several experiments which reflect the characteristics of a body area network. We showed that after reaching a state of operational readiness, which takes about 1 s, the latency for a service invocation is significantly influenced by the inherent "structuredness" of the parameters. Interfaces for web services running on embedded devices should thus be also designed with regard to the performance impact.

We also discovered that the majority of the latency is produced within the DPWS layer, so that a faster processor promises better results in comparison to a faster network connection.

The question whether DPWS is a sensible choice in a body area network depends on the specifics of the use case. DPWS on embedded devices is well suited to support slowly changing topologies of heterogeneous nodes with rare transmissions. Especially the use of freely available toolkits may drastically simplify the development.

Scenarios with continuous streams of data transmissions combined with rapidly changing topologies and very short maximum response times are on the other hand not well suitable for DPWS. Especially regarding these shortcomings of DPWS, we plan to evaluate other middleware approaches, such as UPnP and Jini, in a similar manner in the future.

References

[1] S. Chan, D. Conti, C. Kaler, T. Kuehnel, A. Regnier, B. Roe, D. Sather, J. Schlimmer, H. Sekine, J. Thelin, D. Walter, J. Weast, D. Whitehead, D. Wright, and Y. Yarmosh, *Devices Profile for Web Services*, Feb. 2006. [Online]. Available: http://schemas.xmlsoap.org/ws/2006/02/devprof/

[2] R. Döring, *Performance of XML-RPC*, Sep. 2001. [Online]. Available: http://www.netobjectdays.org/www_old_netobjectdays_org/pdf/01/papers/ws-mik/doering_mik2001.pdf

[3] H. Bohn, A. Bobek, and F. Golatowski, "Sirena - service infrastructure for real-time embedded networked devices: A service oriented framework for different domains," in *ICNICONSMCL '06: Proceedings of the International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies*. Washington, DC, USA: IEEE Computer Society, 2006, p. 43.

[4] E. Zeeb, A. Bobek, H. Bohn, and F. Golatowski, "Service-oriented architectures for embedded systems using devices profile for web services," in *AINAW '07: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 956–963.

[5] S. Prüter, G. Moritz, E. Zeeb, R. Salomon, F. Golatowski, and D. Timmermann, "Applicability of web service technologies to reach real time capabilities," in *ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 229–233.

[6] D. Thompson and C. Miller, *Introducing the .NET Micro Framework: Product Positioning and Technology White Paper*, Sep. 2007. [Online]. Available: http://download.microsoft.com/download/a/9/c/a9cb2192-8429-474a-aa56-534fffb5f0f1/.NET%20Micro%20Framework%20White%20Paper.doc

[7] A. Bobek, *WS4D - About*, 2008. [Online]. Available: http://ws4d.e-technik.uni-rostock.de/?page_id=6