Investigation of the Use of Embedded Web Services in Smart Metering Applications

Vlado Altmann, Jan Skodzik, Frank Golatowski and Dirk Timmermann
University of Rostock
Institute of Applied Microelectronics and Computer Engineering
18051 Rostock, Germany
Tel./Fax: +49 (381) 498-7257 / -1187251
Email: vlado.altmann@uni-rostock.de
Web: http://www.imd.uni-rostock.de/networking

***Abstract*—The actual situation in smart metering is characterized by a coexistence of a large number of proprietary and open standards for wired as well as wireless communication. These standards show low or no interoperability to each other. Therefore, it is very difficult to integrate multi-vendor solutions using one sustainable holistic approach. The proposed approach is to use Web Service technology as an open widespread Internet standard for the creation of a heterogeneous network for smart metering devices. Smart metering is an emerging topic for realizing modern energy policies. Monitoring, analyzing, and controlling of power consumption is the precondition for optimizing energy strategies. This work focuses on the integration of Web Service technology in the area of smart meter communication. Furthermore, mechanisms for reducing Web Service traffic overhead by over 90% is presented.**

Index Terms*—*Smart Grids, Smart Metering, Web Services, Home Automation.

I. Introduction

Recently, the rapid growth in the smart metering sector led to a development of many new specific protocols. However, most newly created protocols are neither interoperable with each other nor with already existing standards. Furthermore, the specific protocols are suited to a narrow range of applications making them not future-proof. In contrast, Web-based technologies continuously penetrate other spheres of activity and thus improve their sustainability. As an example, the public switched telephone network is continuing to be suppressed by the voice over Internet protocol technology (VoIP). Web Services (WS) are an approved Internet standard for information interchange. The dynamic structure and the modular design principle are making WS technology fittable to almost every needs. Moreover, the specifications of various WS standards are publicly available.

Prospectively, energy networks have to become smarter to meet all challenges of tomorrow's energy policies. New service-based infrastructures are required to optimize energy consumption and to realize energy efficiency regarding the special needs of green and sustainable energy management. The first step towards this goal is to replace passive meter devices by smart meters using a technology called Automated Meter Reading (AMR). AMR allows suppliers to remotely read meters and collect measured values in an electronic form according to different accounts for billing purposes.

Lately introduced AMR protocols such as Device Language Message Specification/Companion Specification for Energy Metering (DLMS/COSEM), Smart Message Language (SML), METERS&MORE, and others show no interoperability with each other although they serve a similar purpose. The lack of interoperability between different systems can be solved using WS technology. The Devices Profile for Web Services (DPWS) [1] enables WS capabilities on small and resource constrained devices such as energy meters. In order to adapt WS to some specific application, a so called profile can be defined. A profile describes, which standards are used for the specific application. By this means, a WS profile for smart metering can be defined. In this work, a possibility to map existing smart metering standards onto WS is investigated. Moreover, mechanisms for the reduction of WS communication overhead by over 90 % is presented.

The remainder of this paper is structured as follows. Section II gives a short overview about related work in the field of smart meter networks. Section III describes the idea behind DPWS. In Section IV, the SML protocol is introduced. A smart meter profile for embedded WS is described in Section V. Implementation details are presented in Section VI, followed by the conclusion and future work in Section VII.

II. Related work

Several smart meter pilot projects already took place in Italy, Sweden, France, the Netherlands, Greece, Germany, and other countries [2]. These projects are supported and enforced in most cases by the respective government. The Italian project Telegestore was introduced in 2001 and focused on the creation of a smart meter infrastructure for about 30 million residential consumers helping to manage peak demands and allowing to improve coping with billing debtors [3]. In addition to an enhanced version of the Lontalk protocol, a proprietary high level data link control protocol (HDLC) called SITRED was introduced [4]. In Sweden, the government stimulated smart metering through strong legislation. The evolved smart metering system is based on the Lontalk network protocol developed by Echelon Corporation [5]. In Germany, a new protocol called Smart Message Language was introduced for AMR. The development was supported by several energy corporations such as EnBW, E.ON, RWE, and other companies. However, the roll-out of smart meters in Germany was slowed down due to the lack of communication security. The German federal office for information security (BSI) introduced a protection profile for the gateway of a smart metering system [6]. The proposed gateway must support 3 smart metering protocols such as M-Bus, DLMS/COSEM and SML. Moreover, according to the BSI security profile the messages from smart meters to the secured gateway must be either secured with Transport Layer Security (TLS) or with a symmetric encryption method and an additional signature.

Today, IP is the most commonly used communication protocol for networking technologies. The main benefit using IP is that every device can be addressed independently of the underlying communication technology. IP covers wired (e.g., LAN) as well as wireless (e.g., WLAN, HSPA, UMTS) up to low power wireless network technologies (e.g., 6LoWPAN). Moreover, IP was approved as a smart grid standard by National Institute of Standards and Technology (NIST) [7].

The approach suggested in this work is based on IP and WS and thus can simply be deployed on every existing IP network. Moreover, service oriented architecture is recommended for use in smart grid by the International Electrotechnical Commission (IEC) [8]. The purpose of this work is to show the capabilities of WS for embedded environments. The suggested embedded WS profile does not introduce a new smart metering protocol but an adoption of existing open standards to smart metering needs. It provides a scalable architecture, full interoperability between metering equipment, secure data exchange independent of communication media, and thus complies with EU smart metering mandate M/441.

DPWS was already considered for smart metering in [9]. The authors provide simulation results for timing behavior in an event-based infrastructure. In contrast, this paper focuses on a DPWS profile for smart metering addressing security considerations and WS communication overhead.

III. Devices profile for Web Services

DPWS is a collection of WS standards developed to enable secure WS capabilities on resource constrained devices [10]. DPWS consist of WS-Discovery, WS-Eventing, WS-MetadataExchange, WS-Policy, WS-Transfer, and WS-Addressing (WSA). DPWS is a base technology for inter-device communication, which can easily be composed with and extended by other specifications [11]. DPWS slightly differs from the standard WS architecture to enable device centric service-oriented architectures and infrastructures. One of the major differences between standard WS and DPWS is the multicast service discovery with WS-Discovery that does not require any central service registry such as Universal Description, Discovery and Integration (UDDI) and, therefore, supporting the decentralized nature of device centric networks such as Wireless Sensor Networks (WSN), Home Automation Networks (HAN), and Smart Meter Networks.

The provided service itself is invoked through a standardized interface, fully compatible with standard WS architecture, thus providing a high level of interoperability and Plug&Play capabilities among DPWS-enabled devices (e.g., smart meter) and between DPWS-enabled devices and business WS (e.g., supplier services). Figure 1 shows the DPWS protocol stack. Based on IP technology, the core DPWS protocol is SOAP with its glue specifications SOAP-over-X, where X can be UDP, TCP but also uncommon protocols like SMTP. SOAP itself is based on XML. The Web Service Description Language (WSDL) is used for service description.

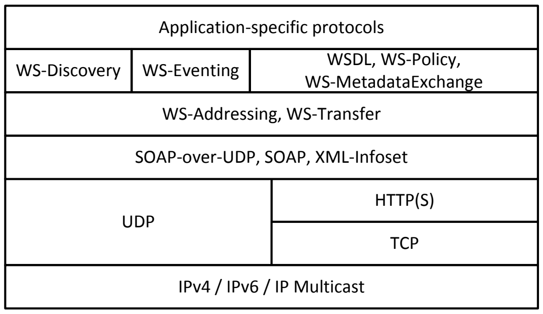
Application-specific protocols
WS-Discovery
WS-Eventing
WSDL, WS-Policy, WS-MetadataExchange
WS-Addressing, WS-Transfer
SOAP-over-UDP, SOAP, XML-Infoset
UDP
HTTP(S)
TCP
IPv4 / IPv6 / IP Multicast

Fig. 1. DPWS protocol stack

An important DPWS feature is the WS-Eventing protocol. WS-Eventing defines publishing and subscription mechanisms, whereas services may subscribe to and may accept subscriptions from other WS. WS-Eventing is used to provide an asynchronous notification mechanism.

IV. Smart Message Language

SML is a communication protocol for AMR. It was primary developed for electricity meters. The basic structure of SML is organized into following elements: data structure for payload transmission, SML binary encoding, SML XML encoding, and SML transport protocol. The latter is used only for point-to-point connections. SML is basically an application layer protocol. For data transport, any standard protocol such as TCP, Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and others can be used. The transported data is modeled as files. An SML file represent a sequence of messages. Three types of SML files are specified: SML request file, SML response file, and SML combined file. SML request files contain requests, SML response files include responses, and SML combined files can contain both. Each SML file begins with an OpenRequest or OpenResponse message and ends with a CloseRequest or CloseResponse message respectively. Between them, any number of request and response messages can be placed. Following messages are further specified: GetProfilePackRequest/Response, GetProfileListRequest/Response, GetProcParameterRequest/Response, GetListRequest/Response, SetProcParameterRequest/Response, and AttentionResponse. All message type descriptions can be found in [12].

In this work, the GetProfileListRequest and the corresponding GetProfileListResponse messages are explained exemplar-

MESSAGE FIELD	DATA TYPE
transactionId	String
groupNo	Unsigned8
abortOnError	Unsigned8
messageBody	SML message body
crc16	Unsigned16
endOfSmlMsg	EndOfSmlMsg

TABLE I
SML MESSAGE STRUCTURE

MESSAGE FIELD	DATA TYPE	Optional
serverId	String	Yes
username	String	Yes
password	String	Yes
withRawdata	Boolean	Yes
beginTime	Unsigned32	Yes
endTime	Unsigned32	Yes
parameterTreePath	String list	No
object_List	String list	Yes
dasDetails	SML tree type	Yes

TABLE II
GETPROFILELISTREQUEST MESSAGE

ily. All SML messages show the same structure as depicted in Table I. *TransactionId* is created by a requester. It is then used by a responder in order to assign responses to requests. With the aid of *groupNo*, message groups can be built. With this parameter, a processing order of messages can be specified. *abortOnError* indicates the behavior of further processing in case of errors. The body of the message is placed in *messageBody* field. In order to detect transmission errors, a check sum is present in *crc16* field. *EndOfSmlMsg* indicates the end of the message.

The GetProfileListRequest message is used to request the measured values. In the response, the values are represented as a simple list. The GetProfileListRequest message body structure is depicted in Table II. *serverId* denotes addressed SML data source or specific software module. *WithRawdata* allows additional raw data to be sent. *BeginTime* and *endTime* indicate a measuring period. *ParameterTreePath* denotes the requested measurements. The measurements can be read from different channels specified in *object_List*. With *dasDetails*, additional request parameters can be taken into account.

If the request was successful, the requested values are transmitted in the GetProfileListResponse message. The structure of this message body is shown in Table III. The field *actTime* indicates the time when the server started to process the request. The duration of the current log period is present in *regPeriod*. The field *valTime* provides time of value generation. Additional raw data can be sent in a string form with *rawData*. For integrity check, a signature of the message can be provided with *periodSignature*. The requested measurements is sent as a list in *period_List*. The measured values is sent together with a value name, a unit, a scaler, and an optional value signature. The structure and description of other messages can be found in [12].

MESSAGE FIELD	DATA TYPE	Optional
serverId	String	No
actTime	SML time	No
regPeriod	Unsigned32	No
parameterTreePath	String list	No
valTime	SML time	No
status	Unsigned64	No
period_List	SML period list	No
rawData	String	Yes
periodSignature	String	Yes

TABLE III
GETPROFILELISTRESPONSE MESSAGE

V. SMART METER PROFILE FOR EMBEDDED WEB SERVICES

In this section, the possibility to map newly developed protocols onto existing WS standards is shown on the basis of the SML protocol and DPWS. In SML, the messages are part of a file. In the WS world, a message is a stand alone entity. Thus, to keep interoperability with WS standards as well as to gain more benefit from WS technology, the SML file is broken up and its parts are mapped to WS standards. Hereby, one WS message represents a request or a response and corresponds to an SML file with only one request or response message. This change has impact only on files with different message types. Multiple measurements can still be retrieved through one request message. The suggested approach is explained by taking the example of GetProfileListRequest. The functionality of the GetProfileListRequest message can be reflected by HTTP, SOAP, and WS-Addressing. First, HTTP is considered. *ServerId* corresponds to the resource path on the server. In the WS world, it is represented by the Uniform Resource Identifier (URI). This representation has the structure *serverHost:serverPort/serverId*. URI is placed in HTTP as shown below:

```
POST /serverId HTTP/1.1
Host: serverHost:serverPort
```

The fields *groupNo* and *abortOnError* can be omitted as only one message is present in a request or a response. Message type and *transactionId* are placed into SOAP header with the aid of WS-Addressing as follows:

```
<soap:Header>
  <wsa:Action>messageType</wsa:Action>
  <wsa:MessageID>transactionId</wsa:MessageID>
</soap:Header>
```

The *Action* tag indicates the content of the SOAP body, i.e., GetProfileListRequest operation. In order to provide authorization by means of username and password WS-Security standard is utilized. This kind of authorization is only applicable for encrypted communications. An example of WS-Security is presented below:

```
<soap:Header>
  <wsse:Security>
    <wsse:UsernameToken>
      <wsse:Username>username</wsse:Username>
      <wsse:Password>password</wsse:Password>
    </wsse:UsernameToken>
  </wsse:Security>
</soap:Header>
```

The remaining parameters of the GetProfileListRequest are placed into SOAP body. For simplicity, only mandatory fields

SML ATTRIBUTE	BASIC STANDARD
ServerId	HTTP
TransactionId	WS-Addressing
Message type	WS-Addressing
Username/Password	WS-Security
Signature	WS-Security
Other attributes	SOAP

TABLE IV
MAPPING OF SML ATTRIBUTES ONTO WEB SERVICES

ENTITY	DATA SIZE	OVERHEAD
SML GetProfileListRequest	52 byte	55.8 %
SML request file	124 byte	81.5 %
SML GetProfileListResponse	104 byte	38.5 %
SML response file	176 byte	63.6 %
DPWS request	708 byte	96.8 %
DPWS response	1001 byte	93.6 %

TABLE V
COMMUNICATION OVERHEAD COMPARISON BETWEEN SML AND DPWS

are considered below. The optional fields can be added analogously. For example, an SML file with two parameter requests is assumed. Thus, *parameterTreePath* field of the request can be reflected in a SOAP body as follows:

```
<soap:Body>
  <sml:GetProfileListReq>
    <sml:parameterTreePath>valueRequest1</sml:parameterTreePath>
    <sml:parameterTreePath>valueRequest2</sml:parameterTreePath>
  </sml:GetProfileListReq>
</soap:Body>
```

After receiving an operation invocation, the server generates an invocation response. The corresponding invocation response to this request is also transported in a SOAP body:

```
<soap:Body>
  <sml:GetProfileListRes>
    <sml:actTime>time</sml:actTime>
    <sml:regPeriod>period</sml:regPeriod>
    <sml:parameterTreePath>valueRequest1</sml:parameterTreePath>
    <sml:parameterTreePath>valueRequest2</sml:parameterTreePath>
    <sml:period>
      <sml:objName>name1</sml:objName>
      <sml:unit>unit1</sml:unit>
      <sml:scaler>scaler1</sml:scaler>
      <sml:value>value1</sml:value>
    </sml:period>
    <sml:period>
      ...
    </sml:period>
  </sml:GetProfileListRes>
</soap:Body>
```

Any other messages and parameters can be composed in the same way. SML message fields *crc16* and *endOfSmlMsg* are not reflected by WS as they are already present in lower layers. Thus, the complete SML communication can be mapped onto existing WS standards. The suggested mapping of the SML onto DPWS is presented in Table IV.

Moreover, SML profile for WS can be extended with the standard DPWS features such as device and service discovery. Thus, available features of specific device can be simply discovered. Basically, all requirements on smart meters stated in Section II can be satisfied with DPWS. Especially, complex mechanisms such as error reporting and fraud detection can be easily solved with WS-Eventing.

A. Evaluation of DPWS profile

In [13], the authors showed the possibility to bring DPWS into constrained devices. The developed prototype requires 46 KB ROM and 7 KB RAM and thus can run on a mote. The main problem of WS approach in constrained environments still remains high communication overhead. Communication overhead over a broadband is not an issue. However, in the area of smart metering, power line communication (PLC) must be taken into account. PLC provides much less bandwidth and thus the traffic overhead is undesired. The highest overhead occurs when only one value, i.e., an integer has to be transmitted. In order to compare communication overhead between SML and DPWS, open source implementations jSML from Fraunhofer Institute for Solar Energy Systems and Java Multi Edition DPWS Stack (JMEDS) from Web Services for Devices (WS4D) were used. For comparison purpose, further assumptions related to transmitted values are necessary. Following assumptions were made: *Action* - 4 Byte, *serverId* - 6 Byte, *parameterTreePath* - 17 Byte. The length of the string of *parameterTreePath* corresponds to Object Identification System hex notation (6 groups with separators).

Let us assume a client (gateway) requests current power consumption from a server (meter). In SML, this request is sent as a request file with GetProfileListRequest message. In DPWS, this request is represented by an invocation of the GetProfileListReq operation. The server responds with a response file including GetProfileListResponse message or with an invocation response respectively. All optional fields excluding *serverId* are omitted to produce the worst case overhead. An overhead rate can be calculated with the Formula 1.

$$OverheadRate = \frac{PacketContent - Payload}{PacketContent} \qquad (1)$$

Since, SML as well as DPWS use the same transport layer protocol, a packet content is considered as TCP payload. Request payload is the *serverId* and the *parameterTreePath*. Response payload are all mandatory values of the GetProfileListResponse. The resulted overhead calculation is provided in Table V. As expected, DPWS produces very high communication overhead due to the text-based nature of HTTP and XML encoding of SOAP. This makes WS approach hardly applicable in constrained environments such as smart metering. For traffic reduction, compression methods can be applied. Standard HTTP compression methods such as gzip or deflate are ineligible because they produce additional significant CPU load. Compression mechanisms that can reach high compression rates without causing additional CPU load are presented in the following.

B. Compression

In order to compress HTTP headers, Constrained Application Protocol (CoAP) is suggested. At the time of writing, CoAP is available as a draft in version 0.9 [14]. However, it is on the way to become standard. CoAP was developed for

NUMBER	NAME	FORMAT
1	Content-Type	uint
5	Uri-Host	string
7	Uri-Port	string
9	Uri-Path	string

TABLE VI
MINIMUM SET OF COAP OPTIONS

information exchange in sensor networks. It has HTTP-alike structure so that HTTP can be easily mapped onto CoAP. In contrast to HTTP, CoAP is a binary encoded protocol. The possibility of the SOAP over CoAP binding was already presented in [15]. Natively, CoAP supports only UDP. However, TCP binding is left open for future specifications. In order to satisfy security demands mentioned in Section II, TCP binding is proposed. Hereby, TLS can be used for data transport security.

As TCP is a connection oriented protocol that ensures successful message delivery, CoAP over TCP is used in a non-confirmable mode. The length of the message in the CoAP over UDP binding is derived from the UDP header. Since TCP header does not have a special field for the message length, it must be reflected in the CoAP header. The minimal CoAP header is 4 byte. It consists of version number, type, option count, code, and the message ID fields. The type field specifies the type of the message and the code indicates whether the message carries a request or a response. Message ID is a 2 byte field that is only used in confirmable mode. Thus, this field can be utilized for the message length in the non-confirmable mode. The functionality of this field would then correspond to the ”Content-Length” header of HTTP. In order to ensure DPWS functionality, a minimum set of CoAP options (headers) must be supported by the client and the server. Optional HTTP headers that do not have any impact on functionality of DPWS is omitted during compression. The suggested minimum set of CoAP options is depicted in Table VI. Content-Type option corresponds to the Content-Type header of HTTP. Uri-Host, Uri-Port, and Uri-Path represent altogether the URI. According to the CoAP specification, Uri-Host and Uri-Port options have default values if the option is not present in the message. In this case, Uri-Host and Uri-Port is derived from the destination IP address and the port that is present in the TCP header. In most cases, it would be sufficient to specify only the Content-Type as well as Uri-Path options.

Using CoAP as a binary encoding for HTTP high compression rates can be achieved without increasing the CPU load. However, sole compression of HTTP would not lead to the high compression of DPWS communication overhead. In order to achieve a goal, further compression of XML and SOAP is necessary. For this purpose Efficient XML Interchange (EXI) is suggested.

EXI represents a binary XML encoding. At the time of writing, EXI specification is available in version 1.0 [16]. XML items are represented as events. EXI document consist of the sequence of events. With the aid of additional content that can be associated with events, XML attribute values can be described. In contrast to XML, element values of XML-Schema are encoded respective to their data type. For instance, an integer is represented as a 4 byte value instead of a 10 byte string. EXI has different options to accurately specify a trade-off between compression rate and lossless XML compression. For instance, XML namespace prefixes can be preserved during encoding. This results in an accurate XML transformation but also in a higher data volume. For encoding of XML documents EXI uses grammars. The grammars are used to produce event codes. Events that are most likely to occur are encoded with fewer bits. EXI distinguish two modes: Schema-less and Schema-informed. Schema-less mode is used if no Schema-information is provided. Built-in grammars are used in this case. These grammars are extended with learning mechanisms. Data associated with an event have to be encoded only once. For instance, a closing tag string of XML document will not be encoded again as it was already encoded with a corresponding opening tag. Much better compression rate can be reached in a Schema-informed mode. In this case, possible events are known in advance and thus event codes can be efficiently encoded with less bits. Schema information can be further subdivided into strict and none-strict mode. In the strict mode, an encoding document must exactly match the provided XML-Schema. Any deviation would result in encoding error. However, the length of event codes can be reduced. In the non-strict mode, deviations are allowed and will be encoded with a built-in learning grammars. This mode results in longer event codes compared to the strict mode, however any document can be encoded.

ENTITY	EXI MODE	DATA SIZE	OVERHEAD
SML GetProfileListRequest	n.A.	52 byte	55.8 %
SML request file	n.A.	124 byte	81.5 %
SML GetProfileListResponse	n.A.	104 byte	38.5 %
SML response file	n.A.	176 byte	63.6 %
DPWS request	Schema-less	225 byte	89.7 %
DPWS request	Non-strict	46 byte	50.0 %
DPWS request	Strict	45 byte	48.9 %
DPWS response	Schema-less	358 byte	83.8 %
DPWS response	Non-strict	82 byte	29.3 %
DPWS response	Strict	77 byte	24.7 %

TABLE VII
COMMUNICATION OVERHEAD COMPARISON BETWEEN SML AND DPWS WITH COMPRESSION APPLIED

For further overhead reduction of DPWS, a commonly used Universally Unique Identifier (UUID) for *MessageID* in WS-Addressing was replaced with a short 4 byte identifier. In order to measure the resulting traffic overhead of the suggested DPWS profile, JMEDS stack was extended with CoAP for HTTP and EXI for SOAP compression. For EXI parsing and generation open source implementation EXIficient was used. The achieved results are presented in Table VII. Since the suggested smart metering profile covers a narrow range of applications namely automated meter reading, EXI Schema-informed compression with the strict mode can be utilized.

The detailed compression results are described in Table VIII. As set out above, the suggested approach can reduce communication overhead of DPWS by over 90 %. Thus,

Message	Protocol	Uncompressed	Compressed	Rate
Request	HTTP	160 byte	13 byte	91.9 %
	SOAP	548 byte	32 byte	94.2 %
	DPWS	708 byte	45 byte	93.6 %
Response	HTTP	137 byte	6 byte	95.6 %
	SOAP	864 byte	71 byte	91.8 %
	DPWS	1001 byte	77 byte	92.3 %

TABLE VIII
DPWS COMPRESSION RATES WITH CoAP AND EXI APPLIED

DPWS request and response messages will cause less traffic compared to SML as far as SML files with one request/response message are concerned. Multiple values can still be requested (or sent) with one DPWS message. SML will only cause less communication overhead carrying more than one different message types in one request or response file due to a decreasing overhead share of lower layers (TCP/IP). However, it is less likely to use different request/response methods with one specific equipment. The proposed compression method does not restrict the WS functionality. It can be easily extended with other WS standards.

VI. Security considerations and implementation details

In order to meet high security requirements, the communication between a smart meter and a reading unit must be encrypted with TLS, which has been demanded by BSI. This security measure is already part of the DPWS specification. Moreover, the measurements can be further protected by a signature. For signature generation, the WS-Security standard can be used. In order to enable data encryption, the host equipment must provide enough resources and processing power. The mentioned adapted DPWS JMEDS implementation was tested with enabled TLS encryption on FOX Board G20. The present board has the following specifications: Atmel ARM9 @ 400 MHz CPU, 64MB of RAM, and a 2 GB microSD. As operating system embedded Debian with OpenJDK 1.6 was used. The size of the resulting executable Jar file is 3.7 MB. The running program consumes 19 MB RAM as well as nearly 0 % CPU in idle and ca. 60 % CPU while processing a request. The system consumes 1.7 W power in idle and 1.9 W while processing a request respectively. Most efficient resources consumption can be reached with C implementation as it was already shown with uDPWS. However, this work was focused on DPWS profile for smart metering as well as reduction of WS overhead using existing standard technologies. The proposed approach including compression methods can be easily integrated in every DPWS implementation.

VII. Conclusion and future work

In the proposed work, the smart meter profile for WS based on SML and DPWS was introduced. It was demonstrated that using DPWS-enabled smart meters, a heterogeneous meter network can be built using existing open standards. From a technical perspective, the suggested WS-based solution is not bound to any physical layer technology so that any possible technology supporting IP can be used. This eases the setup of heterogeneous and vendor independent meter networks.

Due to the sensitivity of meter data, energy profiles, and controlling capabilities, special security requirements have to be satisfied. Authorization, encryption, and signature are common security measures. They are already integrated in DPWS, thereby providing required privacy and security for AMR. In order to reduce communication overhead, CoAP is used for HTTP and EXI for XML binary encoding. Thus, traffic overhead is reduced by over 90 %. In the future work, the evaluation results and the comparison with other smart metering protocols will be investigated. The suggested approach can be easily applied to nearly all smart metering protocols.

VIII. Acknowledgment

We would like to thank the German Federal Institute for Research on Building, Urban Affairs and Spatial Development (BBSR) within the Federal Office for Building and Regional Planning for their support in this project. This work is partially granted by BBSR within the scope of Zukunft Bau.

References

[1] OASIS. (2009) Devices profile for web services version 1.1. [Online]. Available: http://docs.oasis-open.org/ws-dd/dpws/wsdd-dpws-1.1-spec.html
[2] A. Weidlich, S. Karnouskos, and J. Ringelstein, "Technology trends for smarthouse/smartgrid," *European Commision*, pp. 1–62, 2010.
[3] S. Rogai. (2007) Telegestore project progress & results. ENEL Distributione S.p.A. [Online]. Available: http://www.ieee-isplc.org/2007/docs/keynotes/rogai.pdf
[4] B. Botte, V. Cannatelli, and S. Rogai, "The telegestore project in enel's metering system," *18th International Conference on Electricity Distribution*, pp. 1–4, June 2005.
[5] J. O'Shaughnessy, M. Barash, and C. Stanfield. (2006) E.on sweden selects echelon's nes smart electricity metering system for 370,000 customers. Echelon Corporation. [Online]. Available: http://www.echelon.com/company/press/2006/eonse.htm
[6] Federal Office for Information Security Germany, *Protection Profile for the Gateway of a Smart Metering System*, 2011.
[7] National Institute of Standards and Technology, *NIST Framework and Roadmap for Smart Grid Interoperability Standards, Release 2.0*, 2012.
[8] SMB Smart Grid Strategic Group, *IEC Smart Grid Standardization Roadmap*. International Electrotechnical Commission, 2010.
[9] S. Karnouskosa and A. Izmaylova, "Simulation of web service enabled smart meters in an event-based infrastructure," *7th IEEE International Conference on Industrial Informatics*, pp. 125–130, 2009.
[10] S. Chan, C. Kaler *et al.*, "Devices profile for web services," *Microsoft Developers Network Library*, pp. 1–39, May 2005.
[11] G. Moritz, E. Zeeb *et al.*, "Device profile for web services and the rest," *8th IEEE International Conference on Industrial Informatics (INDIN)*, pp. 1–8, 2010.
[12] M. Wisy, *Smart Message Language 1.03*. EMSYCON GmbH, 2008.
[13] C. Lerche, N. Laum *et al.*, "Implementing powerful web services for highly resource-constrained devices," *Pervasive Computing and Communication Workshops*, pp. 332–335, March 2011.
[14] Z. Shelby, K. Hartke *et al.*, "Constrained Application Protocol (CoAP)," *Internet-Draft*, 2012. [Online]. Available: http://datatracker.ietf.org/doc/draft-ietf-core-coap
[15] G. Moritz, F. Golatowski, and D. Timmermann, "A lightweight soap over coap transport binding for resource constraint networks," *8th IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, pp. 861–866, October 2011.
[16] J. Schneider and T. Kamiya, "Efficient XML Interchange (EXI) Format 1.0," *W3C Recommendation*, March 2011. [Online]. Available: http://www.w3.org/TR/exi/